

# i.MX31 PDK 1.5 Windows Embedded CE 6.0

## Reference Manual

Part Number: 926-77201  
Rev. 1.5  
2/2009



## **How to Reach Us:**

### **Home Page:**

[www.freescale.com](http://www.freescale.com)

### **Web Support:**

<http://www.freescale.com/support>

### **USA/Europe or Locations Not Listed:**

Freescale Semiconductor, Inc.  
Technical Information Center, EL516  
2100 East Elliot Road  
Tempe, Arizona 85284  
+1-800-521-6274 or  
+1-480-768-2130  
[www.freescale.com/support](http://www.freescale.com/support)

### **Europe, Middle East, and Africa:**

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

### **Japan:**

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

### **Asia/Pacific:**

Freescale Semiconductor China Ltd.  
Exchange Building 23F  
No. 118 Jianguo Road  
Chaoyang District  
Beijing 100022  
China  
+86 010 5879 8000  
[support.asia@freescale.com](mailto:support.asia@freescale.com)

### **For Literature Requests Only:**

Freescale Semiconductor  
Literature Distribution Center  
P.O. Box 5405  
Denver, Colorado 80217  
+1-800 441-2447 or  
+1-303-675-2140  
Fax: +1-303-675-2150  
[LDCForFreescaleSemiconductor@hibbertgroup.com](mailto: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 which 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 or registered trademarks of Freescale Semiconductor, Inc. in the U.S. and other countries. All other product or service names are the property of their respective owners. The Power Architecture and Power.org word marks and the Power and Power.org logos and related marks are trademarks and service marks licensed by Power.org.

© Freescale Semiconductor, Inc., 2008-2009. All rights reserved.



# Contents

Paragraph Number	Title	Page Number
---------------------	-------	----------------

## About This Book

Audience .....	xix
Suggested Reading .....	xix
Conventions .....	xix
Acronyms and Abbreviations .....	xx

## Chapter 1 Introduction

1.1	Getting Started .....	1-1
1.2	SDK System Architecture .....	1-1
1.2.1	Tools and Bootloader .....	1-2
1.2.2	BSP Layer .....	1-2
1.2.3	Middleware and Core OS Service Layer .....	1-2
1.2.4	Application Layer .....	1-2
1.3	Windows Embedded CE 6.0 Architecture .....	1-4

## Chapter 2 ACC Driver

2.1	ACC Driver Summary .....	2-1
2.2	Supported Functionality .....	2-2
2.3	Hardware Operation .....	2-2
2.4	Software Operation .....	2-2
2.4.1	Application / User Interface to ACC drives .....	2-2
2.4.2	ACC Driver Configuration .....	2-2
2.4.3	Loading and Initialization .....	2-2
2.4.4	Mode Selection .....	2-3
2.4.5	G-Level Selection .....	2-3
2.4.6	Output Resolution .....	2-3
2.4.7	Detection Axis .....	2-3
2.4.8	Calibration .....	2-4
2.4.9	Power Management .....	2-4

## Chapter 3 ATA Driver

3.1	ATA Driver Summary .....	3-1
3.2	Requirements .....	3-1

# Contents

Paragraph Number	Title	Page Number
3.3	Hardware Operation.....	3-2
3.3.1	Conflicts with other Peripherals and Catalog Options.....	3-3
3.3.2	Cabling.....	3-3
3.4	Software Operation.....	3-3
3.4.1	Application / User Interface to ATA drives .....	3-3
3.4.2	ATA Driver Configuration .....	3-4
3.4.3	Power Management .....	3-4
3.4.4	Registry Settings .....	3-5
3.4.5	DMA Support .....	3-7
3.5	Unit Test.....	3-7
3.5.1	Unit Test Hardware .....	3-7
3.5.2	Unit Test Software .....	3-8
3.5.3	Building the Storage Device Tests.....	3-8
3.5.4	Running the Storage Device Tests .....	3-8
3.6	Basic Elements for Driver Development .....	3-9
3.6.1	BSP Environment Variables.....	3-10
3.6.2	Mutual Exclusive Drivers .....	3-10
3.6.3	Dependencies of Drivers.....	3-10
3.7	Block Device API Reference .....	3-10
3.7.1	IOCTL_DISK_DEVICE_INFO .....	3-10
3.7.2	IOCTL_DISK_GET_STORAGEID.....	3-11
3.7.3	IOCTL_DISK_GETINFO .....	3-11
3.7.4	IOCTL_DISK_GETNAME.....	3-11
3.7.5	IOCTL_DISK_READ .....	3-12
3.7.6	IOCTL_DISK_SETINFO.....	3-12
3.7.7	IOCTL_DISK_WRITE.....	3-12
3.7.8	IOCTL_DISK_FLUSH_CACHE.....	3-12

## Chapter 4 Audio Driver

4.1	Audio Driver Summary .....	4-1
4.2	Requirements .....	4-2
4.3	Hardware Operation.....	4-3
4.3.1	Audio Playback.....	4-4
4.3.2	Speaker output .....	4-5
4.3.3	Required SoC Peripherals.....	4-5
4.3.4	Conflicts with Other SoC Peripherals.....	4-5
4.3.5	Known Issues.....	4-5
4.3.6	Required MC13783 PMIC Components.....	4-5

# Contents

Paragraph Number	Title	Page Number
4.4	Software Operation .....	4-6
4.4.1	Audio Playback .....	4-6
4.4.2	Audio Recording .....	4-6
4.4.3	Audio Driver Compile-time Configuration Options .....	4-6
4.4.4	DMA Support .....	4-8
4.4.5	Power Management .....	4-10
4.4.6	Audio Driver Registry Settings .....	4-11
4.5	Unit Test .....	4-12
4.5.1	Unit Test Hardware .....	4-12
4.5.2	Unit Test Software .....	4-12
4.5.3	Building the Audio Driver CETK Tests .....	4-13
4.5.4	Running the Audio Driver CETK Tests .....	4-13
4.6	System-level Audio Driver Tests .....	4-13
4.6.1	Checking for a Boot-time Musical Tune .....	4-13
4.6.2	Confirming Touchpanel Taps and Keypad Key Presses .....	4-14
4.6.3	Playing Back Sample Audio and Video Files Using the Media Player .....	4-14
4.6.4	Using the SDK Sample Audio Applications for Testing .....	4-14
4.7	Audio Driver API Reference .....	4-14
4.8	Audio Driver Troubleshooting Guide .....	4-14
4.8.1	Checking Build-time Configuration Options .....	4-14
4.8.2	Confirming Audio Driver Loading During Device Boot .....	4-15
4.8.3	Media Player Application Not Found .....	4-15
4.8.4	Media Player Fails to Load and Play an Audio File .....	4-15

## Chapter 5 Backlight Driver

5.1	Backlight Driver Summary .....	5-1
5.2	Requirements .....	5-1
5.3	Hardware Operation .....	5-2
5.4	Software Operation .....	5-2
5.4.1	Backlight Driver Registry Settings .....	5-2
5.5	Unit Test .....	5-2
5.5.1	Unit Test Hardware .....	5-3
5.5.2	Unit Test Software .....	5-3
5.5.3	Running the Backlight Application Test .....	5-3
5.6	Backlight API Reference .....	5-4

# Contents

Paragraph Number	Title	Page Number
<b>Chapter 6</b>		
<b>Battery Driver</b>		
6.1	Battery Driver Summary .....	6-1
6.2	Requirements .....	6-1
6.3	Hardware Operation .....	6-1
6.3.1	Conflicts with other SoC Peripherals .....	6-2
6.4	Software Operation .....	6-2
6.4.1	Battery Driver Registry Settings .....	6-2
6.4.2	Power Management .....	6-2
6.5	Unit Test .....	6-2
6.5.1	Unit Test Hardware .....	6-3
6.6	Battery API Reference .....	6-3
6.6.1	Battery PDD Functions .....	6-3
6.6.2	Battery Driver Structures .....	6-4

## **Chapter 7** **Bluetooth Driver**

7.1	Bluetooth Driver Summary .....	7-1
7.2	Supported Functionality .....	7-2
7.3	Hardware Operation .....	7-2
7.3.1	Conflicts with Other Peripherals and Catalog Items .....	7-3
7.4	Software Operation .....	7-3
7.4.1	Registry Settings .....	7-4
7.5	Unit Test .....	7-4
7.5.1	Unit Test Hardware .....	7-4
7.5.2	Unit Test Software .....	7-4
7.5.3	Running the Unit Tests .....	7-4
7.5.4	Operation Attention Items and Tips .....	7-7
7.5.5	Known Issues .....	7-8

## **Chapter 8** **Camera Driver**

8.1	Camera Driver Summary .....	8-1
8.2	Supported Functionality .....	8-1
8.3	Hardware Operation .....	8-2

# Contents

Paragraph Number	Title	Page Number
8.4	Software Operation .....	8-2
8.4.1	Communicating with the Camera .....	8-2
8.4.2	Camera Registry Settings.....	8-2
8.4.3	Power Management .....	8-3
8.5	Unit Test.....	8-4
8.5.1	Unit Test Hardware .....	8-4
8.5.2	Unit Test Software .....	8-4
8.5.3	Building the Camera Tests .....	8-5
8.5.4	Running the Camera Tests .....	8-6
8.6	Camera Driver API Reference .....	8-6

## Chapter 9 Chip Support Package Driver Development Kit (CSPDDK)

9.1	CSPDDK Driver Summary .....	9-1
9.2	Supported Functionality .....	9-1
9.3	Hardware Operation.....	9-2
9.3.1	Conflicts with Other Peripherals.....	9-2
9.4	Software Operation .....	9-2
9.4.1	Communicating with the CSPDDK .....	9-2
9.4.2	Compile-Time Configuration Options .....	9-2
9.4.3	Registry Settings .....	9-3
9.4.4	Power Management .....	9-3
9.5	CSPDDK DLL Reference .....	9-3
9.5.1	CSPDDK DLL System Clocking (DDK_CLK) Reference .....	9-3
9.5.2	CSPDDK DLL GPIO (DDK_GPIO) Reference .....	9-6
9.5.3	CSPDDK DLL IOMUX (DDK_IOMUX) Reference .....	9-10
9.5.4	CSPDDK DLL SDMA (DDK_SDMA) Reference .....	9-14

## Chapter 10 Display Driver

10.1	Display Driver Summary .....	10-1
10.2	Supported Functionality .....	10-1
10.3	Hardware Operation.....	10-2
10.3.1	Rotation Control .....	10-2
10.3.2	TV Output Mode.....	10-2
10.4	Software Operation .....	10-3
10.4.1	Communicating with the Display .....	10-3
10.4.2	Configuring the Display.....	10-4
10.4.3	Power Management .....	10-5

# Contents

Paragraph Number	Title	Page Number
10.5	Unit Test.....	10-5
10.5.1	Unit Test Hardware .....	10-6
10.5.2	Unit Test Software .....	10-6
10.5.3	Building the Display Tests .....	10-7
10.5.4	Running the Display Tests .....	10-7
10.6	Display Driver API Reference .....	10-7

## Chapter 11 Dynamic Voltage and Frequency Control (DVFC) Driver

11.1	DVFC Driver Summary .....	11-1
11.2	Supported Functionality .....	11-1
11.3	Hardware Operation .....	11-2
11.3.1	Pin Settings and Conflicts .....	11-2
11.4	Software Operation .....	11-2
11.4.1	Loading and Initialization .....	11-2
11.4.2	Clock Tree Dependency .....	11-2
11.4.3	Processor Workload Tracking .....	11-2
11.4.4	Setpoint Consideration .....	11-3
11.4.5	Lock and Performance .....	11-3
11.4.6	DDK Interface .....	11-3
11.4.7	Power Management .....	11-3
11.5	Unit Test .....	11-4

## Chapter 12 FM Radio Driver

12.1	Radio Driver Summary .....	12-1
12.2	Supported Functionality .....	12-1
12.3	Hardware Operation .....	12-1
12.4	Software Operation .....	12-1
12.4.1	Radio Driver Registry Settings .....	12-2
12.4.2	Power Management .....	12-2
12.5	Unit Test .....	12-2
12.5.1	Unit Test Hardware .....	12-3
12.5.2	Building the Radio Tests .....	12-3
12.5.3	Running the Radio Tests .....	12-3
12.6	Radio IOCTL Reference .....	12-3
12.6.1	Radio Driver IOCTLS .....	12-3
12.6.2	Radio Driver Structures .....	12-6



# Contents

Paragraph Number	Title	Page Number
------------------	-------	-------------

## Chapter 13 General Purpose Timer (GPT) Driver

13.1	GPT Driver Summary .....	13-1
13.2	Supported Functionality .....	13-1
13.3	Hardware Operation .....	13-1
13.3.1	Conflicts with Other Peripherals .....	13-2
13.4	Software Operation .....	13-2
13.4.1	Communicating with the GPT .....	13-2
13.4.2	Creating a Handle to the GPT .....	13-2
13.4.3	Configuring the GPT .....	13-2
13.4.4	Write Operations .....	13-3
13.4.5	Closing the Handle to the GPT .....	13-3
13.4.6	Power Management .....	13-4
13.4.7	GPT Registry Settings .....	13-4
13.5	Unit Test .....	13-4
13.5.1	Unit Test Hardware .....	13-4
13.5.2	Unit Test Software .....	13-5
13.5.3	Building the GPT Tests .....	13-5
13.5.4	Running the GPT Tests .....	13-5
13.6	GPT Driver API Reference .....	13-6
13.6.1	GPT Driver Functions .....	13-6
13.6.2	GPT Driver Structures .....	13-9

## Chapter 14 Global Positioning System Driver

14.1	GPS Driver Summary .....	14-1
14.1.1	Application layer .....	14-2
14.1.2	GPS Core Driver Layer .....	14-3
14.1.3	GPS HAL driver layer .....	14-3
14.2	Supported Functionality .....	14-3
14.3	Hardware Operation .....	14-3
14.3.1	UART Port .....	14-3
14.3.2	GPIO Control .....	14-3
14.3.3	Conflicts with Other Peripherals .....	14-4
14.4	Software Operation .....	14-4
14.4.1	Communicating with the GPS Module .....	14-4
14.4.2	Power Management .....	14-4
14.4.3	GPS Driver Registry Settings .....	14-4
14.5	Unit Test .....	14-4

# Contents

Paragraph Number	Title	Page Number
------------------	-------	-------------

## Chapter 15 Inter-Integrated Circuit (I2C) Driver

15.1	I2C Driver Summary .....	15-1
15.2	Requirements .....	15-1
15.3	Hardware Operation.....	15-1
15.3.1	Conflicts with other SoC peripherals.....	15-2
15.4	Software Operation .....	15-2
15.4.1	Communicating with the I2C.....	15-2
15.4.2	Creating a Handle to the I2C .....	15-2
15.4.3	Configuring the I2C.....	15-3
15.4.4	Data Transfer Operations .....	15-4
15.4.5	Closing the Handle to the I2C.....	15-6
15.4.6	Power Management .....	15-6
15.4.7	I2C Registry Settings.....	15-6
15.5	Unit Test.....	15-7
15.6	I2C Driver API Reference .....	15-7
15.6.1	I2C Driver IOCTLs .....	15-7
15.6.2	I2C Driver Macros .....	15-9
15.6.3	I2C Driver Structures .....	15-13

## Chapter 16 Keypad Driver

16.1	Keypad Driver Summary .....	16-1
16.2	Requirements .....	16-1
16.3	Hardware Operation.....	16-1
16.3.1	The Keypad.....	16-2
16.3.2	Conflicts with other SoC peripherals.....	16-2
16.4	Software Operation .....	16-2
16.4.1	Keypad Scan Codes and Virtual Keys .....	16-3
16.4.2	Power Management .....	16-3
16.4.3	Keypad Registry Settings.....	16-4
16.5	Unit Test.....	16-4
16.5.1	Unit Test Hardware .....	16-4
16.5.2	Unit Test Software .....	16-4
16.5.3	Building the Keyboard Tests.....	16-5
16.5.4	Running the Keyboard Tests.....	16-5
16.6	Keypad Driver API Reference .....	16-5
16.6.1	Keypad PDD Functions .....	16-5

# Contents

Paragraph Number	Title	Page Number
---------------------	-------	----------------

## Chapter 17 LAN9217 Product Ethernet Driver

17.1	LAN9217 Product Ethernet Driver Summary .....	17-1
17.2	Requirements .....	17-1
17.3	Hardware Operation.....	17-2
17.3.1	Conflicts with other SoC peripherals.....	17-2
17.4	Software Operation .....	17-2
17.4.1	Power Management .....	17-2
17.4.2	Product Ethernet Registry Settings .....	17-2
17.5	Unit Test.....	17-3
17.5.1	Unit Test Hardware .....	17-3
17.5.2	Unit Test Software .....	17-3
17.5.3	Building the LAN9217 Product Ethernet Tests .....	17-4
17.5.4	Running the LAN9217 Product Ethernet Tests .....	17-5
17.6	LAN9217 Product Ethernet Driver API Reference .....	17-6

## Chapter 18 MBX Direct3D Mobile/OpenGL ES Drivers

18.1	Direct3D Mobile/OpenGL ES Drivers Summary .....	18-1
18.2	Supported Functionality .....	18-2
18.3	Hardware Operation.....	18-2
18.3.1	Conflicts with other Peripherals .....	18-2
18.4	Software Operation .....	18-2
18.4.1	Application / User Interface to MBX Drivers .....	18-2
18.4.2	Configuring the LCD Display Panels .....	18-3
18.4.3	Float Pointing Acceleration using the ARM VFP Library .....	18-4
18.5	Unit Test.....	18-5
18.5.1	Unit Test Hardware .....	18-5
18.5.2	Unit Test Software .....	18-5
18.5.3	Building the Direct3D Mobile Tests .....	18-6
18.5.4	Running the Direct3D Mobile Tests .....	18-6
18.5.5	Direct3D Mobile/OpenGL ES Application Samples/Demos .....	18-6
18.5.6	Direct3D Mobile Application Samples.....	18-6
18.5.7	Known Issues for MBX CE6 Driver.....	18-7
18.6	Drivers API Reference .....	18-7
18.6.1	Direct3D Mobile .....	18-7
18.6.2	OpenGL ES.....	18-8

# Contents

Paragraph Number	Title	Page Number
<b>Chapter 19</b>		
<b>NAND Flash Media Driver (FMD)</b>		
19.1	NAND FMD Summary .....	19-1
19.2	Requirements .....	19-1
19.2.1	Conflicts with other SoC peripherals .....	19-2
19.3	Software Operation .....	19-2
19.3.1	Compile-Time Configuration Options .....	19-2
19.3.2	Registry Settings .....	19-2
19.3.3	DMA Support .....	19-2
19.3.4	Power Management .....	19-2
19.4	Unit Test .....	19-3
19.4.1	CETK Testing .....	19-3
19.4.2	System Testing .....	19-4
<b>Chapter 20</b>		
<b>Postfilter Driver</b>		
20.1	Postfilter Driver Summary .....	20-1
20.2	Requirements .....	20-1
20.3	Hardware Operation .....	20-2
20.3.1	Conflicts with other SoC peripherals .....	20-2
20.4	Software Operation .....	20-2
20.4.1	Communicating with the Postfilter Driver .....	20-2
20.4.2	Creating a Handle to the Postfilter Driver .....	20-2
20.4.3	Configuring the Postfilter Driver .....	20-2
20.4.4	Executing Postfilter Operations .....	20-3
20.4.5	Closing the Handle to the Postfilter Driver .....	20-4
20.4.6	Postfilter Registry Settings .....	20-4
20.4.7	Power Management .....	20-4
20.5	Unit Test .....	20-5
20.5.1	Unit Test Software .....	20-5
20.5.2	Building the Postfilter Tests .....	20-5
20.5.3	Running the Postfilter Tests .....	20-5
20.6	Postfilter Driver API Reference .....	20-6
20.6.1	Postfilter Driver Functions .....	20-6
20.6.2	PF Driver Enumerations .....	20-10
20.6.3	PF Driver Structures .....	20-11

# Contents

Paragraph Number	Title	Page Number
<b>Chapter 21</b>		
<b>Power Management IC (PMIC)</b>		
21.1	PMIC Driver Summary .....	21-1
21.2	Requirements .....	21-1
21.2.1	PMIC API Framework.....	21-2
21.3	Hardware Operation.....	21-3
21.3.1	MX31 Peripheral Conflicts .....	21-3
21.4	Software Operation .....	21-3
21.4.1	Configuring the PMIC .....	21-3
21.4.2	Creating a Handle to the PMIC.....	21-3
21.4.3	Write Operations .....	21-4
21.4.4	Read Operations.....	21-4
21.4.5	Closing the Handle to the PMIC.....	21-4
21.4.6	Power Management .....	21-4
21.4.7	PMIC Registry Settings .....	21-5
21.4.8	A/D Converter and Touch.....	21-5
21.5	Unit Test.....	21-8
21.5.1	Unit Test Hardware .....	21-8
21.5.2	Unit Test Software .....	21-8
21.5.3	Building the PMIC Tests.....	21-9
21.5.4	Running the PMIC Tests .....	21-9
21.6	PMIC Reference API .....	21-10
21.6.1	PMIC Driver IOCTLs .....	21-10
21.6.2	Interrupt Handling.....	21-12
21.6.3	Register Access API .....	21-18
21.6.4	Power Control Reference.....	21-19
21.6.5	PowerCutTimer Functions .....	21-29
21.6.6	Memory Hold Operation functions .....	21-30
21.6.7	Power Cut Counter Functions.....	21-32
21.6.8	Power Management .....	21-33
21.6.9	Voltage Regulator .....	21-33

# Contents

Paragraph Number	Title	Page Number
21.6.10	Data Structures .....	21-33
21.6.11	Switch mode regulator API's .....	21-37
21.6.12	Linear Voltage Regulator API's .....	21-42
21.6.13	Power Management .....	21-44
21.6.14	Battery Charger .....	21-44
21.6.15	Data Structures .....	21-44
21.6.16	Battery Charger API (Compatible with SC55112 API) .....	21-44
21.6.17	Battery Charger API (MC13783 Native For Compatibility with SC55112) .....	21-47
21.6.18	Battery Charger API (MC13783 Native) .....	21-48
21.6.19	Power Management .....	21-54

## Chapter 22 Power Manager

22.1	Power Manager Summary .....	22-1
22.2	Requirements .....	22-1
22.3	Hardware Operation .....	22-1
22.4	3-Stack Software Operation .....	22-1
22.4.1	Power Management .....	22-2
22.4.2	Image Configuration .....	22-2
22.4.3	Registry Settings .....	22-3
22.5	Unit Test .....	22-4
22.6	Power Manager API Reference .....	22-4
22.6.1	Application Interface .....	22-4
22.6.2	Device Driver Interface .....	22-5

## Chapter 23 Secure Digital Host Controller Driver

23.1	SDHC Driver Summary .....	23-1
23.2	Supported Functionality .....	23-1
23.3	Hardware Operation .....	23-1
23.3.1	Conflicts with Other Peripherals .....	23-2
23.4	Software Operation .....	23-2
23.4.1	Required Catalog Items .....	23-2
23.4.2	SDHC Registry Settings .....	23-2
23.4.3	DMA Support .....	23-3
23.4.4	Power Management .....	23-3

# Contents

Paragraph Number	Title	Page Number
23.5	Unit Test.....	23-4
23.5.1	Unit Test Hardware .....	23-4
23.5.2	Unit Test Software .....	23-5
23.5.3	Building the Tests .....	23-5
23.5.4	Running the Tests.....	23-5
23.5.5	System Testing.....	23-6
23.6	Secure Digital Card Driver API Reference.....	23-6

## Chapter 24 Serial Driver

24.1	Serial Driver Summary .....	24-1
24.2	Supported Functionality.....	24-1
24.3	Hardware Operation.....	24-2
24.3.1	Conflicts with Other Peripherals.....	24-2
24.4	Software Operation .....	24-2
24.4.1	Serial Registry Settings.....	24-2
24.4.2	DMA Support .....	24-3
24.5	Unit Test.....	24-4
24.5.1	Unit Test Hardware .....	24-4
24.5.2	Unit Test Software .....	24-4
24.5.3	Building the Serial Port Driver Tests .....	24-4
24.5.4	Running the Serial Port Driver Test.....	24-4
24.6	Serial Driver API Reference .....	24-5
24.6.1	Serial PDD Functions .....	24-6
24.6.2	Serial Driver Macros.....	24-7
24.6.3	Serial Driver Structures .....	24-7

## Chapter 25 Touch Panel Driver

25.1	Touch Panel Driver Summary.....	25-1
25.2	Supported Functionality.....	25-1
25.3	Hardware Operations .....	25-1
25.3.1	Conflicts with Peripherals.....	25-2
25.3.2	Conflicts with i.MX31 3-Stack .....	25-2
25.4	Software Operation .....	25-2
25.4.1	Touch Driver Registry Settings.....	25-2

# Contents

Paragraph Number	Title	Page Number
25.5	Unit Tests .....	25-3
25.5.1	Unit Test Hardware .....	25-3
25.5.2	Unit Test Software .....	25-3
25.5.3	Building the Touch Panel Tests.....	25-4
25.6	Touch Panel API Reference .....	25-4

## Chapter 26 USB Boot and KITL

26.1	USB Boot and KITL Summary .....	26-1
26.2	Supported Functionality.....	26-1
26.3	Hardware Operation.....	26-2
26.3.1	Conflicts with Other Peripherals.....	26-2
26.4	Software Operation .....	26-2
26.4.1	Software Architecture .....	26-2
26.4.2	Source Code Layout.....	26-3
26.4.3	IOMUX and Pinout.....	26-3
26.4.4	Power Management .....	26-3
26.4.5	Registry Settings .....	26-3
26.4.6	DMA Support .....	26-3
26.5	Unit Test.....	26-4
26.5.1	Building the USB Boot and KITL .....	26-4
26.5.2	Testing USB Boot and KITL .....	26-4

## Chapter 27 USB OTG Driver

27.1	USB OTG Driver Summary .....	27-1
27.1.1	OTG Client Driver Summary .....	27-1
27.1.2	OTG Host Driver Summary.....	27-2
27.1.3	OTG Transceiver Driver Summary (For HIGH-SPEED only).....	27-2
27.2	Supported Functionality.....	27-3
27.3	Hardware Operation.....	27-4
27.3.1	Conflicts with Other Peripherals.....	27-4
27.3.2	Signal Quality Requirement.....	27-4



# Contents

Paragraph Number	Title	Page Number
27.4	Software Operation .....	27-4
27.4.1	USB OTG Host Controller Driver .....	27-4
27.4.2	USB Client Driver .....	27-13
27.4.3	USB Transceiver Driver (ID Pin Detect Driver -- XCVR).....	27-18
27.4.4	Power Management .....	27-22
27.4.5	Function Drivers .....	27-25
27.4.6	Class Drivers.....	27-28
27.5	IRAM Patch .....	27-30
27.6	Basic Elements for Driver Development .....	27-30
27.6.1	BSP Environment Variables.....	27-30
27.6.2	Dependencies of Drivers.....	27-31

## Chapter 28 WLAN Driver

28.1	WLAN Driver Summary .....	28-1
28.2	Supported Functionality.....	28-2
28.3	Hardware Operation.....	28-2
28.3.1	Conflicts with Other Peripherals.....	28-2
28.4	Software Operation .....	28-2
28.4.1	Wi-Fi Registry setting.....	28-3
28.5	Unit Test.....	28-4
28.5.1	Unit Test Hardware .....	28-4
28.5.2	Unit Test Software .....	28-5
28.5.3	Running the WLAN Driver Tests .....	28-5
28.5.4	Test the WLAN Communication without Protection .....	28-6

## Appendix A Frequently Asked Questions

A.1	How to Deal with Different Resolutions of the Display Panel? .....	A-1
A.2	How to Deal with Different Display Interface Formats?.....	A-1



# Contents

Paragraph Number	Title	Page Number
---------------------	-------	----------------

# About This Book

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

## Audience

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

## Suggested Reading

Freescale documentation is available from the sources listed on the back cover of this manual.

- *Microsoft Windows Embedded CE Help* can be viewed from the Microsoft Developer Network at <http://msdn.microsoft.com>. Visit the web site and search for the string, "Windows Embedded CE."
- <http://msdn.microsoft.com/embedded/windowsce>
- *i.MX31 PDK Hardware User's Guide*
- *i.MX31 Applications Processor IC Reference Manual*
- *i.MX31 PDK Windows Embedded CE 6.0 Release Notes*
- *i.MX31 PDK Windows Embedded CE 6.0 User's Guide*
- *Windows Embedded CE 6.0 BSP Reference Guide (RTM14)*
- *Visual Studio 2005 Help*

The Freescale manuals can also be found at <http://www.freescale.com>. The manuals can be downloaded directly from the web, or you can also order the printed copies. The Freescale manuals may also be provided with your PDK.

## Conventions

This document uses the following conventions:

- `Courier` is used to identify commands, explicit command parameters, code examples, expressions, data types, and directives.
- **Bold** indicates the menu options or buttons the user can select. Cascaded menu options are delimited with the → symbol.
- *Italic* is used for emphasis, to identify new terms, and for replaceable command parameters.

# Acronyms and Abbreviations

Table i contains acronyms and abbreviations used in this document.

**Table i. Acronyms and Abbreviated Terms**

Term	Meaning
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
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
IPU	Image processing unit
IST	Interrupt service thread
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

**Table i. Acronyms and Abbreviated Terms**

<b>Term</b>	<b>Meaning</b>
PWM	Pulse-width modulation
SD	Secure digital cards
SDC	Synchronous display controller
SDHC	Secure digital host controller
SDIO	Secure digital I/O and combo cards
SDK	Software development kit
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



# Chapter 1

## Introduction

This Freescale i.MX31 PDK Windows Embedded CE 6.0 based product development kit (PDK) helps speed development of applications for the Freescale i.MX31 3-Stack multimedia applications processor. It includes the following:

- Software development kit (SDK), which includes tools, BSP, codecs, basic middleware, and applications
- Hardware board (i.MX31 3-Stack board)
- Documentation

This kit supports the Microsoft Windows® Embedded CE 6.0 operating system, and requires the use of Microsoft Platform Builder, which is an integrated development environment (IDE) for building customized embedded OS designs. To view feature information, see the *i.MX31 PDK Release Notes*.

### NOTE

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

- To view the Platform Builder Help, click **Help** from within the Platform Builder application.
- To view the *Platform Builder User Guide*, visit:  
<http://msdn2.microsoft.com/en-us/library/aa448606.aspx>

## 1.1 Getting Started

For instructions on installing the SDK, and on building, downloading, and running the OS image on the hardware board, refer to the *i.MX31 PDK 1.5 Windows Embedded CE 6.0 User's Guide* included with this distribution.

## 1.2 SDK System Architecture

Figure 1-1 shows the SDK system, which consists of tools, bootloader, BSP layer, a middleware, and core OS service layer, and an application layer.

## 1.2.1 Tools and Bootloader

The PDK tools and boot loader consist of the following components.

Tools	Flashing tool: support image download and flashing from UART
EBOOT	

## 1.2.2 BSP Layer

The BSP layer contains the following components.

ATA	Display LCD	GPT	Serial
Audio DAC	Display TV-Out	I2C	SDHC
Backlight	DVFC	KPP	Touch Panel
Battery	Ethernet	MBX D3DM&OpenGL	USB
Camera	FM Radio	NAND FMD	WiFi
DDK	GPS	PMIC	

## 1.2.3 Middleware and Core OS Service Layer

The middleware and core OS service layer contains the Power Management components.

## 1.2.4 Application Layer

The application layer contains the following components.

FM Radio application	TV-Out application
Camera application	3D Demo Application



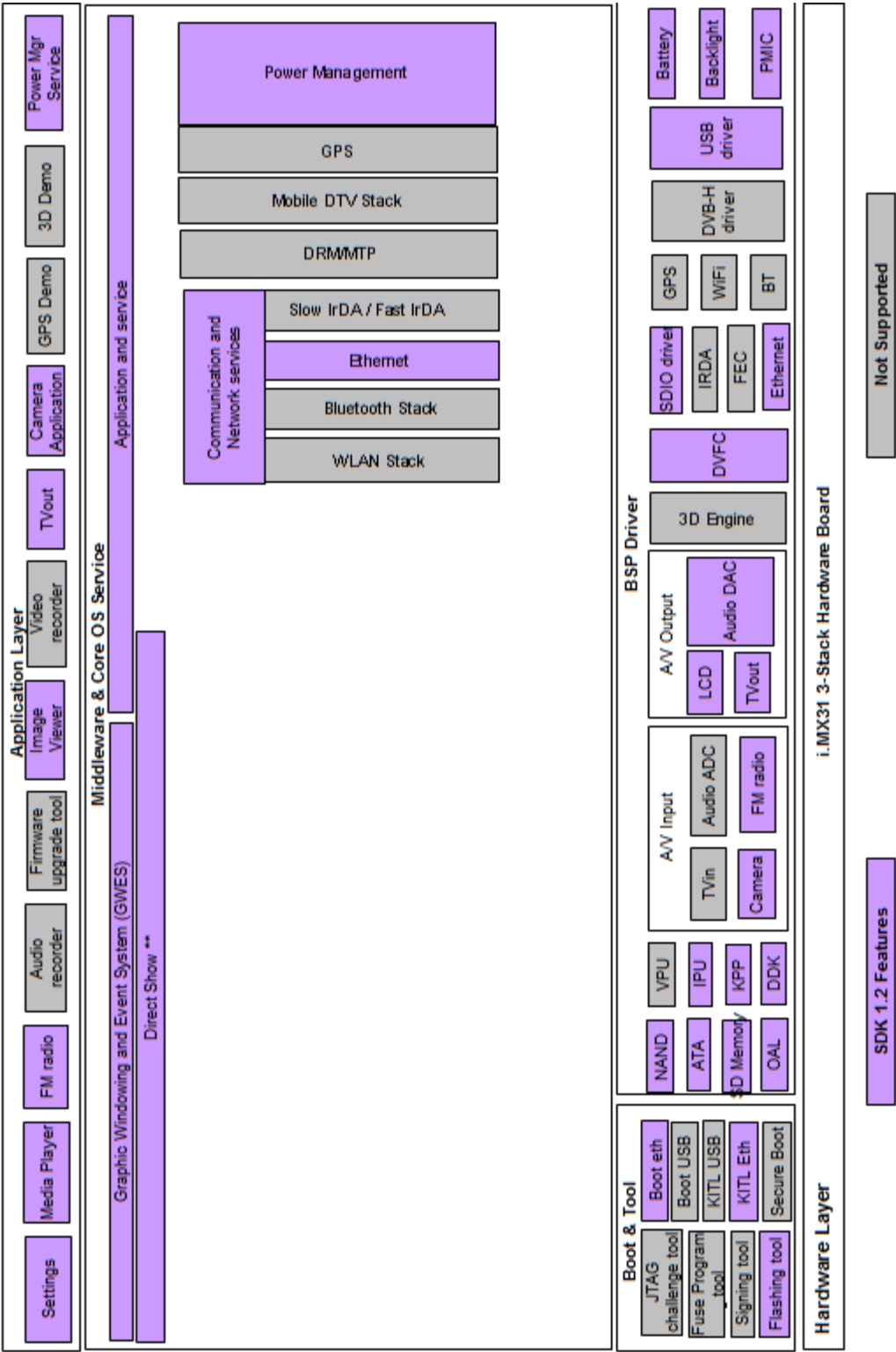


Figure 1-1. WSDK System Diagram

## 1.3 Windows Embedded CE 6.0 Architecture

The Windows Embedded CE 6.0 architecture is a variation of Microsoft's Windows operating system for minimalistic computers and embedded systems. The architecture of the operating system and sub-systems (for example, power management and DirectDraw) are described in several locations in the Help. You may want to begin at the following location in Help:

**Welcome to Windows Embedded CE 6.0 > Windows Embedded CE Architecture**

## Chapter 2

# ACC Driver

The accelerometer (ACC) driver is used as the lower layer for the ACC algorithm layer. The main purpose of the ACC driver is to provide I2C interface for register access, and act as an interrupt process on the MMA7450L accelerometer. The ACC driver is constructed as a stream interface driver that exposes I/O control codes (IOCTL\_ACC\_XXX).

## 2.1 ACC Driver Summary

The following table identifies the source code location, library dependencies, and other BSP information.

Driver Attribute	Definition
Target Platform (TGTPLAT)	IMX313DS
Target SOC (TGTSOC)	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<tgtpplat>\SRC\DRIVERS\ACCELEROMETER
Import Library	(cspddk.lib)
Driver DLL	mma.dll
Catalog Item	Third Party → BSPs → Freescale i.MX313DS:ARMV4I → Device Drivers → Accelerometer
SYSGEN Dependency	
BSP Environment Variable	BSP_ACC

## 2.2 Supported Functionality

The ACC driver enables the MMA7450L sensor on the 3-Stack board to provide the following software and hardware support:

- Flexibility to select STANDBY, MEASUREMENT, LEVEL DETECTION or PULSE DETECTION mode for multifunctional applications
- Flexibility to select 2g, 4g or 8g of acceleration for multifunctional applications
- Flexibility to select an output resolution of 8-bit or 10-bit
- Flexibility to select detection axis, X-axis, Y-axis, Z-axis or arbitrary combination
- Offset calibration

## 2.3 Hardware Operation

For operation and programming information, see the *MCIMX31 and MCIMX31L Applications Processors Reference Manual*.

## 2.4 Software Operation

### 2.4.1 Application / User Interface to ACC drives

The ACC device exports a standard streams interface to the Application/User, and can be accessed using functions such as `CreateFile()` and `CloseHandle()`.

### 2.4.2 ACC Driver Configuration

You configure the driver into the BSP build by dragging the catalog item. Doing so defines the environment variable/configuration option: `BSP_ACC`.

### 2.4.3 Loading and Initialization

The ACC driver is loaded by the device manager in the kernel space. As part of the stream driver loading procedure, the device manager invokes the corresponding stream initialization function exported by the ACC driver. The initialization sequence includes a call to platform-specific code (`BSPEnableACC`) to power on the sensor and set the IOMUX configuration.

## 2.4.4 Mode Selection

The following table describes the four available modes.

Name	Configuration Setting Name	Description
STANDBY	ACC_MODE_STANDBY	The device outputs are turned off, providing significant reduction of operating current.
MEASUREMENT	ACC_MODE_MEASUREMENT	During measurement mode, continuous measurements on all three axes enabled.
LEVEL DETECTION	ACC_MODE_LEVEDETECTION	In level detection mode, the measurements for x, y and z are all enabled with 2g/4g and 8g range available. The detection of thresholds for an acceleration signal level for the combinations of x and y or x, y, and z is enabled.
PULSE DETECTION	ACC_MODE_PULSEDETECTION	In pulse detection mode, both 2g/4g and 8g range are available. Measurements for x, y and z in 2g/4g or 8g mode are enabled. The level detection is also enabled in this mode. The pulse detected by the acceleration signal is enabled with single pulse and double pulse detection allowing the choice of either positive, negative, or absolute value pulse detection.

## 2.4.5 G-Level Selection

The following table describes the four G-Level options.

Name	Configuration Setting	g-Range	Sensitivity
8G(10bit)	ACC_GSEL_8G	-8G~~8G	64 LSB/g
8G(8bit)	ACC_GSEL_8G	-8G~~8G	16 LSB/g
4G	ACC_GSEL_4G	-4G~~4G	32 LSB/g
2G	ACC_GSEL_2G	-2G~~2G	64 LSB/g

## 2.4.6 Output Resolution

There are two available output resolutions.

Resolution	Configuration Setting
8-bit	ACC_GSEL_8G, ACC_GSEL_4G, ACC_GSEL_2G
10-bit	ACC_GSEL_8G

## 2.4.7 Detection Axis

The X, Y and Z axis detection can be enabled separately and combined in any way.

## 2.4.8 Calibration

The offsets of the X, Y and Z axes can be assigned to calibrate the output.

## 2.4.9 Power Management

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

### 2.4.9.1 PowerUp

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

### 2.4.9.2 PowerDown

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

### 2.4.9.3 IOCTL\_POWER\_CAPABILITIES

The D0 or D4 device power states are supported.

### 2.4.9.4 IOCTL\_POWER\_SET

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

### 2.4.9.5 IOCTL\_POWER\_GET

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

## Chapter 3

# ATA Driver

The ATA driver in Windows Embedded CE 6.0 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.

### 3.1 ATA Driver Summary

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

**Table 3-1. ATA Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	IMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\FREESCALE\<TGTSOC>\ATA
CSP Static Library	ata_<TGTSOC>.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\BLOCK\ATA
Import Library	(cspddk.lib)
Driver DLL	ata_mx31.dll
Catalog Item	Third Party → BSP → Freescale i.MX31 3DS: ARMV4I → Storage Drivers → ATA device driver
SYSGEN Dependency	SYSGEN_MSPART,SYSGEN_FATFS,SYSGEN_EXFAT
BSP Environment Variable	BSP_ATA

### 3.2 Requirements

The ATA driver must meet/support the following requirements:

- Provide standard Microsoft Block Storage Device API, including disk information management, formatting, block data read/write with full scatter-gather buffer support
- Support two power management modes, full on and full off
- Support standard bus timing mode for UDMA mode 3 (optional support other modes such as PIO modes 0-4, MDMA modes 0-2, and UDMA modes 0-2 & 4).
- Support full sustained (media) data throughput capacity of Hitachi TravelStar C4K40 (or equivalent) at UDMA mode 3.
- Support full sustained (media) data throughput capacity of SST NANDrive (or equivalent) at UDMA mode 3.

**NOTE**

UDMA5 mode requires 80MHz bus clock or above, so this mode may not apply to MX31 which bus clock is 66.5MHz.

**NOTE**

If SST NANDdrive is set to be write-protected through a jumper or host logic, the ATA driver will output message “WARNING: NANDrive write protected!!!”.

### 3.3 Hardware Operation

For operation and programming information, see the chapter on the Advanced Technology Attachment (ATA) in the MCIMX31 and MCIMX31L Applications Processors Reference Manual.

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

- CPU programmed data transfers through 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, through the MX31's ATA peripheral data FIFO.

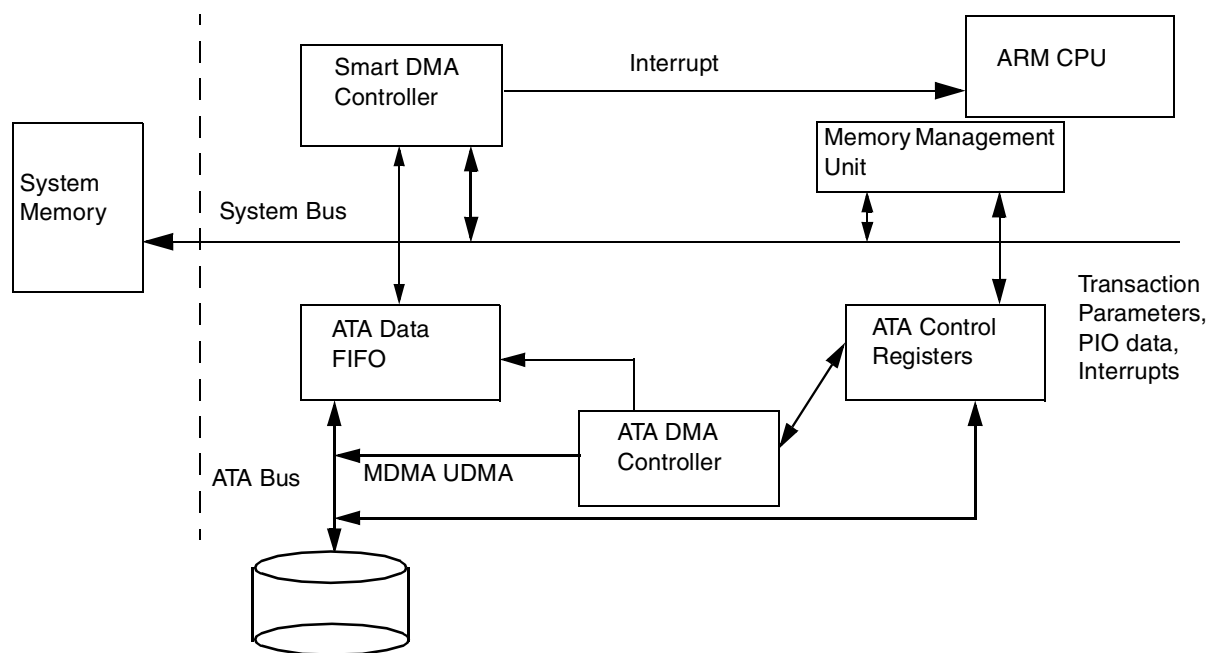


Figure 3-1. ATA Hardware Block Diagram



The MX31 also contains 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.

### **3.3.1 Conflicts with other Peripherals and Catalog Options**

#### **3.3.1.1 Conflicts with SoC Peripherals**

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

#### **3.3.1.2 Conflicts on 3-Stack board**

Because the CSPI1 device, the USB Host 1 port, and the PWM module are not supported on the 3-Stack board, the ATA as implemented for the 3-Stack board has no pin conflicts with the CSPI device (CSPI1), USB Host 1 port, and PWM.

### **3.3.2 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.

## **3.4 Software Operation**

### **3.4.1 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 through 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 Embedded 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.

## 3.4.2 ATA Driver Configuration

The driver is configured into the BSP build by dragging & dropping the catalog item as defined in [Section 3.1, “ATA Driver Summary.”](#) 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.

### 3.4.2.1 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 3.4.4, “Registry Settings.”](#)

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

### 3.4.2.2 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.

### 3.4.2.3 IOMUX and Pinout

The internal MX31 ATA signals can be multiplexed to a choice of pins on IC, as described for the IOMUX in the hardware reference manual.

### 3.4.2.4 Defaults

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

## 3.4.3 Power Management

The ATA supports two power management modes, ON (D0) and OFF (D4). These modes are managed through 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.

### 3.4.3.1 PowerUp

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

### 3.4.3.2 PowerDown

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

### 3.4.3.3 IOCTL\_POWER\_SET

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

## 3.4.4 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]
```

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.

**Table 3-2. ATA Driver Registry Key Values**

Value	Type	Content	Description
Dll	sz	ata_mx31.dll	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
		22	MDMA mode 2
		40	UDMA mode 0
		...	...

Value	Type	Content	Description
		45	UDMA mode 5
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:43 ; 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 indicated below.

**Table 3-3. ATA Standard Registry Values**

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
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
```

### 3.4.5 DMA Support

The ATA driver supports DMA mode and non-DMA mode of transfer. The driver always defaults to DMA mode of transfer. ATA supports three transfer-types: UDMA, MDMA and PIO modes. PIO mode works in non-DMA mode of operation while other modes work in DMA mode. To change the mode of transfer, change the value of “TransferMode” from the registry. When the 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.

## 3.5 Unit Test

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

### 3.5.1 Unit Test Hardware

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

**Table 3-4. ATA Driver Unit Test Hardware Requirements**

Requirements	Description
i.MX31 and attached HITACHI hard disk C4K40 or SST NANDrive.	Other drives supporting up to UDMA mode 3 may be used.

### 3.5.2 Unit Test Software

The following table lists the required software to run the Storage Device Tests.

**Table 3-5. ATA Storage Device Test Software Requirements**

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.
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 Test cases.
rwtest.dll	Test .dll for various read/write options, including multi-threading and various block sizes.
Disktest_perf.dll	Test .dll file used to perform Storage Device Block Driver Performance Test cases.
perflog.dll	Logging library that provides functionality for timing and logging the performance data generated by the test dll.

### 3.5.3 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\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4I
```

### 3.5.4 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 (the help message within the test .dll is not correct) and is used to confirm over-writing of all information on the hard disk.

**NOTE**

Test cases 5019 and 5022 can be safely skipped.

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

```
tux -o -d disktest -c "-p HDProfile -zorch /maxsectors 65536"
```

**NOTE**

The free program memory to be adjusted to be larger than 64Mbytes in control panel, CETK cases 4021 can be safely skipped.

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"
```

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

```
tux -o -d disktest_perf -c "-profile 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 the drive to provide maximum sustained data rate from the media, to ensure the 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, refer to the following:

**Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Storage Device Tests > File System Driver Test**

**Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Storage Device Tests → Storage Device Block Driver API Test**

**Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Storage Device Tests → Storage Device Block Driver Read/Write Test**

**Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Storage Device Tests → Storage Device Block Benchmark Test**

**Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Storage Device Tests → Storage Device Block Driver Performance Test**

## 3.6 Basic Elements for Driver Development

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

### 3.6.1 BSP Environment Variables

Table 3-6. BSP Environment Variables

Names	Definition
BSP_ATA	Set to enable ATA device driver

### 3.6.2 Mutual Exclusive Drivers

Since the ATA as implemented for the 3-Stack board has no pin conflicts with the CSPI device (CSPI1), USB Host 1 port and PWM, there are no mutual exclusive drivers.

### 3.6.3 Dependencies of Drivers

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

Table 3-7. Environment Variables Used in the BSP

Names	Definition
SYSGEN_FATFS	Set to support FAT32 file system
SYSGEN_EXFAT	Set to support EXFAT 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.

## 3.7 Block Device API Reference

The primary interface to the ATA block device is through the standard Windows Embedded 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 6.0 Help for further details.

The driver also supports the standard XXX\_Init, XXX\_Deinit, XXX\_Open and XXX\_Close routines, as used by the 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.

### 3.7.1 IOCTL\_DISK\_DEVICE\_INFO

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

#### Parameters

<i>lpInBuffer</i>	[in] Pointer to a STORAGEDEVICEINFO structure.
<i>nInBufferSize</i>	[in] Specifies the size of the STORAGEDEVICEINFO structure.



*lpBytesReturned* [out] Pointer to a DWORD to receive the total number of bytes returned.

### 3.7.2 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.

### 3.7.3 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.

### 3.7.4 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.

### 3.7.5 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.

### 3.7.6 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.

### 3.7.7 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.

### 3.7.8 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 4

# 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. Note that for the 3-Stack board set, the ssi channel for Voice CODEC is not connected on the CPU engine board. So only playback function is supported. Recording function is not supported, and is disabled in the driver.

An application can access the audio driver using the methods and functions related with waveout function that are described in the following Platform Builder online help section:

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

### 4.1 Audio Driver Summary

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

**Table 4-1. Audio Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 CSP Driver Path	..\PLATFORM\common\src\soc\freescale\mxarm11_fsl_v1\audiodev
IC-specific CSP Driver Path	N/A
CSP Static Library	audiodev_mxarm11_fsl_v1.lib audiodev_record_stubs_mxarm11_fsl_v1.lib audiodev_record_mxarm11_fsl_v1.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\WAVEDEV\MC13783
Import Library	N/A
Driver DLL	wavedev_MC13783.dll
Required Catalog Items	Third Party > BSP > Freescale i.MX31 3DS:ARMV4I > Device Drivers > Audio > MC13783 Audio Driver

Driver Attribute	Definition
Recommended Catalog Items	Core OS > CEBASE > Graphics and Multimedia Technologies > Audio > Waveform Audio Core OS > CEBASE > Graphics and Multimedia Technologies > Media > Audio Codecs and Renderers > MP3 Codec Core OS > CEBASE > Graphics and Multimedia Technologies > Media > Audio Codecs and Renderers > MPEG-1 Layer 1 and 2 Audio Codec Core OS > CEBASE > Graphics and Multimedia Technologies > Media > Audio Codecs and Renderers > MS ADPCM Audio Codec Core OS > CEBASE > Graphics and Multimedia Technologies > Media > Audio Codecs and Renderers > Wave/AIFF/au/snd File Parser Core OS > CEBASE > Graphics and Multimedia Technologies > Media > Audio Codecs and Renderers > Waveform Audio Renderer Core OS > CEBASE > Graphics and Multimedia Technologies > Media > Audio Codecs and Renderers > WMA Codec Core OS > CEBASE > Graphics and Multimedia Technologies > Media > Windows Media Player > Windows Media Player Core OS > CEBASE > Graphics and Multimedia Technologies > Media > Windows Media Player > Windows Media Player OCX Core OS > CEBASE > 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

The Recommended Catalog Items listed in the table above 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. Refer to the following Platform Builder online help section if additional information about these items is required:

### Windows Embedded CE Features → Audio

## 4.2 Requirements

The audio driver must meet/support the following requirements:

1. Conform to the Microsoft audio driver architecture as defined for Windows Embedded CE 6.0 and all related operating systems.
2. Support any Freescale MXARM11-based platform that is compatible with the MC13783 PMIC.
3. Use double-buffered DMA operations to transfer audio data between memory and the SSI FIFO.
4. Support two power management modes, full on and full off.
5. Minimize power consumption at all times using clock gating and by disabling all audio-related hardware components that are not actively being used.

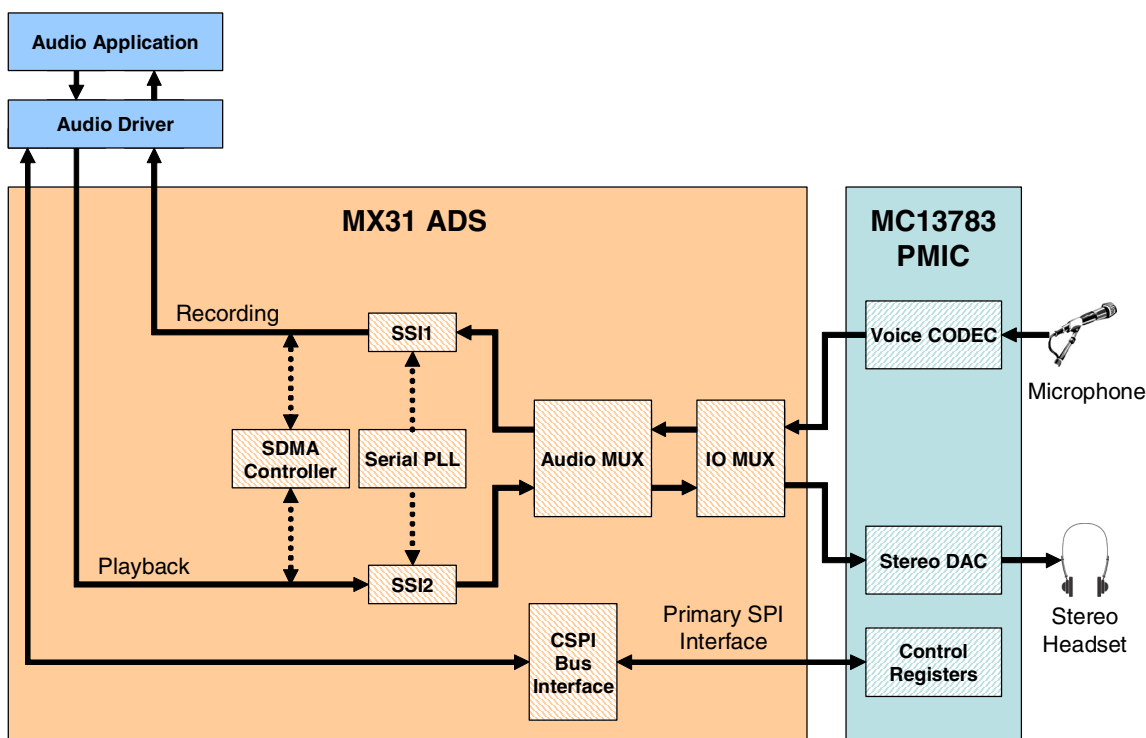


Figure 4-1. Audio Playback and Recording Hardware Components

### 4.3 Hardware Operation

Figure 4-1 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.

Note that on the 3-Stack board, the connection between the IOMUX module of MX31 and Voice CODEC of PMIC is not available.

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.

### 4.3.1 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 slave mode using the CLIA (from MC13783) clock signal to generate the appropriate framesync and bitclock signals. Note that the clock gating scheme is used with SSI2 where 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.
- The SSI2 transmitter watermark levels 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. 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 re-enabled 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 4 (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 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 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 a double-buffering scheme is used, 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 the entire audio stream is transmitted, there will be no more data available to refill the empty DMA buffers. Therefore, the output DMA channel can be disabled 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 disabling SSI2 to avoid any extraneous noise or “pop” that may be heard over the headphones.
- Finally, we also disable and clock gate SSI2.

### 4.3.2 Speaker output

The hardware supports speaker output as follows:

- The system detects in real-time if the headset is plugged in or unplugged.
- If the headset is plugged in, the speaker output is disabled.
- If the headset is unplugged, the speaker output is enabled.

### 4.3.3 Required SoC Peripherals

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

- SSI2 synchronous serial interfaces, used for playback.
- The Serial Clock PLL to provide the master clock signal for SSI2 (for SSI master mode).
- A 26 MHz clock signal generator that supplies the CLIA clock input to the MC13783 PMIC (for MC13783 master mode).
- The Digital Audio MUX to connect 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 to the MC13783 PMIC.
- The SDMA Controller to manage the DMA channels that are used for playback.

### 4.3.4 Conflicts with Other SoC Peripherals

#### 4.3.4.1 i.MX31 Peripheral Conflicts

There are no known conflicts between the SoC peripherals that are required by the audio driver and any other device driver.

### 4.3.5 Known Issues

None.

### 4.3.6 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.
- Digital audio buses in order to transfer data between the SSI and the Stereo DAC
- The CLIA clock input is also required if the Stereo DAC is 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.

## 4.4 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 Embedded CE Drivers → Audio Drivers → Audio Driver Development Concepts.**

### 4.4.1 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 through 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.

### 4.4.2 Audio Recording

Note that the recording function is not supported for hardware limitation and disabled in driver.

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 through 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.

### 4.4.3 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 4-2](#) and [Table 4-3](#).

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.



Table 4-2. Audio Driver Configuration Settings (hwctxt.h)

Configuration Setting Name	Description
OUTCHANNELS	Defines the number of output/playback channels that are available. Can be set to either 1 or 2. Default is 2.
BITSPERSAMPLE	The number of data bits per audio sample. This must match with the HWSAMPLE typedef and the AUDIO_SAMPLE_MAX/AUDIO_SAMPLE_MIN values. Default is 16.
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 2048 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. Audio hardware disabling can be delayed 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 4-3. Audio Driver Configuration Settings (hwctxt.cpp)

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 FALSE for both settings.
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.

Configuration Setting Name	Description
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 PORT5.
STEREO_DAC_AUDMUX_PORT	The external Digital Audio MUX port that is connected to the Stereo DAC. The default is PORT4.
VOICE_CODEC_AUDIO_BUS	The digital audio bus that connects the Audio MUX to the Voice CODEC. The default is AUDIO_DATA_BUS_2.
STEREO_DAC_AUDIO_BUS	The digital audio bus that connects the Audio MUX to the Stereo DAC. The default is AUDIO_DATA_BUS_1.
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.

#### 4.4.4 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 playback) and `BSP_SDMA_SUPPORT_SSI2` (for audio record) 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 4-4](#) is a

summary of the issues and tradeoffs regarding which type of memory should be used for the DMA data buffers.

**Table 4-4. DMA Memory Allocation Issues and Considerations**

Memory Region	Memory Usage Issues and Considerations
Internal	Allows the external memory to be placed in a low power mode while the DMA data buffers are being processed to reduce system power consumption (as long as nothing else on the system requires access to external memory). 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 Windows Embedded CE 6.0 system calls.
	The external memory cannot be placed into a low power mode while the DMA is active.

**Table 4-5. 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 4-5](#) 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()`. Refer to the CSPDDK documentation for additional information about the `DDKSdmaAllocChain()` API.

## 4.4.5 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 6.0 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.

The audio driver can work correctly after resume for power down mode.

### 4.4.5.1 PowerUp

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

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

### 4.4.5.2 PowerDown

This function has been implemented to support suspending all currently active audio I/O 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 and must not block or depend on any hardware interrupts. Therefore, the first thing that this function must do is to signal all of the possible wait events that the normal audio driver thread may be currently waiting on. If it is not done, the `PowerDown` thread may be blocked waiting for a critical section that is currently being held by the normal audio driver thread. This is an error and would deadlock the entire system and prevent it from properly entering the low power state.

Since, all possible waiting events are signaled, the normal audio thread will now be guaranteed (because of priority inversion) to run to the point where it will release the required critical section and allow the `PowerDown` thread to proceed without the possibility of deadlocking.

Now it is ensured that the normal audio thread is not executing inside any critical section, the `PowerDown` thread can safely proceed to disable all active audio DMA operations and to powerdown all of the associated audio hardware components. Once this has been done, the audio driver will remain in its low power state until the `PowerUp` function is called by the Power Manager. This functionality is currently handled by `IOCTL_POWER_SET` and the function is just a stub.

### 4.4.5.3 IOCTL\_POWER\_SET

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

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

This registry entry is required for proper power management functionality.

## 4.4.6 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.

### 4.4.6.1 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
```

### 4.4.6.2 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.

Note that the registry settings for the recording does not function for the 3-Stack board.

```
[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
```

## Audio Driver

```
"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 using the Remote Registry Editor tool that is provided with Platform Builder.


## 4.5 Unit Test

The audio driver is tested using the Waveform Audio Driver Test suite that is included as part of the Windows CE 6.0 Test Kit (CETK). The test suite includes both automated and interactive tests that are used to test various playback and recording functions.

### 4.5.1 Unit Test Hardware

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

**Table 4-6. Unit Test Hardware Requirements**

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 platforms. 
Mono microphone.	This is not required for the MX31 3-Stack board.

### 4.5.2 Unit Test Software

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

Table 4-7. Unit Test Software Requirements

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
wavetest.dll	Test .dll file

### 4.5.3 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\Microsoft Platform Builder\6.00\cepb\wctk\ddtk\armv4i**

### 4.5.4 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. As full-duplex operation is not supported, the command line should be *tux -o -d wavetest -c "-e -t 5 -p"*.

The tread count used in test case 6000 Playback Mixing needs to be reduced to “5” instead of the default value of “9” because of a CETK known issue. Please refer to ticket ENGR76843.

Refer to BSP release notes for the test cases failed as MSFT's known issue.

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

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

## 4.6 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.

### 4.6.1 Checking for a Boot-time Musical Tune

The normal Windows Embedded CE 6.0 boot procedure includes playing a short musical tune just before displaying the 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.

## 4.6.2 Confirming Touchpanel Taps and Keypad Key Presses

The normal Windows Embedded CE 6.0 system configuration includes the ability to play back 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.

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

## 4.6.3 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.

## 4.6.4 Using the SDK Sample Audio Applications for Testing

The Windows Embedded CE 6.0 SDK that is included as part of Platform Builder includes two audio-related sample applications. The wavrec sample application, which is not supported on PDK, 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 Embedded CE Features → Audio → Waveform Audio → Waveform Audio Samples**

## 4.7 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 Embedded CE Drivers → Audio Drivers → Audio Driver Reference → Waveform Audio Driver Reference**

## 4.8 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.

### 4.8.1 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.

## 4.8.2 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 OSDesignsimage, then you should see all of the following messages in the Platform Builder output window:

```
Loaded symbols for
'D:\WINCE600\OSDesigns\<workspace>\RELDIR\<platform>_ARMV4I_RELEASE\WAVEDEV_MC13783.DLL'
Loaded symbols for
'D:\WINCE600\OSDesigns\<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:\WINCE600\OSDesigns\<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:\WINCE600\OSDesigns\<workspace>\RELDIR\<platform>_ARMV4I_DEBUG\WAVEAPI.DLL'
```

## 4.8.3 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.

## 4.8.4 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.



## Chapter 5

# Backlight Driver

The backlight driver uses the hardware provided by the display module on the chip to control the backlight on the LCD display.

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 use its own defined timer to set the backlight power states.

### 5.1 Backlight Driver Summary

**Table 5-1. Backlight Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\FREESCALE\MXARM11_FSL_V1\BACKLIGHT\DRIVER
CSP Static Library	backlight_mxarm11_fsl_v1.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\BACKLIGHT\DRIVER
Import Library	N/A
Driver DLL	backlight.dll
Catalog Item	Third Party >BSPs > Freescale i.MX31 3DS: ARMV4I —>Device Drivers —>Smart Backlight Control—>Backlight IPU
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_BACKLIGHT_IPU=1 BSP_BACKLIGHT_MC13783 should not be set.

### 5.2 Requirements

The backlight driver should meet the following requirements:

1. Conform to the Device Manager streams interface.
2. Support 0~255 level adjustment.
3. Support power management mode full on / full off.

## 5.3 Hardware Operation

The hardware consists of an independently programmable register in the Image Processing Unit (IPU) module, SDC\_CUR\_BLINK\_PWM\_CTRL Register, which is used to control the signal output at the contrast pin. Default behavior for the Backlight Drivers is to control the pulse-width of the built-in pulse-width modulator, which controls the contrast of the LCD screen.

## 5.4 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

### 5.4.1 Backlight Driver Registry Settings

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

```
[HKEY_CURRENT_USER\ControlPanel\Backlight]

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

## 5.5 Unit Test

The backlight driver is tested by Application test.

### 5.5.1 Unit Test Hardware

The following table lists the required hardware to run the backlight application test.

**Table 5-2. Unit Test Hardware Requirements**

Requirements	Description
Epson L4F00242T03 VGA Panel	Display panel required for display of graphics data.

### 5.5.2 Unit Test Software

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

**Table 5-3. Unit Test Software Requirements**

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.

### 5.5.3 Running the Backlight Application Test

The following table lists the backlight application test procedures:

**Table 5-4. Backlight Application Test Procedures**

Test Cases	Entry Criteria/Procedure/Expected Results
Backlight Level	<p>Entry Criteria:</p> <p>N/A</p> <p>Procedure:</p> <ol style="list-style-type: none"> <li>1. Go to “Setting &gt; Control Panel”</li> <li>2. Double click on the “Display” icon, then click on the “Backlight” tab</li> <li>3. Click on the “Advanced...” button</li> <li>4. Modify the backlight level setting for both battery and external power</li> <li>5. Observe that the backlight level behaves according to the new setting</li> </ol> <p>Expected Result:</p> <p>N/A</p>

Backlight Timeout	<p>Entry Criteria:</p> <p>N/A</p> <p>Procedure:</p> <ol style="list-style-type: none"> <li>1. Go to “Setting &gt;Control Panel”</li> <li>2. Double click on the “Display” icon, then click on the “Backlight” tab</li> <li>3. Modify the backlight timeout setting for both battery and external power, and then click on “OK” button to apply the changes</li> <li>4. Observe the time it takes for the backlight to go out, make sure it correspond with the new settings entered in step 3</li> </ol> <p>Expected Result:</p> <p>N/A</p>
-------------------	---

## 5.6 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 Embedded Drivers → Streams Interface Drivers**”

## Chapter 6

# Battery Driver

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

The battery driver samples the voltage level upon initialization as well as on power up when coming out of suspend. It checks to determine if the charger and/or battery is attached, then decides whether to execute the charging or discharging operations. It also reports the battery capability and power supply state to the OS periodically by measuring battery voltage. During charging, current-limit and voltage-limit will take place to protect the charger and battery, and the operating system will be forbidden to enter suspend mode to prevent the charging operation from losing control.

### 6.1 Battery Driver Summary

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

**Table 6-1. Battery Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\BATTDVR\MC13783
Import Library	N/A
Driver DLL	battdrvr_MC13783.dll
Catalog Item	Third Party → BSP → Freescale i.MX31 3DS:ARMV4I → Device Drivers → Battery → MC13783 Battery
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_BATTERY=1, BSP_PMIC_MC13783=1

### 6.2 Requirements

The battery driver should meet the following requirements:

1. Conform to the Device Manager streams interface.
2. Support the <TGTPLAT> MC13783 PMIC.
3. Support the main battery without the support of the change notification.

### 6.3 Hardware Operation

The battery driver is implemented with the aid of the MC13783 Power Management Integrated Circuit (PMIC). The PMIC is a multi-functional 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.

### 6.3.1 Conflicts with other SoC Peripherals

No conflicts.

## 6.4 Software Operation

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

### 6.4.1 Battery Driver Registry Settings

The following registry keys are required to properly load the 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"
    "Flags"=dword:8 ; DEVFLAGS_NAKEDENTRIES
    "IClass"="{DD176277-CD34-4980-91EE-67DBEF3D8913}"
    "BattFullLiftTime" = dword:8 ;Batt Spec defined: in hr,8hr is assumed
    "BattFullCapacity"=dword:960 ;Batt Spec defined: in mAh,2400mAh is assumed
    "BattMaxVoltage"=dword:1068 ;Batt Spec defined: in mV,4200mV is assumed
    "BattMinVoltage"=dword:BB8 ;Batt Spec defined: in mV,3000mV is assumed
    "BattPeukertNumber"=dword:73 ;Batt Spec defined, 1.15 is assumed
    "BattChargeEff"=dword:50 ;Batt Spec defined, 0.80 is assumed
    "PollInterval"=dword:000003e8 ;Batt poll interval in milliseconds
    "Dll"="battdrvr_MC13783.dll"
```

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

### 6.4.2 Power Management

There is no additional power management implementation done specifically for the battery driver other than the implementation described in section 17.5.6 of Power Management IC (PMIC) reference document.

## 6.5 Unit Test

The battery can be tested, by switching on the system and watching the power properties window. When charging, the LED indicator will turn on and the charge capacity of the battery can be seen increasing until



charged to 100%. When without battery attached or battery fully charged or supplied with battery, the LED indicator will be turned off.

### CAUTION

Please do not plug in or remove the battery after booting up the device. The device can be damaged if the battery is plugged in or removed after boot.

## 6.5.1 Unit Test Hardware

The MX31 3-Stack board is required.

## 6.6 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 Embedded CE Drivers** → **Battery Drivers**

### 6.6.1 Battery PDD Functions

#### 6.6.1.1 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

#### 6.6.1.2 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;
```

## 6.6.2 Battery Driver Structures

### 6.6.2.1 Battery Channels Structure

```
typedef enum _BATTDVR_CHANNELS {  
    BattVoltage,  
    BattCurrent,  
    ChargerVoltage,  
    ChargerCurrent,  
    TotalChannels,  
} BATTDVR_CHANNELS;
```

### 6.6.2.2 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;
```

## Chapter 7

# Bluetooth Driver

The Bluetooth driver is used to drive the APM6628 module to implement Bluetooth functionality compatible with Bluetooth v2.0 +EDR. Bluetooth exchanges data with the through the UART2 port. The APM6628 module adopts BlueCore4 Bluetooth solution of Cambridge Silicon Radio company.

### 7.1 Bluetooth Driver Summary

The Bluetooth driver is provided in binary form instead of source codes. [Table 7-1](#) provides a summary of the source code location, library dependencies, and other BSP information.

**Table 7-1. Bluetooth Driver Summary**

Driver Attribute	Definition
Target Platform	
Target SOC	
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\BLUETOOTH
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<Target Platform>\SRC\DRIVERS\BLUETOOTH
Driver DLL	bthbcsp.dll btp_bchs.dll btp_modules.dll bth_avdrv.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <Target Platform>: ARMV4I > Device Drivers > BlueTooth > CSR BlueTooth Third Party > BSP > Freescale <Target Platform>: ARMV4I > Device Drivers > Serial > UART2 serial port support Core OS > CEBASE > Communication Services and Networking > Networking - Personal Area Network(PAN) > Bluetooth > Bluetooth Protocol Stack with Transport Driver Support > Bluetooth Stack with Universal Loadable Driver (exclude other HCI drivers) Core OS > CEBASE > Applications and Services Development > .NET Compact Framework 2.0 > .NET Compact Framework 2.0 Core OS > CEBASE > Applications and Services Development > Object Exchange Protocol(OBEX) > OBEX Client Core OS > CEBASE > Applications and Services Development > Object Exchange Protocol(OBEX) > OBEX Server > OBEX File Browser Core OS > CEBASE > Applications and Services Development > Object Exchange Protocol(OBEX) > OBEX Server -> OBEX Inbox

SYSGEN Dependency	SYSGEN_BTH=1 SYSGEN_DOTNETV2=1 SYSGEN_OBEX_FILEBROWSER=1 SYSGEN_OBEX_CLIENT=1 SYSGEN_OBEX_INBOX=1
BSP Environment Variables	BSP_CSR_BLUETOOTH =1 BSP_SERIAL_UART2 =1

The Catalog Items in [Table 7-1](#) should be included in the OS design in order to provide Bluetooth Profiles. When **Third Party > BSP > Freescale <Target Platform>: ARMV4I > Device Drivers > Bluetooth > CSR Bluetooth** is selected, other catalog items are selected automatically.

## 7.2 Supported Functionality

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

- Drives Bluetooth module in APM6628 chip
- Provides communication between Bluetooth module and UART2 driver with serial baudrate up to 921.6kbps
- Supports A2DP SOURCE (Advanced Audio Distribution Profile)
- Supports AVRCP (Audio Video Remote Control Profile)
- Supports FTP (File Transfer Profile)

## 7.3 Hardware Operation

The Bluetooth driver exchanges data and commands between the BCHS (BlueCore Host Software) stack and Bluetooth hardware via UART2 port.

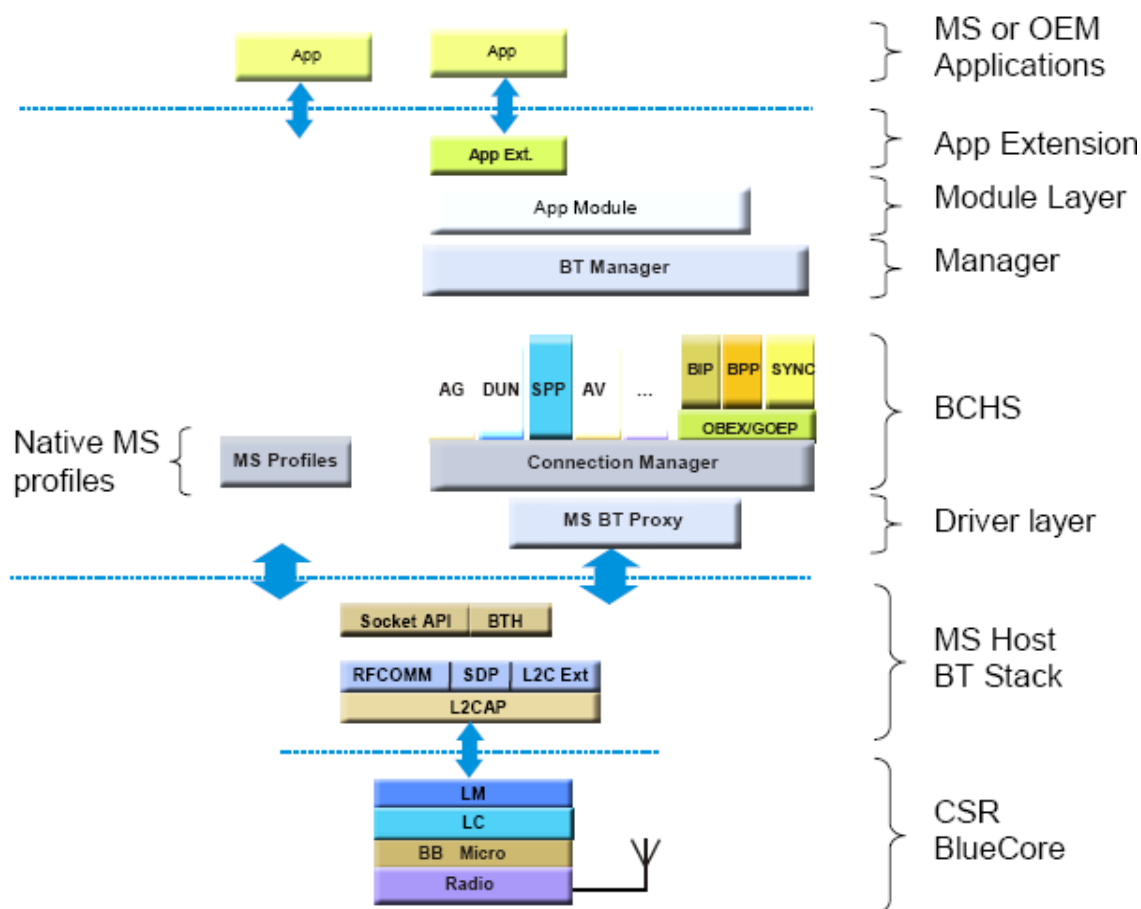
## 7.3.1 Conflicts with Other Peripherals and Catalog Items

### 7.3.1.1 Conflicts with SoC Peripherals

### 7.3.1.2 Conflicts with 3-Stack Peripherals

## 7.4 Software Operation

The overall software architecture with existing Microsoft Bluetooth stack and CSR BCHS stack is shown in [Figure 7-1](#)



**Figure 7-1. Software Architecture of Bluetooth Driver and Protocol**

The BCHS is an embedded Bluetooth software package complementing the already existing Microsoft Bluetooth profiles delivered as part of the Microsoft Windows CE OS. BCHS is developed to operate on top of the native Microsoft Bluetooth stack and not as a replacement of the Microsoft Bluetooth stack.

## 7.4.1 Registry Settings

## 7.5 Unit Test

Bluetooth test includes CETK test and manual tests for A2DP, AVRCP and FTP.

### 7.5.1 Unit Test Hardware

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

**Table 7-2. Hardware Requirements**

Requirement	Description
Bluetooth Headset	Bluetooth Headset which supports SBC decoder for testing A2DP and AVRCP feature. HT820 headset is used
Mobile phone or PC	with Bluetooth feature. Nokia mobile phone and laptop DELL D610 is used
Two 3-stack boards	CETK for Bluetooth needs two Bluetooth boards on TCP/IP networking

### 7.5.2 Unit Test Software

Table 7-3 lists the required software to run the unit tests.

**Table 7-3. Software Requirements**

Requirement	Description
Tux.exe	Tux test harness, which is required for executing the test.
Kato.dll	Kato logging engine, which is required for logging test data.
Tooltalk.dll	Application required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation.
Netall.dll	Provides functions that generate random numbers, output data, and parse command lines
Btwsrvr22.exe, Btw22.exe	CETK MS Bluetooth Test
bthapitst.dll	CETK Bluetooth API Test
Perflog.dll, Perf_bluetooth.dll	CETK Bluetooth Performance Test
hciqa_con.dll, ddx.dll	CETK Bluetooth HCI Transport Driver Test

## 7.5.3 Running the Unit Tests

### 7.5.3.1 Running Bluetooth CETK

#### 7.5.3.1.1 Running the CETK Bluetooth API Test

The API test requires two Bluetooth boards: one for the client and one for the server. The test steps are as follows:

1. Bootup the two 3-Stack boards

2. Copy Kato.dll, Tooltalk.dll, Netall.dll and bthapitst.dll into the Windows directory in client board
3. In the client board, open **Run** from START and enter **tux -o -d bthapitst.dll -c "-s server\_bt\_addr"** command to execute this test. Where *server\_bt\_addr* is the Bluetooth address of the Windows Embedded CE based device running as a server. For example, if the server address is 0123456789ab, the command line should read: **tux -o -d bthapitst.dll -c "-s 0123456789ab"**.

### 7.5.3.1.2 Running the CETK Bluetooth Performance Test

The performance test requires two Bluetooth boards: one for the client and one for the server. The test steps are as follows:

1. Bootup two 3-Stack boards.
2. Copy Kato.dll, Tooltalk.dll, Perflog.dll and Perf\_bluetooth.dll into the Windows directory in both boards
3. In the server board, open **Run** from START and enter **tux -o -d perf\_bluetooth -c "-i NumberOfIterations -b NumberOfBuffers -p ServerChannelNumber"** command to execute this test. Such as **tux -o -d perf\_bluetooth -c "-i 10 -p 6 -b 163840"**
4. In the client board, open **Run** from START and enter **tux -o -d perf\_bluetooth -c "-s server\_bt\_addr -i NumberOfIterations -b NumberOfBuffers -p ServerChannelNumber"** command to execute this test. Such as **tux -o -d perf\_bluetooth -c "-s 0123456789ab -i 10 -p 6 -b 163840"**

To view the test results:

1. Copy the .log file to the development workstation.
2. From <Platform Builder installation path>\Cepb\Wcetek\Ddtk\Desktop, copy Pparse.exe to the directory that contains the log file.
3. In the directory that contains the log file, run the following command: **pparse log\_filename parsed\_filename**, where *log\_filename* is the name of the log file and *parsed\_filename* is the name of the .csv file that you want to create to store the parsed test results.
4. In Excel, open the .csv file.

### 7.5.3.1.3 Running the CETK Bluetooth HCI Transport Driver Test

The HCI transport driver test requires two Bluetooth boards: one for the client and one for the server. The test steps are as follows:

1. Bootup two 3-Stack boards.
2. Copy Kato.dll, Tooltalk.dll, hciqa\_con.dll and ddlx.dll into the Windows directory in both boards.
3. In the server board, open **Run** from START and enter **tux -o -d ddlx.dll -c "-d hciqa\_con.dll -i 2 -c /accept /class 0x010000"** command to execute this test.
4. In the client board, open **Run** from START and enter **tux -o -d ddlx.dll -c "-d hciqa\_con.dll -i 2 -c /class 0x010000"** command to execute this test.

#### NOTE

Refer to <http://msdn.microsoft.com/en-us/library/bb203069.aspx> for detailed CETK information.

### 7.5.3.2 Manual Test Bluetooth

#### NOTE

Follow the steps shown below exactly, otherwise there may be unexpected results.

#### 7.5.3.2.1 Running the Bluetooth A2DP Test

The purpose of the A2DP test is to listen to stereo music played by MediaPlayer from the Bluetooth headset. The test steps are as follows:

1. Make Bluetooth headset entering *pairing* mode
2. Open the Bluetooth Device Properties tools in the control panel and click *scan device* icon
3. The 3-stack board sets up the audio connection with Bluetooth headset. Music played by MediaPlayer, may be listened from your Bluetooth headset.

#### 7.5.3.2.2 Running the Bluetooth AVRCP Test

1. After A2DP has been setup, play a music file with the MediaPlayer. Long-press the volume-up or volume-down button on the headset and the music volume from headset changes accordingly.
2. Click **PLAY/PAUSE/STOP** button, the MediaPlayer pauses the music. Then re-click this button, and the MediaPlayer plays the music again. Long-press this button, and the MediaPlayer stops.

#### 7.5.3.2.3 Running the Bluetooth FTP Test

1. Open the Bluetooth Device Properties tools in the control panel and click the scan device icon
2. If you open the Bluetooth Neighborhood folder under My Device, the paired Bluetooth device (phone or PC) is asked to permit the 3-stack board access. After you click OK and input the default password (0000), also input 0000 in the window in the 3-Stack board. A shared folder of the paired Bluetooth device appears and you may get/put files from/to this shared folder



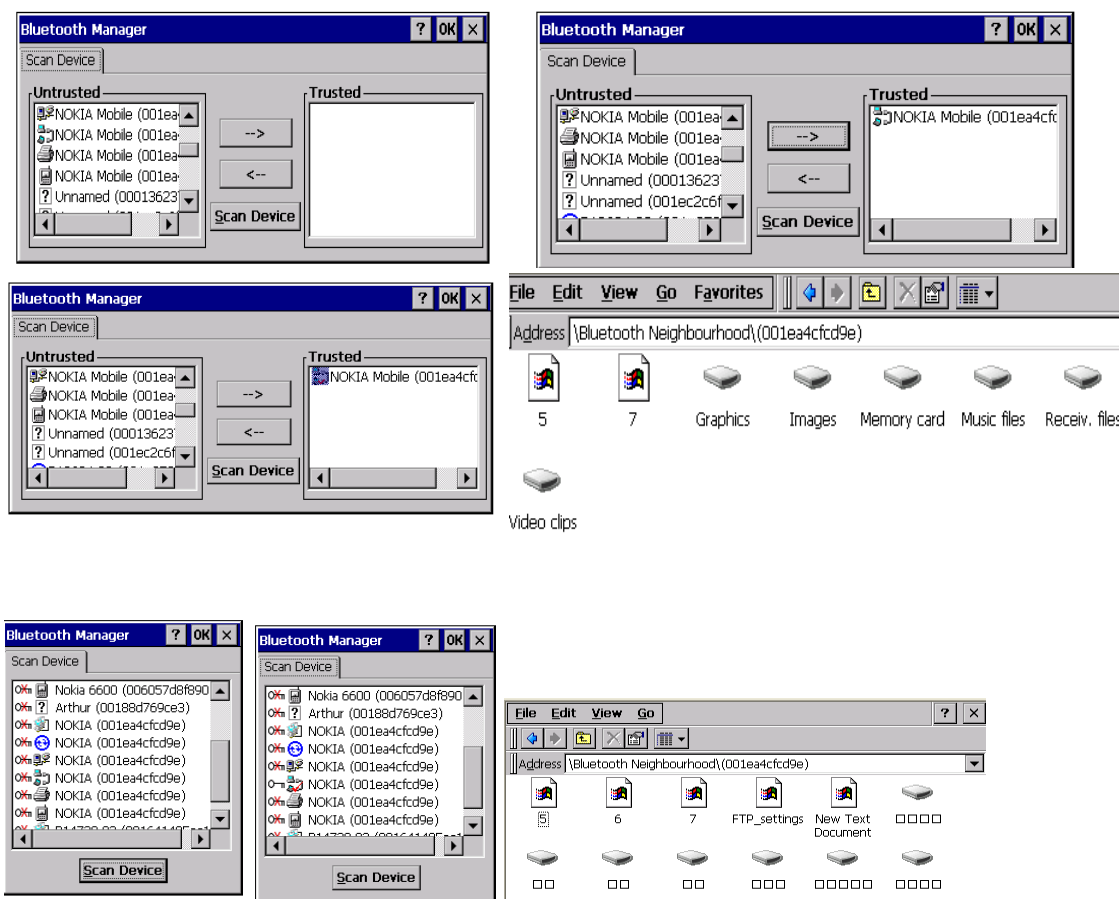


Figure 7-2. Bluetooth FTP Test

**NOTE**

Make sure you use the correct Bluetooth A2DP and FTP icons, and do not use other icons.

## 7.5.4 Operation Attention Items and Tips

You must strictly follow the Bluetooth manual test steps given above. This section reaffirms the items to pay close attention to.

- Ensure that the Bluetooth headset is in pairing mode, then begin to scan the device from the Bluetooth Manager Window
- After the Bluetooth headset is scanned, a password window appears. Quickly input the default password '0000'. If the headset icon in Bluetooth Manager window is a question mark and headset is not in pair mode, the password was inputted to slow. Set the headset in pair mode and re-scan. If the headset icon in the Bluetooth Manager window is a question mark, and headset is in pair mode, the password is incorrect. In this case, trust the headset icon in the manager window, then

un-trust it to delete the headset password information. Then set the headset in pair mode and re-scan it and input the correct password.

- Pay attention to the A2DP and FTP icon which may refer to `WINCE600\PUBLIC\COMMON\OAK\DRIVERS\NETUI` and use the right icon.
- It is better to use non-activating than deleting, because after deleting, you must re-pair the Bluetooth device if you want to reuse it.

### 7.5.5 Known Issues

- If you move the Bluetooth headset from the right block to the left block before the headset is activated, do not again move the headset to the right block. This confuses the Bluetooth. The correct operation is the following steps after headset is removed into left block
  - Ensure headset in pair mode
  - Rescan and input password
  - Move headset into right block and active it.

The reason is that the Bluetooth Property Application provided by Microsoft deletes the trusted Bluetooth headset security register.

- When scan is running, do not reopen the Bluetooth Property Application in the control panel, otherwise the Bluetooth Property Application will be in an unexpected state, such as cannot stop or cannot reopen if you close this window. This reason is that Bluetooth Property Application provided by Microsoft is not handled well for unique instance.
- Bluetooth API CETK fails in hold mode test, because CSR Bluetooth protocol does not support this mode. CSR fixes this in a later version.

## Chapter 8

# Camera Driver

The camera driver is based on the Windows CE 6.0 Camera Device Driver Interface. 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 OV2640 camera sensor modules.

At the lower layer, the camera driver performs several tasks including:

- Communicating with and configuring a camera sensor through the I<sup>2</sup>C interface
- Interfacing with the Image Processing Unit (IPU) to perform pre-processing tasks on captured images
- Configuring the IPU Synchronous Display Controller (SDC) for direct display of video preview data

### 8.1 Camera Driver Summary

Table 8-1 provides a summary of source code location, library dependencies and other BSP information.

**Table 8-1. Camera Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 SOC Driver Path	..\PLATFORM\COMMON\SRC\SOC\freescall\mxarm11_fsl_v1\ipu\camera
SOC Driver Path	N/A
SOC Static Library	camera_mxarm11_fsl_v1.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\IPU\CAMERA
Import Library	N/A
Driver DLL	camera.dll
Catalog Items	Third Party > BSPs > Freescale i.MX31 3DS: ARMV4I > Device Drivers > Camera > Camera
SYSGEN Dependency	SYSGEN_DSHOW_CAPTURE=1
BSP Environment Variables	BSP_CAMERA=1

### 8.2 Supported Functionality

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

- Supports the Windows CE Camera Device Driver Interface
- Supports Preview, Capture, and Still pins

- Supports a Direct-to-Display preview mode
- Supports the OV2640 camera sensors
- Supports power management operations

## 8.3 Hardware Operation

Several hardware modules are involved in the operation of the camera driver. The OV2640 camera sensor captures external image data. All other hardware elements of the camera driver are within the 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 is subsequently pre-processed by the IPU Image Converter (IC) module. 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.

## 8.4 Software Operation

### 8.4.1 Communicating with the Camera

Communication with the camera driver is accomplished through camera APIs defined by Microsoft for Windows CE 6.0. Applications may access these APIs directly or through the DirectShow video capture support.

#### 8.4.1.1 Using the Windows CE Camera Device Driver Interface

The Windows CE Camera Device Driver Interface provides basic support for video and still image capture devices. Refer to the following Windows CE 6.0 Help Documentation section for information on using these Camera APIs:

**Developing a Device Driver > Windows Embedded CE Drivers > Camera Drivers > Camera Driver Reference.**

#### 8.4.1.2 Using DirectShow for Video Capture

DirectShow provides support for the creation of filter graphs for video capture. Information on using DirectShow for video capture can be found in the following Windows CE 6.0 Help Documentation section:

**Windows Embedded CE Features > Encoded Media > DirectShow > DirectShow Application Development > DirectShow Architecture > Audio and Video Capture Support > Video Capture.**

### 8.4.2 Camera Registry Settings

Two sets of registry settings are needed for proper camera driver operation: One for the camera sensor and another for the camera driver.

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
"CameraId"=dword:3
"IClass"=multi_sz:
"{CB998A05-122C-4166-846A-933E4D7E3C86}", "{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

The CameraId registry key selects among the available camera sensor modules. Table 8-2 shows the valid values and the corresponding camera sensors.

**Table 8-2. CameraId Registry Key Settings**

Value	Camera Sensor
0	iMagic IM8803
1	iMagic IM8201
2	Magna521DA
3	OV2640

```
[HKEY_LOCAL_MACHINE\Software\Microsoft\DirectX\DirectShow\Capture]
"Prefix"="PIN"
"Dll"="camera.dll"
"Order"=dword:20
"Index"=dword:1
"PinCount"=dword:3 ;Pin count. Max = 3; default = 2
"MemoryModel"=dword:1 ; Pin memory mode.
"IClass"=multi_sz: "{C9D092D6-827A-45E2-8144-DE1982BFC3A8}",
                  "{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

### 8.4.3 Power Management

The camera driver consumes power primarily through the operation of various IPU sub-modules, such as:

- CSI which synchronizes and receives image data from the camera sensor
- 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. Support for transition to the Suspend and Resume states is provided through the IOCTL\_POWER\_SET IOCTL.

#### 8.4.3.1 PowerUp

This function is not implemented for the camera driver.

#### 8.4.3.2 PowerDown

This function is not implemented for the camera driver.

### 8.4.3.3 IOCTL\_POWER\_SET

The camera driver implements the IOCTL\_POWER\_SET IOCTL API with support for the D0, D1, D2 (Full on) and the D3, D4 (Off) power states. These states are handled in the following manner:

- D0-D2 – Action is only taken when resuming from the D4 state. If the camera was running when the transition to the D4 state occurred, the camera returns to a running state, re-enabling the CSI and IC modules.
- D3-D4 – Action is only taken if the camera is running when the request to transition to the D4 state occurs.

## 8.5 Unit Test

The camera driver is subject to the following test suites provided with the Windows CE Test Kit (CETK):

- Camera Driver Data Structure Verification Test - queries the driver for the various properties and formats, and verifies that the data structures returned are valid
- Camera Driver I/O Test - verifies the functionality of the preview and capture streams on the camera driver
- Camera and DirectShow Integration Test - verifies the functionality of the camera driver when used under DirectShow
- Camera Performance Test - gathers performance data for a number of DirectShow capture scenarios

Additionally, for Windows CE 6.0, a Camera Application written by Microsoft may be used to preview and capture still images.

### 8.5.1 Unit Test Hardware

Table 8-3 lists the required hardware to run the Windows CE 6.0 Camera CETK test and the camera application.

**Table 8-3. Hardware Requirements**

Requirements	Description
Camera functionality	The device must have camera functionality, currently OV2640 sensor is used

### 8.5.2 Unit Test Software

#### 8.5.2.1 Custom Camera CETK Test

Table 8-4 lists the required software to run the Camera CETK Test.

**Table 8-4. Camera CEKT Test Software Requirements**

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
CameraGraphTests.dll	Library containing the camera and directshow integration test cases
CamTestProperties.dll	Library containing the camera driver data structure verification test cases
CamIOTests.dll	Library containing the camera driver I/O test cases
CameraPerfTests.dll	Library containing the camera performance test cases
CameraGrabber.dll	Filter required by many command line parameters to track and output information about media samples

SYSGEN\_DSHOW\_CAPTURE (DirectShow capture) is also required. In addition, the configuration file “capconfig.ini” is required for CameraPerfTests.dll.

### 8.5.2.2 Freescale Camera Application

Table 8-5 lists the required software to run the custom camera application.

**Table 8-5. Custom Camera Application Software Requirements**

Requirements	Description
Camapp.exe	Executable file for the camera application

### 8.5.2.3 Camera Application

No additional actions are required to include the Windows CE 6.0 camera application in an OS image beyond the required registry keys.

## 8.5.3 Building the Camera Tests

### 8.5.3.1 Camera CETK Test

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

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

### 8.5.3.2 Freescale Camera Application

The following steps can be used to build the custom camera application:

1. Build an OS image for the desired configuration
2. Add a new folder named “APP” under the folder “\WINCE600\PLATFORM\imx313ds\src\”
3. Copy the folder of “Camapp” under the folder “APP”
4. Setup a new blank dirs file under the folder “Camapp”
5. Enter the build command at the prompt and press build camapp
6. Find the “camapp.exe” file in “obj\release” or “obj\debug” folder under folder “camapp”

## 8.5.4 Running the Camera Tests

### 8.5.4.1 Running the Camera CETK Test

The following are the tests available and the test procedures for each of the tests. For detailed information on these tests see the relevant subsections under “CETK Tests” in the Windows CE 6.0 Help Documentation section: **Windows Embedded CE Test Kit > CETK Tests and Test Tools > CECETK Tests > Camera Tests.**

- The command line *tux -o -d CameraGraphTests.dll* runs the camera and directshow integration test
- The command line *tux -o -d CamTestProperties.dll* runs the Camera Driver Data Structure Verification Test
- The command line *tux -o -d CamIOTests.dll* runs Camera Driver I/O Test
- The command line *tux -o -d cameraperftests.dll -c "-p \release\capresults.csv -c \release\capconfig.ini"* runs the Camera Performance Test. Note that this test requires the “capconfig.ini” configuration file which specifies what is to be tested, copying the file under the corresponding folder such as “\release” before testing from the following location:  
[Drive]:\Program Files\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4I

#### NOTE

The tests that involve audio capture are skipped due to a hardware audio recording limitation. For more information, see [Chapter 4, “Audio Driver.”](#)

### 8.5.4.2 Running the Freescale Camera Application

In the target control command prompt, use the following command to execute the custom camera application:

```
s camapp.exe
```

## 8.6 Camera Driver API Reference

Documentation for the camera driver APIs can be found within the latest Windows CE 6.0 Documentation.

Reference information on basic camera driver functions, methods, and structures can be found at the following location in the Windows CE 6.0 Help Documentation:

**Developing a Device Driver > Windows Embedded CE Drivers > Camera Drivers > Camera Driver Reference**



## Chapter 9

# Chip Support Package Driver Development Kit (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)

### 9.1 CSPDDK Driver Summary

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

**Table 9-1. CSPDDK Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\FREESCALE\MXARM11_FSL_V1\CSPDDK
CSP Driver Path	\PLATFORM\COMMON\SRC\SOC\FREESCALE\MX31_FSL_v1\CSPDDK
CSP Static Library	ddk_mx31_fsl_v1.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

### 9.2 Supported Functionality

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

- Supports 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)

- Exposes exported functions that can be invoked without incurring a system call (i.e. not a stream driver)

## 9.3 Hardware Operation

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

### 9.3.1 Conflicts with Other Peripherals

No conflicts.

## 9.4 Software Operation

### 9.4.1 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.

### 9.4.2 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. [Table 9-2](#) describes the available CSPDDK compile options.

**Table 9-2. CSPDDK Compile Options**

Compilation Variable	Header Location	Description
IMAGE_WINCE_DDKSDMA_IRAM_PA_START	image_cfg.h	Physical starting address in <u>internal</u> RAM (IRAM) where the shared SDMA data structures will be located
IMAGE_WINCE_DDKSDMA_IRAM_OFFSET	image_cfg.h	Offset in bytes from the base of IRAM for the SDMA data structures
IMAGE_WINCE_DDKSDMA_IRAM_SIZE	image_cfg.h	Size in bytes of the IRAM reserved for SDMA data structures
IMAGE_WINCE_DDKSDMA_RAM_PA_START	image_cfg.h	Physical starting address in <u>external</u> RAM where the shared SDMA data structures will be located. This address must correspond to the region reserved in config.bib
IMAGE_WINCE_DDKSDMA_RAM_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.

BSP_SDMA_MC0PTR	bsp_cfg.h	Starting address for the shared SDMA data structures. Set to IMAGE_SHARE_IRAM_SDMA_PA_START to use internal RAM. Set to IMAGE_SHARE_SDMA_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 manages the allocation of buffer descriptor chains for drivers and applications. The allocation scheme first attempts 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 dynamically allocates the necessary storage from external memory.

To decrease power consumption in 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.

### 9.4.3 Registry Settings

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

### 9.4.4 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 are invoked by other drivers during power state transitions.

## 9.5 CSPDDK DLL Reference

### 9.5.1 CSPDDK DLL System Clocking (DDK\_CLK) Reference

The DDK\_CLK interface allows device drivers to configure and query system clock settings.

#### 9.5.1.1 DDK\_CLK Enumerations

Table 9-3. 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

## 9.5.1.2 DDK\_CLK Functions

### 9.5.1.2.1 DDKClockSetGatingMode

This function sets the clock gating mode of the peripheral.

```

BOOL DDKClockSetGatingMode(
    DDK_CLOCK_GATE_INDEX index,
    DDK_CLOCK_GATE_MODE mode)

```

#### Parameters

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

*mode* [in] Requested clock gating mode for the peripheral.

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

### 9.5.1.2.2 DDKClockGetGatingMode

This function retrieves the clock gating mode of the peripheral.

```

BOOL DDKClockGetGatingMode(
    DDK_CLOCK_GATE_INDEX index,
    DDK_CLOCK_GATE_MODE *pMode)

```

#### Parameters

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

*pMode* [out] Current clock gating mode for the peripheral.

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

### 9.5.1.2.3 DDKClockGetFreq

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

```

BOOL DDKClockGetFreq(
    DDK_CLOCK_SIGNAL sig,
    UINT32 *freq)

```

#### Parameters

*sig* [in] Clock signal.

*freq* [out] Current frequency in Hz.

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

#### 9.5.1.2.4 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.

#### 9.5.1.2.5 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.

#### 9.5.1.2.6 DDKClockSetpointRequest

This function requests the specified setpoint optionally blocks until the setpoint.

```

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 request has submitted.

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

### 9.5.1.2.7 DDKClockSetpointRelease

This function releases a setpoint previously requested using DDKClockSetpointRequest.

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

#### Parameters

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

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

### 9.5.1.3 DDK\_CLK Examples

#### Example 9-1. 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 9-2. Example: CSPDDK Clock Rate Query

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

UINT32 freq;

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

## 9.5.2 CSPDDK DLL GPIO (DDK\_GPIO) Reference

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

### 9.5.2.1 DDK\_GPIO Enumerations

Table 9-4. 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

## 9.5.2.2 DDK\_GPIO Functions

### 9.5.2.2.1 DDKGpioSetConfig

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

```
goodBOOL 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.

### 9.5.2.2.2 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 is 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.

### 9.5.2.2.3 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.

### 9.5.2.2.4 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.

### 9.5.2.2.5 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.

### 9.5.2.2.6 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.

#### 9.5.2.2.7 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 is shifted to the LSB.

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

#### 9.5.2.2.8 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.

#### 9.5.2.2.9 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 is shifted to the LSB.

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

#### 9.5.2.2.10 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.

### 9.5.2.3 DDK\_GPIO Examples

#### Example 9-3. 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);
```

### 9.5.3 CSPDDK DLL IOMUX (DDK\_IOMUX) Reference

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

### 9.5.3.1 DDK\_IOMUX Enumerations

Table 9-5. 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

### 9.5.3.2 DDK\_IOMUX Functions

#### 9.5.3.2.1 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 to be configured.

*outMux* [in] Output path configuration.

*inMux* [in] Input path configuration.

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

#### 9.5.3.2.2 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**

<i>pin</i>	[in] Functional pin name used to select the IOMUX output/input path to be configured.
<i>pOutMux</i>	[out] Output path configuration.
<i>pInMux</i>	[out] Input path configuration.
<b>Return Values:</b>	Returns TRUE if successful, otherwise returns FALSE.

**9.5.3.2.3 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**

<i>pad</i>	[in] Functional pad name used to select the pad to be configured.
<i>slew</i>	[in] Slew rate configuration.
<i>drive</i>	[in] Drive strength configuration.
<i>mode</i>	[in] CMOS/open-drain output mode configuration.
<i>trig</i>	[in] Trigger configuration.
<i>pull</i>	[in] Pull-up/pull-down/keeper configuration.
<b>Return Values:</b>	Returns TRUE if successful, otherwise returns FALSE.

**9.5.3.2.4 DDKIomuxGetPadConfig**

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

```

BOOL DDKIomuxGetPadConfig(
    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**

<i>pad</i>	[in] Functional pad name used to select the pad to be configured.
<i>pSlew</i>	[in] Slew rate configuration.
<i>pDrive</i>	[in] Drive strength configuration.
<i>pMode</i>	[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.

#### 9.5.3.2.5 DDKIomuxSetGpr

This function writes a value into the IOMUX GPR register. The GPR is used to control the muxing of signals within the SoC.

```
BOOL DDKIomuxSetGpr(
    UINT32 mask,
    UINT32 data)
```

##### Parameters

*mask* [in] Bit mask for GPR bits to be written.  
*data* [in] Data to be written.  
**Return Values:** Returns TRUE if successful, otherwise returns FALSE.

#### 9.5.3.2.6 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.

### 9.5.3.3 DDK\_IOMUX Examples

---

#### Example 9-4. 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 9-5. 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);
```

## 9.5.4 CSPDDK DLL SDMA (DDK\_SDMA) Reference

The DDK\_SDMA interface allows device drivers to allocate, configure, and control shared SDMA resources.

### 9.5.4.1 DDK\_SDMA Enumerations

Table 9-6. 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

### 9.5.4.2 DDK\_SDMA Functions

#### 9.5.4.2.1 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

<i>dmaReq</i>	[in] Specifies the DMA request to be bound to a virtual channel.
<i>priority</i>	[in] Priority assigned to the opened channel.
<i>lpName</i>	[in] Not currently used. Set to NULL.
<i>irq</i>	[in] Only used if lpName is set to NULL. Specifies the hardware IRQ to 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.

#### 9.5.4.2.2 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 to be bound to a virtual channel.

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

#### 9.5.4.2.3 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.

#### 9.5.4.2.4 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.

*numBufDesc* [in] Number of buffer descriptors to allocate for the chain.

**Return Values:** Returns TRUE if the chain allocation was successful, otherwise returns FALSE.

#### 9.5.4.2.5 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.

### 9.5.4.2.6 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

<i>chan</i>	[in] Virtual channel returned by DDKSdmaOpenChan.
<i>index</i>	[in] Index of buffer descriptor within the chain to be configured.
<i>modeFlags</i>	[in] Specifies the buffer descriptor mode word flags that control the “continue”, “wrap”, and “interrupt” settings.
<i>memAddr1PA</i>	[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.
<i>memAddr2PA</i>	[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.
<i>dataWidth</i>	[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.
<i>numBytes</i>	[in] Virtual channel returned by DDKSdmaOpenChan.
<b>Return Values:</b>	Returns TRUE if successful, otherwise returns FALSE.

### 9.5.4.2.7 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

<i>chan</i>	[in] Virtual channel returned by DDKSdmaOpenChan.
<i>index</i>	[in] Index of buffer descriptor within the chain.
<i>pStatus</i>	[in] Points to a buffer to be filled with the status of the buffer descriptor.



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

#### 9.5.4.2.8 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 to be filled with the status of each buffer descriptor in the chain.

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

#### 9.5.4.2.9 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.

#### 9.5.4.2.10 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.

#### 9.5.4.2.11 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.

**9.5.4.2.12 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.

**9.5.4.2.13 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 10

## Display Driver

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

The display driver supports the following display types:

- EPSON L4F00242T03 VGA LCD panel
- PAL and NTSC TV through the Chronitel CH7024 TV encoder chip

## 10.1 Display Driver Summary

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

**Table 10-1. Display Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
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\DLL
Import Library	ddgpe.lib, gpe.lib
Driver DLL	ddraw_ipu.dll
Catalog Items	Third Party > BSPs > Freescale i.MX31 3DS: ARMV4I > Device Drivers > Display > EPSON L4F00242T03 (VGA) Third Party > BSPs > Freescale i.MX31 3DS: ARMV4I > Device Drivers > TV Output > TV Output CH7023/CH7024
SYSGEN Dependency	SYSGEN_DDRAW=1
BSP Environment Variables	BSP_PP=1 BSP_DISPLAY_EPSON_L4F00242T03 = 1 for Epson LCD Panel BSP_TVOUT_CHRONTEL_CH702X = 1 for TV Output

## 10.2 Supported Functionality

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

- Supports EPSON 2.8" VGA Display With Touch Screen (L4F00242T03)
- Supports VGA portrait display resolution(480x640)
- Supports RGB565 interface

- Supports the DirectDraw Hardware Abstraction Layer (DDHAL)
- Supports overlay surface
- Supports video overlays containing image data in the FOURCC UYVY & YV12 pixel format
- Supports hardware-accelerated color space conversion in video overlays
- Supports hardware-accelerated image resizing in video overlays
- Supports overlay surface color key feature
- Supports overlay surface alpha blending feature
- Supports two power management modes, full on and full off
- Supports system suspend
- Supports screen rotation
- Supports dynamic switch between TV and LCD display

## 10.3 Hardware Operation

Refer to the chapter on the image processing unit (IPU) in the hardware specification document for detailed operation and programming information.

### 10.3.1 Rotation Control

Application rotate.exe provides a way to change the screen orientation while the Windows Embedded CE 6.0 image is running. Clicking rotate application toggles the orientation of the screen between a 0 and 270 degree rotation angle. The default path of rotate.exe is “\windows”.

#### NOTE

Due to lack of support for the co-existence of GDI screen rotation and DirectDraw (see the Windows CE Help documentation, 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/DIRECTDRAW CETK test suite. It is recommended to run these CETK tests with rotation support disabled or under 0 rotation degree.

### 10.3.2 TV Output Mode

Application tvout.exe provides a way to switch between LCD and TV Output mode (PAL standard). Clicking tvout application toggles between these two modes. The default path of tvout.exe is “\windows”. The tvout application sets the TV output mode to PAL or NTSC standards by receiving one parameter: “tvout.exe 0” sets TV output mode to PAL, “tvout.exe 1” sets to NTSC.

The display driver always ensures the output is LCD mode, when responding the power management to enter the power states D0 (Full On) and D4 (Off). TV output mode requires an 270 degree rotation so that the primary surface can fit to the physical resolution 640x480 that TV can support.

**NOTE**

Due to lack of support for the co-existence of GDI screen rotation and DirectDraw (see the Windows CE Help documentation, 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/DIRECTDRAW CETK test suite. It’s recommended to run these CETK tests with rotation support disabled or under 0 rotation degree.

An application tvhotkey.exe is provided. It is a startup application after bootup and listens to the hot key event (ALT + SPACE) to allow switching between LCD and NTSC display modes. This feature can be disabled by setting environment variable "BSP\_NOTVHOTKEY" to “1”.

## 10.4 Software Operation

### 10.4.1 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.

#### 10.4.1.1 Using the GDI

The Graphics Device Interface provides basic controls for the display of text and graphics. Refer to the following help section for information on using the GDI:

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

#### 10.4.1.2 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 Embedded CE Features > 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 YUV pixel format
- Overlaying using a color key for the overlay surface for RGB colors
- 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 YUV overlay data to RGB. This conversion is required in order to combine the overlay data with RGB graphics plane data in the IPU SDC.

- Resizing of the overlay surface
- Rotation of the overlay surface (used when the screen orientation is rotated)
- Resizing and rotation of the primary graphics surface when TV Output mode is enabled and active. This is required to obtain a 640x480 resolution image for output to a TV.

#### NOTE

Setting environment variable "BSP\_DISPLAY\_DELAYFLIP" to 1 enables the feature to support delay flip (which is synchronous) for overlay surfaces. Other than this environment variable, application should explicitly set DDFLIP\_WAITNOTBUSY flag when flipping.

### 10.4.1.3 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 CE Help documentation:

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

## 10.4.2 Configuring the Display

The display is configured based on the **PanelType** registry key, which is described in [Section 10.4.2.2, “Display Registry Settings](#). The **PanelType** registry key indicates the display panel that is being used. There is only one supported display panel: The EPSON L4F00242T03 VGA LCD panel.

### 10.4.2.1 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 supports rotation. If it is set to FALSE, it does not support rotation.

#### NOTE

Due to lack of support for the co-existence of GDI screen rotation and DirectDraw (see the Windows CE Help documentation, 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/DIRECTDRAW CETK test suite. It is recommended to run these CETK tests with rotation support disabled or under 0 rotation degree.

### 10.4.2.2 Display Registry Settings

The following registry keys are optionally included, depending on the display panel catalog item included in the OS design.

If the Epson 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      ; Epson VGA dumb Panel
```

If TV Output is included in the OS design, the following registry keys are also included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
"TVSupported"=dword:1 ; Flag indicates TV out mode is supported
```

### 10.4.3 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) codes like IOCTL\_POWER\_CAPABILITIES, IOCTL\_POWER\_QUERY, IOCTL\_POWER\_GET and IOCTL\_POWER\_SET.

#### 10.4.3.1 PowerUp

This function is not implemented for the display driver.

#### 10.4.3.2 PowerDown

This function is not implemented for the display driver.

#### 10.4.3.3 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. The IPU's Display Interface (DI) and SDC modules are enabled.
- D4 – The DI and SDC modules of the IPU are disabled. The display panel is disabled.

## 10.5 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 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.

## 10.5.1 Unit Test Hardware

Table 10-2 lists the required hardware to run the GDI and DirectDraw tests.

**Table 10-2. Hardware Requirements**

Requirements	Description
EPSON L4F00242T03 VGA Panel	Display panel required for display of graphics data

## 10.5.2 Unit Test Software

### 10.5.2.1 GDI Tests

Table 10-3 lists the required software to run the GDI tests.

**Table 10-3. GDI Test Software Requirements**

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
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.

### 10.5.2.2 DirectDraw Tests

Table 10-4 lists the software required to run the DirectDraw tests.

**Table 10-4. Direct Draw Test Software Requirements**

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
DDrawTK.dll	Test .dll file

### 10.5.2.3 Windows Media Player Tests

Table 10-5 lists the software required to perform WMV playback with Windows Media Player.

**Table 10-5. Windows Media Player Test Software Requirements**

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



### 10.5.3 Building the Display Tests

The GDI and DirectDraw tests come pre-built as part of the CETK. Ensure you are using the latest CETK suite. No steps are required to build these tests. Refer to the help documentation for more detailed information:

**Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Display Tests.**

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.

### 10.5.4 Running the Display Tests

#### 10.5.4.1 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 **Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Display Tests > Graphics Device Interface Test** in the CE Help documentation.

#### 10.5.4.2 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 **Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Display Tests > DirectDraw Test** in the CE Help documentation.

#### 10.5.4.3 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” pops 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.

## 10.6 Display Driver API Reference

Documentation for the display driver APIs can be found within the CE Help documentation. 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 CE Help documentation:

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

Reference information on DirectDraw functions, callbacks, and structures can be found at the following location in the CE Help documentation:

**Windows Embedded CE Features > Graphics > DirectDraw**

**Windows Embedded CE Features > Shell, GWES, and User Interface > Graphics, Windowing and Events (GWES) > GWES Reference > GDI Reference**

# Chapter 11

## Dynamic Voltage and Frequency Control (DVFC) Driver

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 APM (Advanced Power Management). The DVFC driver plays an important role in the reduction of i.MX31 CPU power consumption by dynamically adjusting the voltage and frequency settings of the system. The DVFC driver responds to DVFS that is monitoring CPU loading and process performance of i.MX31 ARM platform. The APM algorithm monitors clock tree changes in the system, and applies most adapting performance level to system.

### 11.1 DVFC Driver Summary

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

**Table 11-1. DVFC Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 CSP Driver Path	N/A
CSP Driver Path	..\platform\common\src\soc\freescale\mx31_fsl_v1\dvfc
CSP Static Library	dvfc_mx31_fsl_v1.lib
Platform Driver Path	...\PLATFORM\<TGTPLAT>\SRC\DRIVERS\DVFC\MC13783
Import Library	pmicSdk_mc13783.lib
Driver DLL	dvfc_mc13783.dll
Catalog Item	Third Party > BSPs > Freescale <TGTPLAT> > Device Drivers > DVFC > MC13783 DVFC
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_PMIC_MC13783 = 1 BSP_DVFC_MC13783 = 1

### 11.2 Supported Functionality

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

- Supports APM that depends on clock tree state change
- Supports DVFS and bus scale for power conservation
- Provides integrated voltage control supplied from MC13783
- Exposes interface for CE6 power manager
- Supports D0, D1, D2, and D4 driver power states
- Supports APM and DVFS use MPLL/SPLL switching

## 11.3 Hardware Operation

The DVFC driver is dependent upon the MC13783 interface for dynamic voltage control. The MC13783 chip must be present on the i.MX31 3-stack CPU board. The i.MX31 CPU board must be configured to source power from the MC13783. Refer to the **Windows CE BSP for i.MX31 3-Stack User's Guide** for the proper board configuration.

### 11.3.1 Pin Settings and Conflicts

#### 11.3.1.1 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 connected to the PMIC power ready notification signal and is used by the DVFS hardware to determine when the voltage setting is reached. This pin should not be configured for GPIO purposes.

#### 11.3.1.2 PMIC and DVS Pin settings

The signals of DVS0 and DVS1 are configured in combination mode. MC13783 switches SW1 and SW2 that are connected together provide four voltage selection.

## 11.4 Software Operation

### 11.4.1 Loading and Initialization

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

### 11.4.2 Clock Tree Dependency

The DVFC driver uses APM algorithm to control and regulate system performance. All the clock modules are mapped to an unique request system setpoint. With manipulating clock tree changing event, DVFC driver is able to select the best satisfied setpoint to all BSP modules with optimized performance and power.

### 11.4.3 Processor Workload Tracking

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 of the load tracking hardware, DVFS hardware can control the CPU loading characteristics that trigger DVFS transitions. The DVFS can trigger system to raise or lower setpoint based on CPU workload.

## 11.4.4 Setpoint Consideration

There are four setpoints defined in the DVFC driver to select processor work performance. [Table 11-2](#) describe the setpoint definition and power conservation schema in DVFC driver.

**Table 11-2. DVFC Setpoint Definition**

Name	Performance (MHz)	Voltage (V)	Power Target
TURBO	528/132/66	1.600	Fastest for speed, high voltage
HIGH	396/132/66	1.350	Reduced speed and voltage to save power for most uses case
MEDIUM	132/66/66	1.300	System bus scale
LOW	132/33/33	1.250	Peripheral bus scale

## 11.4.5 Lock and Performance

Since system clock tree status can be changed at any time, DVFC driver holds a exclusion lock to DDK threads when it is updating system setpoint.

The setpoint updating performance depends on six factors:

- Communication efficiency to PMIC
- Regulator speed in PMIC for voltage ready
- PLL lock and switch time
- Mutex lock of DDK threads
- Critical section of Shared CSPI to PMIC
- Bus ready time

DVFS hardware also triggers asynchronous events to request setpoint change. The DVFC daemon thread synchronizes to DVFS with mutex protection to change the system state. Because the APM algorithm priority is higher than DVFS, DVFS gives up the mutex lock if it conflicts with APM setpoint turning.

## 11.4.6 DDK Interface

The DVFC driver 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:

- DDKClockSetpointRequest
- DDKClockSetpointRelease

## 11.4.7 Power Management

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

#### **11.4.7.1 PowerUp**

This function is not implemented for the DVFC driver.

#### **11.4.7.2 PowerDown**

This function is not implemented for the DVFC driver.

#### **11.4.7.3 IOCTL\_POWER\_CAPABILITIES**

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

#### **11.4.7.4 IOCTL\_POWER\_SET**

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

#### **11.4.7.5 IOCTL\_POWER\_GET**

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

### **11.5 Unit Test**

No unit test cases provided.

## Chapter 12

# FM Radio Driver

The FM radio driver is used to control the Si4702 chip, and is compatible with the Stream Interface driver framework. This chapter provides information about developing the FM radio application, which interfaces directly to the hardware component Si4702 chip.

## 12.1 Radio Driver Summary

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

**Table 12-1. Radio Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\RADIO
Import Library	N/A
Driver DLL	fm_radio.dll
Catalog Item	Third Party – BSP > Freescale i.MX31 3DS:ARMV4I > Device Drivers > Radio Driver > Si4702 FM Radio
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_RADIO=1

## 12.2 Supported Functionality

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

- Conforms to the Device Manager streams interface
- Supports the Si4702 chip
- Supports the main functions of FM radio: power on/off, set frequency, set volume, muted, auto scan

## 12.3 Hardware Operation

The driver uses I<sup>2</sup>C to interact with Si4702 hardware. Details refer to *Silicon Laboratories Si4702.pdf*.

## 12.4 Software Operation

The only interface to control the radio driver is IOCTLs.

## 12.4.1 Radio Driver Registry Settings

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

; These registry entries load the FM Radio driver. The IClass value be GUID for generic ; power-managed devices.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\RADIO]
"Prefix"="RDO"
"Dll"="fm_radio.dll"
"Index"=dword:1
"Order"=dword:30
"IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

## 12.4.2 Power Management

The primary method for limiting power consumption in the radio module is to power down the chip when no longer using the FM radio driver. The application can call `IOCTL_SET_POWER` with parameter `POWER_OFF` to power down the chip.

### 12.4.2.1 PowerUp

This function is not implemented for the radio driver.

### 12.4.2.2 PowerDown

This function is not implemented for the radio driver.

### 12.4.2.3 IOCTL\_POWER\_CAPABILITIES

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

### 12.4.2.4 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 the radio driver. Any request that is not D0 is changed to a D4 request and results in the system entering into a suspend state, while for a value of D0 the system resumes.

### 12.4.2.5 IOCTL\_POWER\_GET

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

## 12.5 Unit Test

The Radio CETK test cases verify the functionality of the radio driver for Si4702 chip. Also the FM Radio Application can be used to verify the radio driver. Refer to FM Radio Application section of user guide.



## 12.5.1 Unit Test Hardware

The i.MX31 3-Stack board is required.

## 12.5.2 Building the Radio Tests

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

Build an OS image for the desired configuration.

1. Within Platform Builder, go to the **Build** menu option and select the **Open Release Directory** menu option. This opens a DOS prompt.
2. Change to the Radio Tests directory. (\WINCE600\SUPPORT\MX31\TESTS\RADIO)
3. Enter **set WINCEREL=1** on the command prompt and hit return. This copies the built DLL to the flat release directory.
4. Enter the build command (*build -c*) at the prompt and press return.

After the build completes, the radio\_test.dll file is located in the \$(\_FLATRELEASEDIR) directory.

## 12.5.3 Running the Radio Tests

The command line for running the radio tests is *tux -o -d radio\_test*. You can provide an additional option *-f* if you wish to redirect the test results to a file. Radio tests do not contain any test specific command line options.

## 12.6 Radio IOCTL Reference

This section consists of descriptions for the RADIO I/O control codes (IOCTLs). These IOCTLs are used in calls to DeviceIoControl to issue commands to the radio device modules. Only relevant parameters for the IOCTL have a description provided. Most of the IOCTLs are explained in the specific sections where they are most relevant.

### 12.6.1 Radio Driver IOCTLs

#### 12.6.1.1 RADIO\_IOCTL\_GET\_CAPS

This **DeviceIoControl** request gets capability of hardware.

##### Parameters

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	NULL
<i>pBufOut</i>	pointer to RADIO_CAPS type data return to caller

#### 12.6.1.2 RADIO\_IOCTL\_GET\_TUNER

This **DeviceIoControl** request gets tuner data of hardware.

**Parameters**

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	NULL
<i>pBufOut</i>	pointer to RADIO_TUNER type data return to caller

**12.6.1.3 RADIO\_IOCTL\_SET\_TUNER**

This **DeviceIoControl** request sets tuner data of hardware.

**Parameters**

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	pointer to RADIO_TUNER type data filled by caller
<i>pBufOut</i>	NULL

**12.6.1.4 RADIO\_IOCTL\_GET\_AUDIO**

This **DeviceIoControl** request gets audio data of hardware.

**Parameters**

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	NULL
<i>pBufOut</i>	pointer to RADIO_AUDIO type data return to caller

**12.6.1.5 RADIO\_IOCTL\_SET\_AUDIO**

This **DeviceIoControl** request sets audio data of hardware such as volume or muted.

**Parameters**

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	pointer to RADIO_AUDIO type data filled by caller
<i>pBufOut</i>	NULL

**12.6.1.6 RADIO\_IOCTL\_GET\_FREQ**

This **DeviceIoControl** request gets the current frequency of the hardware.

**Parameters**

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	NULL
<i>pBufOut</i>	pointer to the current frequency return to caller

### 12.6.1.7 RADIO\_IOCTL\_SET\_FREQ

This **DeviceIoControl** request tunes to the frequency.

#### Parameters

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	pointer to the frequency filled by caller
<i>pBufOut</i>	NULL

### 12.6.1.8 RADIO\_IOCTL\_GET\_POWER

This **DeviceIoControl** request gets the power state of hardware.

#### Parameters

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	NULL
<i>pBufOut</i>	pointer to RADIO_POWER type data return to caller

### 12.6.1.9 RADIO\_IOCTL\_SET\_POWER

This **DeviceIoControl** request sets power state of hardware.

#### Parameters

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	pointer to RADIO_POWER type data filled by caller
<i>pBufOut</i>	NULL

### 12.6.1.10 RADIO\_IOCTL\_AUTO\_TUNE

This **DeviceIoControl** request auto scan all available channels.

#### Parameters

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	pointer to RADIO_AUTOTUNE type data filled by caller.
<i>pBufOut</i>	NULL

### 12.6.1.11 RADIO\_IOCTL\_GET\_LAST\_ERROR

This **DeviceIoControl** returns the last return code.

#### Parameters

<i>hOpenContext</i>	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
<i>pBufIn</i>	NULL
<i>pBufOut</i>	pointer to the current return code returned to the caller

## 12.6.2 Radio Driver Structures

### 12.6.2.1 Radio Tuner Structure

```
typedef struct
{
    TCHAR    name[32]; // i.e "FM"
    INT32    band_id;
    UINT32   range_low; //KHZ
    UINT32   range_high; //KHZ
    UINT32   signal;
    UINT32   normal_signal; //acceptable signal
    UINT32   mode; //MONO, STEREO
    UINT32   reserved;
} RADIO_TUNER;
```

### 12.6.2.2 Radio Caps Structure

```
typedef struct
{
    TCHAR    driver[32]; // i.e. "Radio"
    TCHAR    chip[32]; // i.e. "Silicon Laboratories Si4702"
    UINT32   version; //
    UINT32   caps; // Device capabilities
    UINT32   bands;
    UINT32   reserved;
} RADIO_CAPS;
```

### 12.6.2.3 Radio Audio Structure

```
typedef struct
{
    UINT32   volume;
    UINT32   muted;
} RADIO_AUDIO;
```

### 12.6.2.4 Radio Power State Structure

```
typedef enum
{
    RADIO_POWER_OFF = 0,
    RADIO_POWER_ON
} RADIO_POWER;
```

### 12.6.2.5 Radio Auto Tune Structure

```
typedef enum
```

```
{
    RADIO_AUTOTUNE_FROM_BEGIN = 0,
    RADIO_AUTOTUNE_FROM_CUR,
    RADIO_AUTOTUNE_FROM_END
} RADIO_AUTOTUNE_POS;

typedef enum
{
    RADIO_AUTOTUNE_SEEKUP = 0,
    RADIO_AUTOTUNE_SEEKDOWN
} RADIO_AUTOTUNE_DIR;

typedef struct
{
    RADIO_AUTOTUNE_POS pos;
    RADIO_AUTOTUNE_DIR dir;
} RADIO_AUTOTUNE;
```



## Chapter 13

# 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. There is only one general purpose timer supported in i.MX31.

### 13.1 GPT Driver Summary

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

**Table 13-1. GPT Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 CSP Driver Path	..\SOC\FREESCALE\MXARM11_FSL_V1\GPT
CSP Driver Path	N/A
CSP Static Library	gpt_mxarm11_fsl_v1.lib
Platform Driver Path	..\PLATFORM\IMX313DS\SRC\DRIVERS\GPT
Import Library	N/A
Driver DLL	gpt.dll
Catalog Item	Third Party > BSPs > Freescale i.MX31 3DS: ARMV4I > Device Drivers > Timers > GPT
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_GPT=1

### 13.2 Supported Functionality

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

- Configured as a loadable .dll module so that other drivers can use the interface of the GPT driver
- Supports clock source selection, including external clock source
- Supports both reset and free-run mode count operation
- Supports two power management modes, power on and power off
- Supports the SDK interface
- The unit test cases are created for testing the GPT driver

### 13.3 Hardware Operation

Refer to the chapter on General Purpose Timer in the hardware specification document for detailed hardware operation and programming information.

### 13.3.1 Conflicts with Other Peripherals

No conflicts.

## 13.4 Software Operation

### NOTE

If Platform Builder profiling support is to be used, the GPT driver cannot be included in the workspace

### 13.4.1 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 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. It is necessary to include the `gptsdk_mxarm11_fsl_v1.lib` library to use this API.

### 13.4.2 Creating a Handle to the GPT

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, see the **GptOpenHandle** section under the GPT API reference.

### 13.4.3 Configuring the GPT

Configuring the GPT for communications involves selecting timer source by:

- Calling **GptSetTimerSrc** API
- Starting the timer and enabling the timer event trigger by calling **GptStart** API
- Showing the current timer source by calling **GptShowTimerSrc** API

Before this action can be taken, a handle to the GPT port must already be opened.

Call the **GptSetTimerSrc** API to select timer source.

```
// selecting the GPT source
GptSetTimerSrc(g_hGpt, pGptTimerSrcPkt) ;
```

Call the **GptStart** API to enable and start the timer.



```
// configuring and starting the GPT, the second parameter contains timer mode and
// timer length
GptStart(g_hGpt, pTimerConfig) ;
```

Call the **GptShowTimerSrc** API to show current timer source.

```
// showing current GPT timer source
GptShowTimerSrc(g_hGpt) ;
```

For more information on this API, see the **GptStart** section under the GPT API reference.

### 13.4.4 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*. The period has the unit of microsecond.

```
// 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, see **GptSetTimer** section of the GPT API reference.

### 13.4.5 Closing the Handle to the GPT

To close the GPT handle, call the **GptCloseHandle** API. Before performing the close operation, 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, see the **GptReleaseTimerEvent**, **GptStop** and **GptCloseHandle** section under the GPT API reference.

## 13.4.6 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()**.

### 13.4.6.1 PowerUp

This function restores the state of the GPT clocks back to the state before entering suspend. If the GPT was counting before suspend, GPT continues to count from the place where it was stopped.

### 13.4.6.2 PowerDown

This function disables the clock to the GPT module. If the GPT was counting, then the count value freezes at the point when the clock is removed.

### 13.4.6.3 IOCTL\_POWER\_CAPABILITIES

N/A

### 13.4.6.4 IOCTL\_POWER\_SET

N/A

### 13.4.6.5 IOCTL\_POWER\_GET

N/A

## 13.4.7 GPT Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\GPT]
"Prefix"="GPT"
"Dll"="gpt.dll"
"Index"=dword:1
```

## 13.5 Unit Test

The GPT tests verify that the GPT driver properly initializes and controls the general purpose timer.

### 13.5.1 Unit Test Hardware

Table 13-2 lists the required hardware to run the unit tests.

**Table 13-2. Hardware Requirements**

Requirements	Description
No additional hardware required	

## 13.5.2 Unit Test Software

Table 13-3 lists the required software to run the unit tests.

**Table 13-3. Software Requirements**

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
GPTTEST.dll	Test .dll file

## 13.5.3 Building the GPT Tests

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

Build an OS image for the desired configuration:

1. Within Platform Builder, go to the **Build** menu option and select the **Open Release Directory** menu option. This opens a DOS prompt.
2. Change to the GPT Tests directory: \WINCE600\SUPPORT\MX31\TESTS\GPT
3. Enter **set WINCEREL=1** on the command prompt and hit return. This copies the built DLL to the flat release directory.
4. Enter the build command at the prompt and press return.

After the build completes, the GPTTEST.dll file is located in the \$(\_FLATRELEASEDIR) directory.

## 13.5.4 Running the GPT Tests

The command line for running the GPT tests is *tux -o -d gptest*. The GPT tests do not contain any test specific command line options.

To add the GPT test to CETK, perform the following steps:

1. Go to the **Tests** menu and select **User Defined**.
2. Follow the wizard and add the GPTTEST.dll located in the release folder as the test module.
3. Follow the wizard until it finishes.

Table 13-4 describes the test cases contained in the GPT tests.

**Table 13-4. GPT Test Cases**

Test Case	Description
1: TST_StartBeforeCfg	Attempt to start the GPT timer without setting the timer period (expected failure)
2: TST_OpenMultipleHandle	Attempt to open multiple GPT Handles (expected failure)
3: TST_ComparewithSysTick	Check timer accuracy with system clock
4:TST_ChangeClockSrc	Run the timer with different timer source
5:TST_PeriodicMode	Periodic mode test

6: TST_FreerunMode	Free run mode test
7: TST_StopAndResume	Stop and resume test

## 13.6 GPT Driver API Reference

### 13.6.1 GPT Driver Functions

#### 13.6.1.1 GptOpenHandle

This API creates a handle to the GPT stream driver.

```
HANDLE GptOpenHandle(
    void
);
```

<b>Parameters</b>	This API accepts no parameters.
<b>Return Values:</b>	An open handle to the specified file indicates success. INVALID_HANDLE_VALUE indicates failure.
<b>Remarks</b>	Use the GptCloseHandle function to close the handle returned by GptOpenHandle().

#### 13.6.1.2 GptCreateTimerEvent

This API is used to create the GPT Timer event.

```
HANDLE GptCreateTimerEvent(
    HANDLE hGpt,
    LPTSTR eventName
);
```

<b>Parameters</b>	
<i>hGpt</i>	[in] Handle to the GPT driver returned by <b>GptOpenHandle</b> API.
<i>eventName</i>	[in] Pointer to a null-terminated string that specifies the name of the object.
<b>Return Values:</b>	A non-null handle to the specified event indicates success. NULL indicates failure.
<b>Remarks</b>	Use the <b>GptReleaseTimerEvent</b> 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.

#### 13.6.1.3 GptShowTimerSrc

This API show the current timer source for the GPT.

```
BOOL GptShowTimerSrc(
    HANDLE hGpt
);
```

**Parameters**

<i>hGpt</i>	[in] Handle to the GPT driver returned by <b>GptOpenHandle</b> API.
<b>Return Values:</b>	TRUE on success and FALSE indicates a failure.
<b>Remarks</b>	Prints a message indicating which clock source is selected.

**13.6.1.4 GptSetTimerSrc**

This API show the current timer source for the GPT.

```

BOOL GptSetTimerSrc(
    HANDLE hGpt,
    PGPT_TIMER_SRC_PKT pGptTimerSrcPkt
);

```

**Parameters**

<i>hGpt</i>	[in] Handle to the GPT driver returned by <b>GptOpenHandle</b> API.
<i>pGptTimerSrcPkt</i>	[in] An object of the <b>PGPT_TIMER_SRC_PKT</b> structure
<b>Return Values:</b>	TRUE on success and FALSE indicates a failure.
<b>Remarks</b>	Select clock source between CLK_HIFRQ, CLK_32K, CLK_IPG, CLK_EXT

**13.6.1.5 GptStart**

This API enables the GPT interrupt and starts the GPT timer.

```

BOOL GptStart(
    HANDLE hGpt,
    PGPT_Config pTimerConfig
);

```

**Parameters**

<i>hGpt</i>	[in] Handle to the GPT driver returned by <b>GptOpenHandle</b> API.
<i>pTimerConfig</i>	[in] An object of the <b>pGPT_Config</b> structure.
<b>Return Values:</b>	TRUE on success and FALSE indicates a failure.
<b>Remarks</b>	Set desired event trigger time and start GPT.

**13.6.1.6 GptUpdatePeriod**

This API updates the counter compare value on regarding to the current counter value and the new time length submitted.

```

BOOL GptUpdatePeriod(
    HANDLE hGpt,
    DWORD period
);

```

**Parameters**

<i>hGpt</i>	[in] Handle to the GPT driver returned by <b>GptOpenHandle</b> API.
<i>period</i>	[in] new time length (in micorsecond) submitted.

### 13.6.1.7 GptGetCounterValue

This API gets the current counter register value.

```
BOOL GptGetCounterValue(
    HANDLE hGpt,
    PDWORD pTimerCount
);
```

#### Parameters

*hGpt* [in] Handle to the GPT driver returned by **GptOpenHandle** API.  
*pTimerCount* [in] point to the variable which receives current counter value

### 13.6.1.8 GptResume

This API reactivates the GPT.

```
BOOL GptResume(
    HANDLE hGpt
);
```

#### Parameters

*hGpt* [in] Handle to the GPT driver returned by **GptOpenHandle** API.

**Remarks** Often called after a stop.

### 13.6.1.9 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.

### 13.6.1.10 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().

### 13.6.1.11 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().

## 13.6.2 GPT Driver Structures

### 13.6.2.1 GPT\_Config

```
typedef struct
{
    timerMode_c timerMode;
    UINT32 period;
} GPT_Config, *pGPT_Config;
```

#### Members

*timerMode* Selects between two supported modes: reset or periodic mode (timerModePeriodic) and free-running mode (timerModeFreeRunning).

*period* Counter period (in microsecond)

### 13.6.2.2 GPT\_TIMER\_SRC\_PKT

```
typedef struct
{
    timerSrc_c timerSrc;
} GPT_TIMER_SRC_PKT, *PGPT_TIMER_SRC_PKT;
```

#### Members

*timerSrc* Selects between 4 supported timer source, GPT\_IPGCLK, GPT\_HIGHCLK, GPT\_EXTCLK and GPT\_32KCLK





## Chapter 14

# Global Positioning System Driver

The global positioning system (GPS) enables a GPS receiver to determine its location, speed/direction, and time. This 3-Stack platform supports the BroadCom BCM4750 Single Chip A-GPS Solution. BCM4750 is an A-GPS solution that integrates a high performance A-GPS baseband signal processor with a low-noise GPS RF Tuner into a single CMOS die. BCM4750 delivers exceptional sensitivity (-162 dBm), low power consumption and fast time-to-first-fix (TTFF) in a small, inexpensive package.

The external GPS module is supported using the UART port and GPIO resources. Because the chipset features a host-based architecture, certain software components must be loaded onto the platform in order to enable full operation.

### 14.1 GPS Driver Summary

Most GPS software modules are provided in binary form only. This application also provides source code format for the driver that supports access to the hardware. To enable the GPS module, select the corresponding elements from the platform builder catalog for the current OS design. The binary files and the registry settings that correspond to the elements selected are included in the OS run-time image.

The GPS module uses UART on the 3-Stack platform. Reset and power on/ power off to the GPS module are controlled by the GPIO pins of the i.MX31. The GPS module functionality is segmented into subsystems. Not all of the subsystems need to be selected in order to enable GPS on the platform.

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

**Table 14-1. GPS Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 CSP Driver Path	N/A
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<tgtplat>\SRC\DRIVERS\GPS ..\PLATFORM\<tgtplat>\SRC\DRIVERS\GPSCTRL
Import Library	N/A
Driver DLL	Gpsct.exe;GpsNavigationLauncher.exe;GpsctServiceLauncher.exe;GpsctService.dll; Glvcdriver.dll;gpscontroldriver.dll LtoManager.exe, LtoManager.exe.config, LtoManagerLauncher.exe, log4net.dll, OpenNetCF_GL.dll, OpenNetCF.Net_GL.dll, OpenNetCF.Windows.Forms_GL.dll
Catalog Items	Third Party > BSPs > Freescale <tgtplat> > Device Drivers > GPS > GPS core drivers Third Party > BSPs > Freescale <tgtplat> > Device Drivers > GPS > GPS control driver

SYSGEN Dependency	N/A
BSP Environment Variables	BSP_GPS_COREDRIVER = 1 BSP_GPS_CONTROL_DRIVER= 1 BSP_SERIAL_UART3 = 1

Figure 14-1 shows the architecture of GPS driver, showing the following layers in the GPS software system:

- Application layer
- GPS core driver layer
- GPS HAL driver layer

### 14.1.1 Application layer

Handset applications, TCP/IP stack, and GSM layer3 in Figure 14-1 belong to application layer. Handset applications, such as VisualGpsce.exe or any other mapping software, can receive standard NMEA data to show position with a friendly user interface. TCP/IP stack and GSM layer3 can provide A-GPS navigation service to enhance GPS functionality even when satellite signal is not strong enough to get fix.

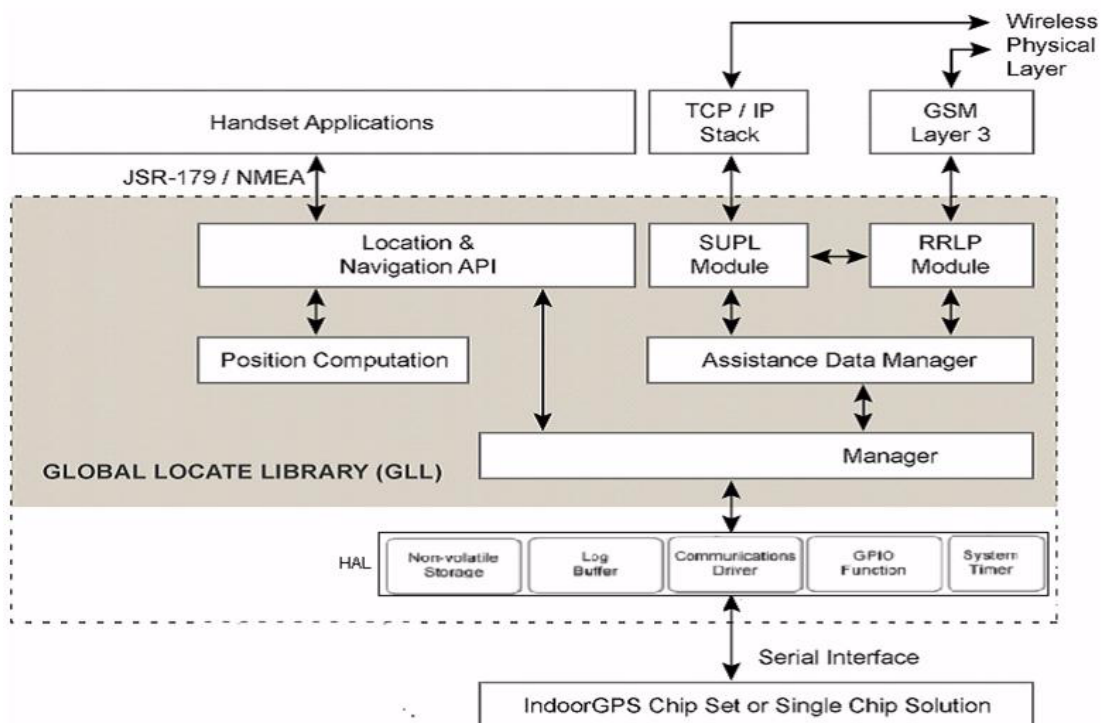


Figure 14-1. Software Architecture of GPS Driver

## 14.1.2 GPS Core Driver Layer

The middle section of [Figure 14-1](#) shows the Global Local Library (GLL) which belongs to GPS core driver layer. The GPS core driver runs at host and communicates with GPS chip by calling GPS HAL driver. The driver is used for position calculation and assistance data management.

## 14.1.3 GPS HAL driver layer

GPS HAL drivers provide hardware related resource, such as serial port driver, non-volatile storage, and GPIO functions. Only GPIO functions are provided here to control GPS power state and reset. The driver is called `gpscontroldriver.dll`, and source code is available at

```
..\PLATFORM\<tgtplat>\SRC\DRIVERS\GPSCTRL.
```

## 14.2 Supported Functionality

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

- Integrates the BCM4750 GPS module from BroadCom company
- Supports power management mode full on/full off

## 14.3 Hardware Operation

### 14.3.1 UART Port

For i.MX31-3DS board, UART3 is used to communicate with the GPS module. If a different UART is used for this purpose, then the following registry should be changed correspondingly:

```
..\PLATFORM\<tgtplat>\SRC\DRIVERS\GPS\GlobalLocate-Gpsct-flatrom.reg:
```

```
"GpsComPort"="COMx:"
```

Here “x” should be specified according to the UART actually used ("COM3:").

### 14.3.2 GPIO Control

Some GPIO pins of the 3-Stack platform are used to control the GPS module ([Table 14-2](#)). If different pins are used for such purpose, then some source code must be updated to reflect the difference. Refer to the following source file for details: `..\PLATFORM\<tgtplat>\SRC\DRIVERS\GPS\GpsCtlPdd.cpp`

**Table 14-2. GPIO Control**

GPIO Name	PIN	Value Description
BSP_GPIO_GPS_RESET	MCU2_15	0: Reset of GPS module is asserted 1: Reset of GPS module is de-asserted
BSP_GPIO_PWR_EN_GPS	MCU3_2	1: GPS module is powered on 0: GPS module is powered off
FM_CLK_EN	MCU2_3	Must be 1 to enable 32KHz RTC

### 14.3.3 Conflicts with Other Peripherals

No conflicts.

## 14.4 Software Operation

### 14.4.1 Communicating with the GPS Module

Software applications communicate with the GPS module through a virtual COM port (i.e. COM8). The virtual COM port is a standard stream interface driver, and is thus accessed through the file system APIs. For example, the Win32 API CreateFile() call can be used to obtain a handle and ReadFile() can be used to read the NMEA data stream output by the GPS module.

### 14.4.2 Power Management

The 3-Stack platform functions GPS\_PowerUp and GPS\_PowerDown are used to bring the GPS module into and out of standby mode. The code is designed to keep the power consumption of the GPS module at a minimal level when the standby power state is invoked.

### 14.4.3 GPS Driver Registry Settings

#### 14.4.3.1 Configuration Registry Keys

Contact BroadCom for details.

## 14.5 Unit Test

A navigation application is necessary to test GPS driver. Freescale does not provide a navigation application. The user is responsible for providing a navigation application (contact BroadCom for more information).

## Chapter 15

# Inter-Integrated Circuit (I2C) Driver

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

### 15.1 I2C Driver Summary

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

**Table 15-1. I2C Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 CSP Driver Path	..\PLATFORM\common\src\soc\freescale\mxarm11_fsl_v1\i2c
CSP Driver Path	..\PLATFORM\common\src\soc\freescale\<TGTSOC>\DRIVERS\I2C
CSP Static Libraries	i2c_mxarm11_fsl_v1.lib, i2c_<TGTSOC>.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\I2C
Import Library	N/A
Driver DLL	i2c.dll
Catalog Item	Third Party → BSP → Freescale i.MX31 3DS:ARMV4I → Device Drivers → I2C Bus
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_I2CBUS=1

### 15.2 Requirements

The I2C driver should meet the following requirements:

1. Support the I2C communication protocol.
2. Support multiple I2C controllers.
3. Support the I2C master mode of operation.
4. Not support the I2C slave mode of operation.
5. Function as a stream interface driver implementing the programming interface defined in this document.
6. Support two power management modes, full on and full off.

### 15.3 Hardware Operation

Refer to the chapter on I2C 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

The i.MX31 platform contains three I2C modules, but only one of these modules may be used on the i.MX31 PDK board, the I2C1 module. 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.

## 15.4 Software Operation

### 15.4.1 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.

### 15.4.2 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, complete the following steps:

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.

### 15.4.3 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, complete the following steps:

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);
```

### 15.4.4 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, complete the following steps:

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 through 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.

### 15.4.5 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.

### 15.4.6 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 6.0 <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.

#### 15.4.6.1 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.

#### 15.4.6.2 PowerDown

This function is not implemented for the I2C driver.

#### 15.4.6.3 IOCTL\_POWER\_SET

This function is not implemented for the I2C driver.

### 15.4.7 I2C Registry Settings

The following registry keys are required to properly load the I2C1 module.

```
IF BSP_I2CBUS
; @XIPREGION IF PACKAGE_OEMDRIVERS
```

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\I2C1]
    "Prefix"="I2C"
    "Dll"="i2c.dll"
    "Index"=dword:1
    "Order"=dword:4

ENDIF
```

## 15.5 Unit Test

No CETK Test for I2C.

### NOTE

The camera module uses the I2C interface to control its setting, so the I2C function can be verified by the camera module.

## 15.6 I2C Driver API Reference

### 15.6.1 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.

#### 15.6.1.1 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

<i>lpOutBuffer</i>	Pointer to the divisor index. The true clock frequency is platform dependent. Refer to I2C specification for more information.
<i>nOutBufferSize</i>	Size in bytes of the divisor index.

#### 15.6.1.2 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

<i>lpOutBuffer</i>	Pointer to the current I2C device address. The valid range of the address is [0x00, 0x7F].
<i>nOutBufferSize</i>	Size in bytes of the I2C device address.

### 15.6.1.3 I2C\_IOCTL\_IS\_MASTER

This **DeviceIoControl** request determines whether the I2C is currently in Master mode.

#### Parameters

<i>lpOutBuffer</i>	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.
<i>nOutBufferSize</i>	Size in bytes of the return value. This should be one byte.

### 15.6.1.4 I2C\_IOCTL\_IS\_SLAVE

This **DeviceIoControl** request determines whether the I2C is currently in Slave mode.

#### Parameters

<i>lpOutBuffer</i>	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.
<i>nOutBufferSize</i>	Size in bytes of the return value. This should be one byte.

### 15.6.1.5 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.

### 15.6.1.6 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, use the macro **I2C\_MACRO\_SET\_FREQUENCY**.

#### Parameters

<i>lpInBuffer</i>	Pointer to the divisor index. Refer to the I2C specification to obtain the true clock frequency.
<i>nInBufferSize</i>	Size in bytes of the divisor index.

### 15.6.1.7 I2C\_IOCTL\_SET\_FREQUENCY

This **DeviceIoControl** request estimates the nearest clock rate acceptable for the I2C device and initializes the I2C device to use the estimated clock rate divisor. If the estimated clock rate divisor index is required, refer to the macro **I2C\_MACRO\_GET\_CLOCK\_RATE** to determine the estimated index.

#### Parameters

<i>lpInBuffer</i>	Pointer to the desired I2C frequency.
<i>nInBufferSize</i>	Size in bytes of the I2C frequency requested.

### 15.6.1.8 I2C\_IOCTL\_SET\_MASTER\_MODE

This **DeviceIoControl** request sets the I2C device to Master mode.

### 15.6.1.9 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.

### 15.6.1.10 I2C\_IOCTL\_SET\_SLAVE\_MODE

This **DeviceIoControl** request sets the I2C device to Slave mode.

### 15.6.1.11 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.

## 15.6.2 I2C Driver Macros

### 15.6.2.1 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.

### 15.6.2.2 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.

### 15.6.2.3 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.

### 15.6.2.4 I2C\_MACRO\_IS\_SLAVE

This macro determines whether the I2C is currently in Slave mode.

```
I2C_MACRO_IS_SLAVE(  
    HANDLE hDev,  
    BOOL bIsSlave  
);
```

**Parameters**

<i>hDev</i>	The I2C device handle retrieved from CreateFile().
<i>bIsSlave</i>	TRUE if the I2C device is in Slave mode.
<b>Return Values</b>	Returns TRUE or FALSE. If the result is TRUE, the operation is successful.

**15.6.2.5 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**

<i>hDev</i>	The I2C device handle retrieved from CreateFile().
<b>Return Values</b>	Returns TRUE or FALSE. If the result is TRUE, the operation is successful.

**15.6.2.6 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**

<i>hDev</i>	The I2C device handle retrieved from CreateFile().
<i>wClkRate</i>	Contains the divisor index. Refer to the I2C specification to obtain the true clock frequency.
<b>Return Values</b>	Returns TRUE or FALSE. If the result is TRUE, the operation is successful.
<b>Remarks</b>	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, use the macro I2C_MACRO_SET_FREQUENCY.

**15.6.2.7 I2C\_MACRO\_SET\_FREQUENCY**

This macro will estimate the nearest clock rate acceptable for the I2C device and initialize the I2C device to use the estimated clock rate divisor. If the estimated clock rate divisor index is required, 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.

### 15.6.2.8 I2C\_MACRO\_SET\_MASTER\_MODE

This macro set the I2C device to Master mode.

```
I2C_MACRO_SET_MASTER_MODE(  
    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.

### 15.6.2.9 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.

### 15.6.2.10 I2C\_MACRO\_SET\_SLAVE\_MODE

This macro sets the I2C device to Slave mode.

```
I2C_MACRO_SET_SLAVE_MODE(  
    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.

### 15.6.2.11 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.

## 15.6.3 I2C Driver Structures

### 15.6.3.1 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.

### 15.6.3.2 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

<b>pI2CPackets</b>	A pointer to an array of I2C_PACKET objects.
<b>iNumPackets</b>	The number of I2C_PACKET objects pointed to by <i>pI2CPackets</i> .

## Chapter 16

# 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.

### 16.1 Keypad Driver Summary

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

**Table 16-1. Keypad Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\FREESCALE\MXARM11_FSL_V1\KEYBD
CSP Driver Path	N/A
CSP Static Library	Keypad_mxarm11_fsl_v1.lib PddList_mxarm11_fsl_v1.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\KEYBD
Import Library	N/A
Driver DLL	Kbdmouse.dll
Catalog Item	Third Party → BSP → Freescale i.MX31 3DS: ARMV4I → Device Drivers → Input Devices → Keyboard/Mouse > 3DS Keypad
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_KBDMOUSE_EVBKPD=1

### 16.2 Requirements

The keypad driver should meet the following requirements:

1. Conform to the Microsoft Layout Manager Interface.
2. Support multiple simultaneous key presses.
3. Support two power management modes, full on and full off.

### 16.3 Hardware Operation

Refer to the chapter on the KPP in the hardware specification document for detailed operation and programming information.

### 16.3.1 The Keypad

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

The 9-key keypad is located in the personality board and the mapping is as follows:

**Table 16-2. Keypad Labels and Key Values**

Label	Key value
S7(up)	UP
S8(down)	DOWN
S10(left)	LEFT
S9(right)	RIGHT
S11(enter)	ENTER
S12(menu 1)	ALT
S15(menu 2)	TAB
S16(menu 3)	SPACE
S17(menu 4)	ESC

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:

**Table 16-3. ALT Keystroke Combinations**

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.

### 16.3.2 Conflicts with other SoC peripherals

No conflicts.

## 16.4 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 CE help documentation at the following location:

**“Developing a Device Driver → Windows Embedded CE Drivers → Keyboard Drivers → Keyboard Driver Development Concepts”**

### 16.4.1 Keypad Scan Codes and Virtual Keys

Each key on the keypad has a unique scan code, which is added to a buffer whenever that key is pressed or released. These scan codes, which are hardware-specific, are 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 :

**Table 16-4. Keypad Scan Codes and Virtual Keys**

Key	Keypad Scan Code	Virtual Key
UP	0	VK_UP
DOWN	3	VK_DOWN
LEFT	4	VK_LEFT
RIGHT	1	VK_RIGHT
ENTER	7	VK_RETURN
ALT	2	VK_MENU
ESC	11	VK_ESCAPE
TAB	5	VK_TAB
SPACE	8	VK_SPACE

### 16.4.2 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.

#### 16.4.2.1 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.

#### 16.4.2.2 BSPKppPowerOff

This function powers down the keypad. But before turning off the module, the current state of the clock settings for this module is saved and then there is a delay until the keypad does not report any key-down/key-up event. Then the clocks to this module are turned off.

### 16.4.2.3 IOCTL\_POWER\_CAPABILITIES

N/A

### 16.4.2.4 IOCTL\_POWER\_SET

N/A

### 16.4.2.5 IOCTL\_POWER\_GET

N/A

## 16.4.3 Keypad Registry Settings

The following registry keys are required to properly load the 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"
```

## 16.5 Unit Test

As the keypad has only 9 keys, it is not a full-function keypad and it cannot pass the Keyboard Test included as part of the Windows Embedded CE 6.0 Test Kit (CETK). A specific procedure is designed to test all keys.

### 16.5.1 Unit Test Hardware

N/A

### 16.5.2 Unit Test Software

N/A

## 16.5.3 Building the Keyboard Tests

N/A

## 16.5.4 Running the Keyboard Tests

The procedure for keyboard tests is as follows:

1. Run the application "Microsoft WordPad".
2. Input "Tab".
3. Input "Space".
4. Input "Alt" to open the menu bar.
5. Run the application "Internet Explorer".
6. Open the help document by click the question mark on the "Internet Explorer" application.
7. Input the "ESC" to quit from help document.
8. Input the "Alt + Tab" to call the "Task Manager".
9. Quit the application "Microsoft WordPad". In the pop up dialog box, click the "Yes" button.

## 16.6 Keypad Driver API Reference

Detailed reference information for the keypad driver may be found in CE help documentation at the following location:

**Developing a Device Driver** → **Windows Embedded CE Drivers** → **Keyboard Drivers** → **Keyboard Driver Reference**

### 16.6.1 Keypad PDD Functions

The following table shows a mapping of keyboard PDD functions to the functions used in the keypad driver:

**Table 16-5. Keypad PDD Pointers and Driver Functions**

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 17

# LAN9217 Product Ethernet Driver

The LAN9217 Product Ethernet driver is used for connectivity with an IEEE 802.3 Ethernet using the SMSC LAN9217 Ethernet Controller. The driver provides support to communicate with the Ethernet at 10/100 Mbps speed, as the LAN9217 Ethernet Controller is a 10Base-T/100 Base-TX Ethernet controller. The driver makes use of the LAN9217 internal MII-compatible transceiver.

The LAN9217 Product Ethernet driver is a NDIS 5.0-compliant miniport driver.

## 17.1 LAN9217 Product Ethernet Driver Summary

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

**Table 17-1. LAN9217 Product Ethernet Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 CSP Driver Path	N/A
<TGTPLAT> CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\LAN9217
Import Library	ndis.lib
Driver DLL	lan9217.dll
Catalog Item	Catalog → Third Party → BSP → Freescale <TGTPLAT>: ARMV4I → Device Drivers → Ethernet LAN9217 Driver
SYSGEN Dependency	SYSGEN_NDIS=1 SYSGEN_TCPIP=1 SYSGEN_WINSOCK=1
BSP Environment Variables	BSP_ETHER_LAN9217=1

## 17.2 Requirements

The LAN9217 Product Ethernet driver should meet the following requirements:

- Conform to the Microsoft Network Driver Interface Specification (NDIS) architecture in Windows Embedded CE. 6.0
- Support IEEE 802.3 Ethernet protocols for communication.
- Support two power management modes, full on and full off.

## 17.3 Hardware Operation

The LAN9217 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 LAN9217 data sheet for detailed operation and programming information.

### 17.3.1 Conflicts with other SoC peripherals

#### 17.3.1.1 i.MX31 Peripheral Conflicts

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

## 17.4 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 Embedded CE Drivers** → **Network Drivers** → **Network Driver Development Concepts** → **Miniports, Intermediate Drivers, and Protocol Drivers**.

### 17.4.1 Power Management

The power management is currently not implemented for the LAN9217 Product Ethernet driver.

### 17.4.2 Product Ethernet Registry Settings

The following registry keys are required to properly load the LAN9217 Product Ethernet driver and to configure the TCP/IP for Ethernet interface. In the following specimen, a dynamic IP address is assigned using DHCP, the variable `EnabledDHCP` should be set to 1.

```
[HKEY_LOCAL_MACHINE\Comm\lan9217]
    "DisplayName"="lan9217 Ethernet Driver"
    "Group"="NDIS"
    "ImagePath"="lan9217.dll"

[HKEY_LOCAL_MACHINE\Comm\lan9217\Linkage]
    "Route"=multi_sz:"lan9217"

[HKEY_LOCAL_MACHINE\Comm\lan9217]
    "DisplayName"="lan9217 Ethernet Driver"
    "Group"="NDIS"
    "ImagePath"="lan9217.dll"

[HKEY_LOCAL_MACHINE\Comm\lan9217\Parms]
    "BusNumber"=dword:0
    "BusType"=dword:0
    "InterruptNumber"=dword:1; pio interrupt
    "IoBaseAddress"=dword:B6000000 ; ETHERNET_BASE (Physical Addr)
    "PhyAddress"=dword:FF          ; PHY address (0x20:Auto, 0xFF:Internal)
    "RxDMAMode"=dword:0 ; 1-DMA, 0-PIO
    "TxDMAMode"=dword:0 ; 1-DMA, 0-PIO
    "FlowControl"=dword:1         ; 1-Enabled, 0-Disabled
```

```
; LinkMode will replace Duplex, Speed and FlowControl
; bit7: RESERVED, bit6: ANEG, bit5: ASymmetric Pause, bit4: Symmetric Pause
; bit3: 100FD, bit2: 100HD, bit1: 10FD, bit0: 10HD
"LinkMode"=dword:7F
```

```
[HKEY_LOCAL_MACHINE\Comm\lan9217\Parms\TcpIp]
"EnabledDHCP"=dword:1
"IpAddress"="0.0.0.0"
"Subnetmask"="0.0.0.0"
"DefaultGateway"="0.0.0.0"
"UseZeroBroadcast"=dword:0
```

## 17.5 Unit Test

The LAN9217 Product Ethernet driver is tested using the following:

1. Network utilities/operations: Ping to and from the 3DS device, FTP transfers (file put and get) with 3DS device as FTP server and Internet browsing with Pocket Internet Explorer on the 3DS device.
2. Winsock CETK test cases: Winsock 2.0 Test (v4/v6) and Winsock Performance Test with 3DS device as client.

### 17.5.1 Unit Test Hardware

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

**Table 17-2. Unit Test Hardware Requirements**

Requirements	Description
3DS board with LAN9217	Board that hosts the LAN9217 Product Ethernet driver
PC/machine	To act as counterpart for network operation
An Ethernet or a cross-Ethernet cable	To form an Ethernet

### 17.5.2 Unit Test Software

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

**Table 17-3. Unit Test Software Requirements**

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
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

## 17.5.3 Building the LAN9217 Product Ethernet Tests

### 17.5.3.1 Network utilities related tests

The following registry entries need 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 Core OS → CEBASE → Communication Services and Networking → Networking - General:  
 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 Core OS → CEBASE > Communication Services and Networking > Servers:  
 FTP Server

For Video streaming to work, the following components need to be enabled in the OS design:

Under Core OS → CEBASE → Graphics and Multimedia Technologies → Media:  
 Media Formats → AVI Filter  
 Streaming Media Playback  
 Video Codecs and Renderers → Video/Image Compression Manager  
 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

### 17.5.3.2 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\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4I

### 17.5.3.3 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\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4I

## 17.5.4 Running the LAN9217 Product Ethernet Tests

### 17.5.4.1 Network utilities-related tests

#### 17.5.4.1.1 Ping tests

The ping tests can be run as usual from the 3DS device as well as from the PC side.

#### 17.5.4.1.2 Browsing

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

- DNS servers in the TCP/IP properties of LAN9217 network interface (Control Panel → Network and Dial-up Connections)
- Proxy server, if used in the network used for test, on the Pocket Internet explorer.

#### 17.5.4.1.3 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:

### 17.5.4.2 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.

### 17.5.4.3 Winsock 2.0 Test (v4/v6)

The test can be executed on the <TGTSOC> device using *tux -o -d Ws2bvt.dll* 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.

#### 17.5.4.4 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 LAN9217"` 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.

## 17.6 LAN9217 Product Ethernet Driver API Reference

The LAN9217 Product Ethernet driver conforms to NDIS 5.0 specification by Microsoft for the miniport network drivers.

## 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 compatible with the Epson L4F00242T03 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 call transport, synchronization, and OS integration issues; the driver manages 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 i.MX31 MBX supports the D3DM and OpenGL ES in Windows Embedded CE 6.0.

### 18.1 Direct3D Mobile/OpenGL ES Drivers Summary

The following table identifies the source code location, library dependencies, and other BSP information.

**Table 18-1. Direct3D Mobile/OpenGL ES Drivers Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	IMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
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, IMGEG.L.dll, pvr_d3dm.dll, pvr_kernel.dll, um3partyif.dll
Catalog Item	Third Party BSPs Freescale <tgtplat> Device Drivers Display Epson L4F00242T03 Third Party BSPs Freescale <tgtplat> Device Drivers MBX MBX i.MX31 Base Driver Third Party BSPs Freescale <tgtplat> Device Drivers MBX MBX i.MX31 D3DM Core Third Party BSPs Freescale <tgtplat> Device Drivers MBX MBX i.MX31 Ogles Core

SYSGEN Dependency	SYSGEN_DDRAW=1 SYSGEN_D3DM=1
BSP Environment Variable	BSP_MBX BSP_DISPLAY_EPSON_L4F00242T03 = 1 for epson LCD Panel

## 18.2 Supported Functionality

The MBX D3DM and OpenGL ES drivers enable the 3-Stack board to provide the following software and hardware support:

- Drivers derive from the Graphics Primitive Engine (GPE) class
- Drivers support the DirectDraw Hardware Abstraction Layer (DDHAL), support overlay surface for pixel format RGB565, UYVY, YV12, Overlay surface color key feature
- Driver support TVout
- Drivers support the Epson L4F00242T03 panels
- Direct3D Mobile supports Microsoft Direct3D Mobile Specification
- OpenGL ES supports OpenGL ES 1.1 Specification
- MBX driver uses VFP for hardware accelerate

## 18.3 Hardware Operation

Refer to the chapter on the MBX in the *MCIMX31 and MCIMX31L Applications Processors Reference Manual* for detailed operation and programming information.

### 18.3.1 Conflicts with other Peripherals

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. For details of this architecture and its operation, see the Platform Builder Help:

**Developing a Device Driver > Windows Embedded CE Drivers> Direct3D Mobile Display Drivers**

The MBX driver uses a standalone DirectDraw driver. If MBX is included in the OS image, the IPU DirectDraw driver will be replaced by the MBX DirectDraw driver.

If VFP accelerate is used for MBX, VFP is setted to **runfast** mode. A mathematical operation may fail when it involves a NaN operation. For further information, see the *ARM VFP v2 Floating Point Support Library for Microsoft Windows Embedded CE 6.0 Release Note*.

### 18.4.1 Application / User Interface to MBX Drivers

Communications with the MBX drivers are provided 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, to present rendered output to the user.

## 18.4.2 Configuring the LCD Display Panels

The display configuration is based on the **PanelType** registry key, which is described in the **DisplayRegistry Settings** section below. The **PanelType** registry key indicates the display panel that is being used. The supported display panel is the Epson L4F00242T03 LCD panel.

### 18.4.2.1 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 Epson L4F00242T03 VGA panel is selected, the following registry keys are included:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU_SDC]
"Bpp"=dword:10 ; 16bpp
"PanelType"=dword:1 ; Epson VGA Panel
```

### 18.4.2.2 Power Management

Power management is currently supported in the D3DM and OpenGL ES drivers.

### 18.4.2.3 Direct3D Mobile and OpenGL ES Registry Settings

The following registry keys are required to properly load the MBX D3DM and OpenGL ES drivers.

```
; Disable Power Management
[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:10
"Keep"=dword:1
"BM_POOLO_PHY_BASE"=dword:86700000
; 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:10
"Keep"=dword:1
"CLCDC_MEM_BASE"=dword:87200000
"CLDC_MEM_SIZE"=dword: 500000
; 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\GDI\Drivers]
"Display"="ddi_powervr.dll"
[HKEY_LOCAL_MACHINE\Drivers\Display\PowerVR]
"IsrDll"="GIISR.DLL"
"IsrHandler"="ISRHandler"
; Screen rotation control
"DisableDynamicScreenRotation"=dword:0
"Width"=dword: 1E0
"Height"=dword: 280
"BitsPerPixel"=dword:010
[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\system\gdi\rotation]
"Angle"=dword:0
[HKEY_LOCAL_MACHINE\System\D3DM\Drivers]
"LocalHook"="pvr_d3dm.dll"

```

### 18.4.3 Float Pointing Acceleration using the ARM VFP Library

Because the i.MX31 includes a VFP module, the MBX and other applications or drivers can use VFP to accelerate the mathematical algorithm. You can download the ARM VFP library release for Windows Embedded CE 6.0 from the ARM website and use the information in the release notes to enable the OEM floating point library support.

According to the ARM release document, this VFP library is implemented only in the **runfast** mode. However, the actual arm sample code does not set the VFP to **runfast** mode. You may refer to the following code and then add it in the ARM library init function to enable the **runfast** mode.

```

; in float_assem.s file
EXPORT  |_setFPSCR|
EXPORT  |_getFPSCR|
AREA   MX31_VFP, CODE, READONLY

|_setFPSCR| PROC
FMXRFPSCR, r0
MOVPC, LR; Return
ENDP

|_getFPSCR|
FMXRr0, FPSCR
MOVPC, LR; Return

```

```
ENDP
```

```
END
```

```
; in VFP library init function
_setFPSCR(_getFPSCR() | 0x03000000); //use runfast mode
```

## 18.5 Unit Test

To add all MBX-related drivers on Windows Embedded CE 6.0 to the image, including D3DM, OpenGL ES, DirectDraw, LCD, and IPU SDC, add the following three MBX modules from the catalog:

- MBX MX31 Base driver
- MBX MX31 D3DM Core
- MBX MX31 Ogles Core

To add all MBX-related drivers on Windows Mobile/SmartPhone to the image, uncomment the MBX macros in `MX31.bat`, and ensure that the macros related to the LCD and IPU SDC are available in `MX31.bat`.

The sections below focus on the D3DM Windows CETK test and the D3DM/OpenGL ES demo test. For further information about the DirectDraw/GDI CETK and Windows Media Player tests, see [Chapter 10](#), “Display Driver”.

### 18.5.1 Unit Test Hardware

The Epson L4F00242T03 VGA Panel is needed to run the unit tests. The panel displays the graphics data.

### 18.5.2 Unit Test Software

#### 18.5.2.1 Direct3D Mobile Interface Tests

The following table lists the required software to run the D3DM Interface tests.

**Table 18-2. Direct3D Mobile Interface Test Software Requirements**

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
D3DM_Interface.dll	Library containing the test cases

To run the Direct3D Mobile Interface Test, follow these steps:

1. In your OS design, set the `SYSGEN_D3DM` variable from:  
**Core OS->CEBASE->Graphics and Multimedia Technologies->Graphics->Direct3D Mobile**
2. Include a Direct3D Mobile driver in your OS design.

### 18.5.3 Building the Direct3D Mobile Tests

The D3DM Interface Tests, D3DM Driver Verification Tests and D3DM Driver Comparison Tests come with the CETK. No steps are required to build these tests. The tests are in the following location:

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

### 18.5.4 Running the Direct3D Mobile Tests

#### 18.5.4.1 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, see the Platform Builder Help:

**Windows Embedded CE Test Kit -> CETK Tests and Test Tools -> Display Tests -> Direct3D Mobile Interface Test**

The following table describes the test cases in the D3DM Interface test suite.

**Table 18-3. Direct3D Mobile Interface Test Cases**

Test Case	Description
1-99	Tests the methods for the IDirect3DMobile interface.
101-199	Tests the methods for the IDirect3DMobileDevice interface.
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

### 18.5.5 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.

### 18.5.6 Direct3D Mobile Application Samples

There are seven D3DM application samples located in \WINCE600\PUBLIC\DIRECTX\SDK\SAMPLES\D3DM, which will be built when BSP\_MBX is enabled. The table that follows identifies these tests.

**Table 18-4. Direct3D Mobile Application Samples**

Tests	Pass
D3dm_createdevice.exe	Y
D3dm_fixedpoint.exe	Y
D3dm_lights.exe	Y
d3dm_matrices.exe	Y
d3dm_textures.exe	Y
d3dm_twotri.exe	Y
d3dm_vertices.exe	Y

### 18.5.6.1 Direct3D Mobile/OpenGL ES Demos

Tests	Pass
D3DMEvilSkull.exe	Y
OGLESVase.exe	Y

See the MX31 MBX SDK package for additional demo applications in source code and binary code.

### 18.5.7 Known Issues for MBX CE6 Driver

The following are known issues for the MBX CE6 driver:

- For directdraw cetk, in TVout mode, 9 cases (102/200/210/220/300/310/320/1200/1300) failed.
- After suspending the system, you need to click the panel to resume it.
- While running overlay application, like mosquito or media play, if you rotate or switch between LCD and TV, the application fails to run. To re-run, exit and reopen the application.
- D3DM comparison test 5192 case failed.
- MBX driver only supports suspend/resume from keypad when in tvout mode.

## 18.6 Drivers API Reference

### 18.6.1 Direct3D Mobile

For documentation for the Direct3D driver APIs, see the Platform Builder Help. No additional custom API information is required for the features currently supported in the Direct3D Mobile driver. For reference information on basic Direct3D Mobile driver functions, methods, and structures, see the Platform Builder Help:

**Developing a Device Driver- > Windows Embedded CE Drivers- > Direct3D Mobile Display Drivers- > Direct3D Mobile Driver Reference**

For reference information on Direct3D Mobile functions, callbacks, and structures, see the Platform Builder Help:

## Windows Embedded CE Features -> Graphics -> Direct3D Mobile

### 18.6.2 OpenGL ES

Documentation for the OpenGL driver APIs can be found at <http://www.khronos.org/opengles>

# Chapter 19

## NAND Flash Media Driver (FMD)

### 19.1 NAND FMD Summary

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 NAND flash controller are responsible for the actual I/O with the corresponding NAND flash devices respectively. The NAND FMD driver included in the MX31 BSP is targeted for the NAND flash device shipped with the MX31 PDK, But these drivers can be easily ported to other NAND flash devices.

The NAND flash device can be sorted as two categories: a small page size (page size is 512 bytes) NAND device and a large page size (page size is 2048 bytes) NAND device. For the MX31 PDK, the large page NAND device K9F2G08R0A is supported.

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

**Table 19-1. NAND Flash Media Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 SOC Driver Path	N/A
SOC Driver Path	N/A
SOC Static Library	N/A
Platform Driver Path	\\WINCE600\\PLATFORM\\<TGTPLAT>\\SRC\\COMMON\\NANDFMD \\WINCE600\\PLATFORM\\<TGTPLAT>\\SRC\\DRIVERS\\BLOCK\\NANDFMD
Import Library	fal.lib, fmdhooklib.lib
Driver DLL	nandfmd.dll
Catalog Item	Third Party → BSP → Freescale i.MX31 3DS: ARMV4I → Storage Drivers → MSFlash Drivers → Samsung K9F2G08R0A NAND Flash.
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NAND_FMD=1 & BSP_NAND_K9F2G08R0A=1

### 19.2 Requirements

The NAND FMD should meet the following requirements:

- Support the Windows CE FMD interface.
- Support both large page and small page NAND.
- Support EMI clock gating for power management.

## 19.2.1 Conflicts with other SoC peripherals

### 19.2.1.1 MX31 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 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.

## 19.3 Software Operation

The development concepts for flash media drivers are described in the Windows CE 6.0 Help Documentation section under the topic **Developing a Device Driver > Windows Embedded CE Drivers > Flash Drivers**. The NAND FMD supported in the MX31 PDK BSPs implements the required FMD functions for interfacing to NAND flash devices.

### 19.3.1 Compile-Time Configuration Options

The NAND FMD driver abstracts the details of the NAND flash memory device to a single header file. This header file is found in the `\WINCE600\PLATFORM\<TGTPLAT>\SRC\COMMON\NANDFMD` directory and named according to the NAND device.

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 `\WINCE600\PLATFORM\<TGTPLAT>\SRC\COMMON\NANDFMD\nandfmd.h` and recompile the NAND FMD driver for the new device.

### 19.3.2 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 Windows CE 6.0 Help Documentation section under the topic **Windows Embedded CE Features > File Systems and Data Store > File Systems and Data Store Registry Settings**.

### 19.3.3 DMA Support

The NAND FMD currently does not provide DMA support.

### 19.3.4 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.

## 19.4 Unit Test

The NAND FMD was testing using Windows CE 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.

### 19.4.1 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.

**Table 19-2. CETK Tests and Command Lines**

CETK Test	Command Line
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 -store FlashDisk -z"
Storage Device > Flash Memory Read/Write and Performance Test	For WinCE CETK: tux -o -d flashwear -c "-disk DSK1: -z"  For Mobile CETK: tux -o -d flashwear -c "-z /profile FlashDisk" Note: For 2K sector size format, don't run the test because the CETK doesn't supports 2K sector size.
Storage Device >Storage Device Block Driver Performance Test	tux -o -d disktest_perf -c "-disk DSK1: -z"
Storage Device > File System Driver Test	tux -o -d fsdttst -c "-p FlashDisk -z"

#### NOTE:

- >Read/Write test can recognize the parameter /Profile with QFE04. CETK cases #2001, #2002 failed can be safely ignored.
- > API test: CETK cases #4006, #4007, #4012, #4013, #4022, #4023 can be safely skipped.
- > Flash Memory Read/Write and Performance Test do support the 2K sector size with QFE04 and can recognize parameter /Profile

> File System Driver test: CETK cases #5019,#5022 can be safely skipped.

## 19.4.2 System Testing

The following system tests were performed to verify the operation of the NAND FMD:

Use 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

# 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.

### 20.1 Postfilter Driver Summary

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

**Table 20-1. Postfilter Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\FREESCALE\MXARM11_FSL_V1\I PU\PF
CSP Static Library	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

### 20.2 Requirements

The Postfilter driver should meet the following requirements:

- Support 3 postfiltering modes: H.264 deblocking, MPEG4 deblocking, and MPEG4 deblocking and deringing.
- 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.
- Function as a stream interface driver implementing the programming interface defined in this document.
- Support two power management modes, full on and full off.

## 20.3 Hardware Operation

Refer to the chapter on IPU in the hardware specification document for detailed operation and programming information.

### 20.3.1 Conflicts with other SoC peripherals

#### 20.3.1.1 Peripheral Conflicts

There are no peripheral conflicts on this SoC.

## 20.4 Software Operation

### 20.4.1 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.

### 20.4.2 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, see the **PFOpenHandle** section under the PF API reference.

### 20.4.3 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**.

Following are important pieces of information needed to configure a Postfilter operation:

- The postfiltering mode (for example, H.264 deblocking or MPEG4 deblocking).
- The input frame parameters, including width, height, and stride.
- 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. **PFAllocPhysMem()** API is provided 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);
```

## 20.4.4 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 V 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 signaled, representing the completion of 3 different phases of the Postfilter task: the completion of the Y component, the completion of the Cr component, and the completion of the Cb component, which corresponds to the End-Of-Frame (EOF) of the Postfilter task. These events are signaled 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 signaled 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 signaled.

```
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);
```

## 20.4.5 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.

## 20.4.6 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
```

## 20.4.7 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.

### 20.4.7.1 PowerUp

This function is not implemented for the Postfilter driver.

### 20.4.7.2 PowerDown

This function is not implemented for the Postfilter driver.

### 20.4.7.3 IOCTL\_POWER\_SET

This function is not implemented for the Postfilter driver.

## 20.5 Unit Test

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.

### 20.5.1 Unit Test Software

The following table lists the required software to run the Postfilter driver tests.

**Table 20-2. Unit Test Software Requirements**

Requirements	Description
pftest.exe	Postfilter test execution file.

### 20.5.2 Building the Postfilter Tests

#### 20.5.2.1 Unit Test in Windows CE

To build the Postfilter unit test, 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”).
- Enter “*set WINCEREL=I*” 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” to the \$(\_FLATRELEASEDIR) directory.

### 20.5.3 Running the Postfilter Tests

After downloading an OS image to the board, the unit test can be executed from the “Target Control” shell with the following command:

```
“s |release\pftest.exe”
```

## 20.6 Postfilter Driver API Reference

### 20.6.1 Postfilter Driver Functions

#### 20.6.1.1 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**.

#### 20.6.1.2 PFCloseHandle

This API function closes a handle to the PF driver:

```
BOOL PFCloseHandle(
    HANDLE hPF
);
```

##### Parameters

*hPF*

[in] Handle to the PF driver returned by **PFOpenHandle** API.

##### Return Values

TRUE indicates success.

FALSE indicates failure.

To get extended error information, call `GetLastError`.

An open handle to the specified file indicates success.

##### Remarks

None.

#### 20.6.1.3 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.

## 20.6.1.4 PFStart

This API function starts a Postfilter operation.

```

void PFStart(
    HANDLE hPF,
    pPfStartParams pStartParams
);

```

### Parameters

*hPF*

[in] Handle to the PF driver returned by **PFOpenHandle** API.

*pStartParams*

[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 signaled 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.

## 20.6.1.5 PFStart2

This API function starts a Postfilter operation.

```

void PFStart2(
    HANDLE hPP,
    pPfStartParams pStartParams,

```

```

        unsigned int VirtualFlag,
        unsigned int yOffset
    );

```

## Parameters

*hPF*

[in] Handle to the PF driver returned by **PFOpenHandle** API.

*pStartParams*

[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.

*VirtualFlag*

[in] Flag to indicate if virtual memory pointer.

*yOffset*

[in] offset to the pointer, in byte.

## Return Values

None.

## Remarks

**PFStart2** extends **PFStart** by passing two more parameters.

### 20.6.1.6 PFSetAttributeEx

This API modifies the attributes of virtual memory in upper layer applicas's context:

```

BOOL PFSetAttributeEx(
    HANDLE hPF,
    pPfSetAttributeExData pData
);

```

## Parameters

*hPF*

[in] Handle to the PF driver returned by **PFOpenHandle** API.

*pData*

[in] An object of the *pPfSetAttributeExData* structure.

## Return Values

TRUE indicates success.

FALSE indicates failure.

## Remarks

This function gives the upper layer applications a chance to change their virtual memory's attributes.

### 20.6.1.7 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.

### 20.6.1.8 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**

None.

**20.6.1.9 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**

None.

**20.6.2 PF Driver Enumerations****20.6.2.1 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.

## **20.6.3 PF Driver Structures**

### **20.6.3.1 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.

### **20.6.3.2 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 Postfilter 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.

### 20.6.3.3 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.

### 20.6.3.4 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;
    void* qp_buf;
    void* bs_buf;
    int qp_size;
} 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.

##### qp\_buf

Pointer to current qb buffer

##### bs\_buf

Pointer to current bs buffer

##### qp\_size

qb and bs buffer size

### 20.6.3.5 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.

## **20.6.3.6 pfSetAttributeExData**

This structure is used in calls to **PFSetAttributeEx**.

```
typedef struct pfSetAttributeExDataStruct {  
    LPVOID lpvAddress;  
    DWORD cbSize;  
} pfSetAttributeExData, *pPfSetAttributeExData;
```

### **Members**

#### **lpvAddress**

The starting address of virtual memory.

#### **cbSize**

The size of virtual memory.



# Chapter 21

## Power Management IC (PMIC)

This chapter provides the information that you need to:

- Develop device drivers that interface directly with 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 (for example, audio I/O and USB on-the-go connectivity).

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.

### 21.1 PMIC Driver Summary

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

**Table 21-1. PMIC Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
CSP Driver Path	..\PLATFORM\common\src\soc\freescale\pmic\mc13783_fsl_v1
CSP Static Library	pmicPdk_mc13783_fsl_v1.lib pmicSdk_mc13783_fsl_v1.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\PMIC\MC13783\PDK ..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\PMIC\MC13783\SDK
Import Library	N/A
Driver DLL	pmicPdk_MC13783.dll pmicSdk_MC13783.dll
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_PMIC_MC13783=1

### 21.2 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 Embedded CE Drivers → Stream Interface Drivers**.

The PMIC Stream Interface driver controls the PMIC hardware directly through 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
- Battery
- Regulators
- Keys (Power, PTT)
- ADC /Touch
- End of Life comparator
- Power Fail
- Battery Charger
- GPIO
- CE Bus

### 21.2.1 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 (for example, selecting the USB connectivity operating mode).

A device driver and API framework for Windows Embedded CE 6.0 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.

## 21.3 Hardware Operation

Refer to the MC13783 datasheet for details on the MC13783 PMIC.

### 21.3.1 MX31 Peripheral Conflicts

There are no MX31 pin conflicts on the 3-Stack. The CSPI2 is used to communicate with the MC13783 PMIC on the MX31 platform; the CSPI2 signals are selected in the IOMUX.

## 21.4 Software Operation

### 21.4.1 Configuring the PMIC

The PMIC modules can be used by applications or device drivers. For example, battery APIs of the 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.

### 21.4.2 Creating a Handle to the PMIC

Before calling any of the PMIC APIs, make sure that the PMIC device is attached by calling the **CreateFile** function which opens a file and returns a handle that can be used to access the MC13783 hardware. If the MC13783 hardware does not exist, **CreateFile** returns `ERROR_FILE_NOT_FOUND`.

To open a handle to the PMIC, complete the following steps:

1. Insert a colon after the PMI1 port for the first parameter, *lpFileName*. For example, specify PMI1: as the PMIC port.
2. Specify `FILE_SHARE_READ | FILE_SHARE_WRITE` in the *dwShareMode* parameter. Multiple handles to a PMIC 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 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 steps specified above are performed when PMIC device is attached.  
If hPMI handle is null, then perform the above steps.

### 21.4.3 Write Operations

The PMIC driver does not provide an interface to write through the `PMIC_Write` (stream write) function in the PMIC driver, and `PMIC_Write` is a stub function and always returns success.

### 21.4.4 Read Operations

Like the write operation, the PMIC driver does not provide for reading through the `PMIC_Read` function in the PMIC driver; this is a stub function and always returns success.

### 21.4.5 Closing the Handle to the PMIC

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

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

### 21.4.6 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. The PMIC module clock is enabled whenever any of the PMIC registers needs to be accessed and then disabled once it is finished.

#### 21.4.6.1 PowerUp

This function is not implemented for the PMIC driver.

#### 21.4.6.2 PowerDown

This function is not implemented for the PMIC driver.

#### 21.4.6.3 IOCTL\_POWER\_CAPABILITIES

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

#### 21.4.6.4 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** values in the PMIC driver. Any request that is not D0 is changed to a D4 request and will result in the system entering into suspend state, while for a value of D0 the system will again be resumed.

### 21.4.6.5 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.

### 21.4.7 PMIC Registry Settings

There are no registry settings that need to be modified to use the PMIC APIs.

### 21.4.8 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 events. Refer to [Section 21.6.2, “Interrupt Handling”](#) for a description of interrupt handling.

#### 21.4.8.1 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;

```

## 21.4.8.2 Functions

### 21.4.8.2.1 PmicADCGetSingleChannelOneSample

This function gets one channel and 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

### 21.4.8.2.2 PmicADCGetSingleChannelEightSamples

This function gets one channel and 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

### 21.4.8.2.3 PmicADCGetMultipleChannelsSamples

This function gets a 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

#### 21.4.8.2.4 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.

#### 21.4.8.2.5 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

#### 21.4.8.2.6 PmicADCSetComparatorThresholds

This function sets WHIGH and WLOW for automatic ADC result comparators.

**Prototype** `PMIC_STATUS PmicADCSetComparatorThresholds(UINT16 which, UINT16 wlow);`

**Parameters:** `which [in]`  
a high comparator threshold.  
`wlow [in]`  
a low comparator threshold.

**Returns:** PMIC\_STATUS

#### 21.4.8.2.7 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.

#### 21.4.8.2.8 PmicADCInit

This function initializes PMIC ADC's resources.

**Prototype** *PMIC\_STATUS PmicADCInit(void);*

**Parameters:** None.

**Returns:** PMIC\_STATUS

#### 21.4.8.2.9 PmicADCDeinit

This function deinitializes PMIC ADC's resources.

**Prototype** *void PmicADCDeinit(void);*

**Parameters:** None

**Returns:** PMIC\_STATUS

### 21.4.8.3 Power Management

There is no additional power management implementation done specifically for Atlas ADC other than the implementation described in the Power Management section of this document.

## 21.5 Unit Test

The PMIC CETK test cases verify the functionality of the various PMIC components.

### 21.5.1 Unit Test Hardware

The MX31 3-Stack board is required.

### 21.5.2 Unit Test Software

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

**Table 21-2. Unit Test Software Requirements**



Requirements	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, required for logging test data
PMICtest.dll	Test .dll file

### 21.5.3 Building the PMIC Tests

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

Build an OS image for the desired configuration.

1. Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
2. Change to the PMIC Tests directory. (\WINCE600\SUPPORT\MX31\TESTS\PMIC)
3. Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
4. 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.

### 21.5.4 Running the PMIC Tests

For testing PMIC, it is required to run the tux test suite in Kernel mode. In order to achieve this, copy the 'ktux.dll' file from Program Files\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4i folder into the release directory, and then run the test suite using the following command

**s tux -o -n -d pmictest.dll**

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

**Table 21-3. PMIC Test Cases**

#	Test name	Description
1	PMIC Register Access	This test does read/write verification of IMR register on the PMIC.
2	PMIC Battery	This test verifies the Battery Interface and control.
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 the handset current.
10	AdcTouchRead	This test gets 3 (x,y) coordinates from the touch screen.

## 21.6 PMIC Reference API

### 21.6.1 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 IOCTLs are used within the APIs developed for specific modules of the PMIC device. Most of the IOCTLs will be explained in the specific sections wherever they are more relevant.

#### 21.6.1.1 PMIC\_IOCTL\_LLA\_READ\_REG

This `DeviceIoControl` request reads the register content.

<b>Parameters</b>	<i>hPMI</i> [in] Handle to the device that is to perform the operation. To obtain a device handle, call the <code>CreateFile</code> function. <i>lpInBuffer</i> index of the register. <i>lpOutBuffer</i> [out] Long pointer to a buffer that receives the output data for the operation. Set to <code>NULL</code> if the <code>dwIoControlCode</code> parameter specifies an operation that does not produce output data.
-------------------	---

#### 21.6.1.2 PMIC\_IOCTL\_LLA\_WRITE\_REG

This `DeviceIoControl` request writes the data to the said register of the PMIC device.

<b>Parameters</b>	<i>hPMI</i> [in] Handle to the device that is to perform the operation. To obtain a device handle, call the <code>CreateFile</code> function. <i>lpInBuffer</i> index of the register. <i>lpOutBuffer</i> pointer to data which needs to be written to the said register.
-------------------	--

#### 21.6.1.3 PMIC\_IOCTL\_LLA\_INT\_REGISTER

This `DeviceIoControl` is used to register interrupt.

<b>Parameters</b>	<i>hPMI</i> [in] Handle to the device that is to perform the operation. To obtain a device handle, call the <code>CreateFile</code> function. <i>lpInBuffer</i>
-------------------	---

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);
```

#### 21.6.1.4 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);
```

#### 21.6.1.5 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);
```

#### 21.6.1.6 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);
```

**21.6.1.7 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);
```

**21.6.2 Interrupt Handling****21.6.2.1 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 through 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.

### 21.6.2.2 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.

#### 21.6.2.2.1 PMIC Interrupt Events

The table below shows the events and corresponding MC13783 interrupts.

**Table 21-4. 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 21-4. 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

### 21.6.2.3 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;

```

## 21.6.2.4 Functions

### 21.6.2.4.1 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;
}

```

### 21.6.2.4.2 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;
}
```

### 21.6.2.4.3 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. The code example is below:

```
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;
}
```



#### 21.6.2.4.4 PmicInterruptDisable

The 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. The code example is below.

```
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;
}
```

#### 21.6.2.4.5 PmicInterruptEnable

The PmicInterruptEnable function re-enables 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. The code example is below.

```
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);
```

## 21.6.3 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/applications from accessing some registers. Interrupt registers is one example. The interrupt library functions will be in this Low Level Access DLL.

### 21.6.3.1 Functions

#### 21.6.3.1.1 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

#### 21.6.3.1.2 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.

```
TESTPROC API 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 & 0xFFFFFFFF));
    g_pKato->Log(LOG_COMMENT, TEXT("Now, try to change IMR to 0xFF\r\n"));
    PmicRegisterWrite(1, 0xFF, 0xFFFFFFFF);
    PmicRegisterRead(1, &reg);
    g_pKato->Log(LOG_COMMENT, TEXT("Register IMR is 0X%X\r\n"), (reg & 0xFFFFFFFF));
    Enter ISR loop
    if ((reg&0xFFFFFFFF) == 0xFF)
    {
        GPT_TEST_FUNCTION_EXIT();
        return TPR_PASS;
    }
    else
    {
        GPT_TEST_FUNCTION_EXIT();
        return TPR_FAIL;
    }
}

```

## 21.6.4 Power Control Reference

### 21.6.4.1 PwCtrl API

This section provides information about the API provided by PwCtrl API DLL.

Using the following APIs, the MC13783 power control module can be accessed.

**Table 21-5. PwCtrl API Modules**

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	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 to enable the REGEN pin of selected voltage regulator
PmicPwrctrlDisableRegenAssig	Used to disable the REGEN pin of selected voltage regulator
PmicPwrctrlGetRegenAssig	Used to read the REGEN pin value for said voltage regulator

## 21.6.4.2 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   */
};

```

#### 21.6.4.2.1 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**

#### 21.6.4.2.2 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.

#### 21.6.4.2.3 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.

#### 21.6.4.2.4 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

#### 21.6.4.2.5 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

#### 21.6.4.2.6 PmicPwrctrlGetPowerCutCounter

**Prototype** *PMIC\_STATUS PmicPwrctrlGetPowerCutCounter (UINT8\* counter);*  
 This function is used to get the power cut counter.

**Parameters:** *counter [out]*  
 This function is used to get the counter number

**Returns:** *status*  
 PMIC\_SUCCESS for success and PMIC\_ERROR for failure



#### 21.6.4.2.7 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

#### 21.6.4.2.8 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

#### 21.6.4.2.9 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

#### 21.6.4.2.10 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

#### 21.6.4.2.11 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

#### 21.6.4.2.12 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

#### 21.6.4.2.13 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

#### 21.6.4.2.14 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

#### 21.6.4.2.15 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

#### 21.6.4.3 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

#### 21.6.4.4 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

#### 21.6.4.5 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

#### 21.6.4.6 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

#### 21.6.4.7 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

#### 21.6.4.8 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

#### 21.6.4.9 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

#### 21.6.4.10 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 as 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 through 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

#### 21.6.4.11 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

#### 21.6.4.12 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

### 21.6.5 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 until 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;
}
```

## 21.6.6 Memory Hold Operation functions

The Memory Hold circuit provides power to the memory during a power cut through VBKUP1. To avoid leakage from the VBKUP1 into circuitry connected to BP during a power cut, an external PMOS should 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;
}

```

## 21.6.7 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.

The 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)

```

The 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;
}

```

## 21.6.8 Power Management

There is no additional power management implementation done specifically for Atlas Power Control other than the implementation described in the Power Management section of this document.

## 21.6.9 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 through 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 VRFCF 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; and VMMC1 and VMCC2 for dual multimedia card support or peripheral supply such as Bluetooth PA.

## 21.6.10 Data Structures

```

// switch mode regulator
typedef enum _MC13783_REGULATOR_SREG{
    SW1A = 0,
    SW1B,
    SW2A,
    SW2B,
    SW3,
} MC13783_REGULATOR_SREG;

```

## Power Management IC (PMIC)

```
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,

```

## Power Management IC (PMIC)

```
    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 vrfcpc;
    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;

```

## 21.6.11 Switch mode regulator API's

### 21.6.11.1 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 failure This function is used to turn on the switch mode regulator.

### 21.6.11.2 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

### 21.6.11.3 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

#### 21.6.11.4 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

#### 21.6.11.5 PmicSwitchModeRegulatorSetMode

##### Prototype

```
PMIC_STATUS PmicSwitchModeRegulatorSetMode ( PMIC_REGULATOR_SREG
regulator, PMIC_REGULATOR_SREG_STBY standby, 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

#### 21.6.11.6 PmicSwitchModeRegulatorGetMode

##### Prototype

```
PMIC_STATUS PmicSwitchModeRegulatorGetMode (PMIC_REGULATOR_SREG
regulator, PMIC_REGULATOR_SREG_MODE* mode);
```

This function gets 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

### 21.6.11.7 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

### 21.6.11.8 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

### 21.6.11.9 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.



### 21.6.11.10 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.

### 21.6.11.11 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.

### 21.6.11.12 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.

### 21.6.11.13 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.

## 21.6.12 Linear Voltage Regulator API's

### 21.6.12.1 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.

### 21.6.12.2 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.

### 21.6.12.3 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

### 21.6.12.4 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

### 21.6.12.5 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

### 21.6.12.6 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.

## 21.6.13 Power Management

There is no additional power management implementation done specifically for Atlas Voltage Regulator other than the implementation described in the Power Management section of this document.

## 21.6.14 Battery Charger

## 21.6.15 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;
```

## 21.6.16 Battery Charger API (Compatible with SC55112 API)

### 21.6.16.1 PmicBatterEnableCharger

This function is used to start charging a battery. For different chargers, different voltage and current ranges are supported.

The main battery charger supports a settable voltage and current. The coincell supports only a settable voltage. The trickle 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.

### 21.6.16.2 PmicBatterDisableCharger

This function turns off the selected charger. This is done by setting the current level to zero for the main and trickle 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.

### 21.6.16.3 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.

### 21.6.16.4 PmicBatterGetChargerSetting

This function is used to retrieve what the charger settings 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.

### 21.6.16.5 PmicBatterGetChargeCurrent

This function 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.

### 21.6.16.6 PmicBatterEnableEol

This function enables End-of-Life comparator.

**Prototype** `PMIC_STATUS PmicBatterEnableEol(void);`

**Parameters:** None

**Returns:** This function returns PMIC\_SUCCESS if successful.

### 21.6.16.7 PmicBatterDisableEol

This function disables End-of-Life comparator.

**Prototype** `PMIC_STATUS PmicBatterDisableEol (void);`

**Parameters:** None

**Returns:** This function returns PMIC\_SUCCESS if successful.

### 21.6.16.8 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.

### 21.6.16.9 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.

### 21.6.16.10 PmicBatterSetUnregulated

This function sets limited charging mode on the 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.

## 21.6.17 Battery Charger API (MC13783 Native For Compatibility with SC55112)

These functions are available 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.

### 21.6.17.1 PmicBatteryEnableAdChannel5

This function enables use of AD channel 5 to read the charge current on the SC55112 PMIC.

**Prototype** `PMIC_STATUS PmicBatteryEnableAdChannel5();`

**Parameters:** None.

**Returns:** PMIC\_SUCCESS

### 21.6.17.2 PmicBatteryDisableAdChannel5

This function disables use of AD channel 5 to read the charge current on the SC55112 PMIC.

**Prototype** `PMIC_STATUS PmicBatteryDisableAdChannel5();`

**Parameters:** None.

**Returns:** PMIC\_SUCCESS;

### 21.6.17.3 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;

### 21.6.17.4 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;

### 21.6.17.5 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;

### 21.6.17.6 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;

## 21.6.18 Battery Charger API (MC13783 Native)

### 21.6.18.1 PmicBatterySetChargeVoltage

This function programs the output voltage of the 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



### 21.6.18.2 PmicBatteryGetChargeVoltage

This function returns the output voltage of the charge regulator.

**Prototype** `PMIC_STATUS PmicBatteryGetChargeVoltage(UINT8* chargevoltagelevel);`

**Parameters:** `chargevoltagelevel [OUT]`  
pointer to voltage level

**Returns:** PMIC\_STATUS

### 21.6.18.3 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

### 21.6.18.4 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

### 21.6.18.5 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

### 21.6.18.6 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

### 21.6.18.7 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

### 21.6.18.8 PmicBatteryReverseDisable

This function disables the reverse mode.

**Prototype** `PMIC_STATUS PmicBatteryReverseDisable();`**Parameters:** None**Returns:** PMIC\_STATUS

### 21.6.18.9 PmicBatteryReverseEnable

This function enables the reverse mode.

**Prototype** `PMIC_STATUS PmicBatteryReverseEnable();`**Parameters:** None**Returns:** PMIC\_STATUS

### 21.6.18.10 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`

### 21.6.18.11 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`

### 21.6.18.12 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`

### 21.6.18.13 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`

### 21.6.18.14 PmicBatteryChargeLedDisable

This function disables the charging LED.

**Prototype** `PMIC_STATUS PmicBatteryChargeLedDisable();`

**Parameters:** None

**Returns:** `PMIC_STATUS`

### 21.6.18.15 PmicBatteryChargeLedEnable

This function enables the charging LED.

**Prototype** `PMIC_STATUS PmicBatteryChargeLedEnable();`

**Parameters:** None

**Returns:** PMIC\_STATUS

### 21.6.18.16 PmicBatteryEnablePulldown

This function enables the 5k pull-down resistor used in the dual path charging.

**Prototype** `PMIC_STATUS PmicBatteryEnablePulldown();`

**Parameters:** None.

**Returns:** PMIC\_STATUS.

### 21.6.18.17 PmicBatteryDisablePulldown

This function disables the 5k pull-down resistor used in the dual path charging.

**Prototype** `PMIC_STATUS PmicBatteryDisablePulldown();`

**Parameters:** None.

**Returns:** PMIC\_STATUS.

### 21.6.18.18 PmicBatteryEnableCoincellCharger

This function enables the coincell charger.

**Prototype** `PMIC_STATUS PmicBatteryEnableCoincellCharger();`

**Parameters:** None

**Returns:** PMIC\_STATUS

### 21.6.18.19 PmicBatteryDisableCoincellCharger

This function disables the coincell charger.

**Prototype** `PMIC_STATUS PmicBatteryDisableCoincellCharger();`

**Parameters:** None

**Returns:** PMIC\_STATUS

### 21.6.18.20 PmicBatterySetCoincellVoltage

This function programs the output voltage level of the 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

### 21.6.18.21 PmicBatteryGetCoincellVoltage

This function returns the output voltage level of the coincell charger.

**Prototype** `PMIC_STATUS PmicBatteryGetCoincellVoltage (UINT8* coincellvoltagelevel);`  
**Parameters:** `voltagelevel [OUT]`  
 pointer to voltage level  
**Returns:** PMIC\_STATUS

### 21.6.18.22 PmicBatteryEnableEolComparator

This function enables the end-of-life function instead of the LOBAT.

**Prototype** `PMIC_STATUS PmicBatteryEnableEolComparator();`  
**Parameters:** None  
**Returns:** PMIC\_STATUS

### 21.6.18.23 PmicBatteryDisableEolComparator

This function disables the end-of-life comparator function.

**Prototype** `PMIC_STATUS PmicBatteryDisableEolComparator();`  
**Parameters:** None  
**Returns:** PMIC\_STATUS

### 21.6.18.24 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

## **21.6.19 Power Management**

There is no additional power management implementation done specifically for Atlas Battery other than the implementation described in the Power Management section of this document.

## Chapter 22

# 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.

### 22.1 Power Manager Summary

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

**Table 22-1. Power Manager Driver Attributes**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
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 →CEBase→Core OS Services →Power Management → Power Management (Full)
SYSGEN Dependency	SYSGEN_PM
BSP Environment Variables	N/A

### 22.2 Requirements

Include and test the power manager provided in the Platform Builder public directory.

### 22.3 Hardware Operation

Power Manager does not interface directly to peripheral devices.

### 22.4 3-Stack Software Operation

The Platform Builder helps documents the power manager framework and sample power manager. See **Windows Embedded CE Features > Power Management**.

For information about the system power states implemented in the sample power manager, see Platform Builder help:

**Windows Embedded CE Features > Power Management > Power States > System Power States > Example System Power States**

## 22.4.1 Power Management

Power Manager can export the stream interface and register as a generic power-manageable device driver. It can receive IO\_POWER\_XX notification and configure PMIC and wakeup source in thread context. The following table describes the power consumption goal and the actions that can be taken to achieve that goal.

## 22.4.2 Image Configuration

**Table 22-2. Power Consumption Goals**

Mode	Action
suspend	(1) keep power supply to memory, set SW2A work in low power mode  (2) low down the voltage to ARM and QPER, set SW1 work in lower power mode  (3) Set all PMIC regulators and GPO control by standby pin  (4) Close PMIC SW3 supply
audio playback	(1) keep power supply to memory  (2) Close PIMC regulators that not been used  (3) Close GPO 1,2,3  (4) Close PMIC SW3 supply  (5) replace MSFT WMA codec with FSL codec  (6) Add DVFC mode to NK image
video playback	(1) same as audio playback

To build the audio playback power consumption image, follow these steps:

1. Remove kernel debug and KITL from the NK image.
2. Remove ATA, Camera, TV-Out and USB high-speed Host 2 driver module from the workspace, because the power supply for those items has been gated in power policy enable.
3. Replace Microsoft WMA and MWV codec with Freescale codec.
4. Add the DVFC module to the image.



## 5. Perform a sysgen.

To enable optimization for the audio playback power consumption function, follow these steps:

1. Build an "Optimization audio playback power consumption image".
2. Bootup the device and enter the eboot menu.
  - Set **Power Policy** to **Enable**.
  - Save the changes and boot up the device.

### NOTE

The 3-Stack power policy only recognizes NAND audio/video playback files. Modules that were not called by the audio and video playback may not work well when power policy is enabled.

## 22.4.3 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

[HKEY_LOCAL_MACHINE\System\CurrentControlSet\Control\Power\Timeouts]
"ACUserIdle"=dword:00000000 ; in seconds
"ACSystemIdle"=dword:00000000 ; in seconds
"BattUserIdle"=dword:00000000 ; in seconds
"BattSystemIdle"=dword:00000000 ; in seconds
```

## 22.5 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.

## 22.6 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.

### 22.6.1 Application Interface

The power manager API's for applications are documented in help at:

- **Windows Embedded CE Features > Power Management > Power Manager Interfaces > Application Interface**
- **Windows Embedded CE Features > Power Management > Power Manager Interfaces> Notification Interface**
- **Windows Embedded CE Features > Power Management > Power Management Reference**

## **22.6.2 Device Driver Interface**

The interface for device drivers is documented in help at:

- **Windows Embedded CE Features > Power Management > Power Manager Interfaces > Device Driver Interface**
- **Windows Embedded CE Features > Power Management > Power Management Reference**



## Chapter 23

# Secure Digital Host Controller Driver

The Secure Digital Host Controller (SDHC) module supports Multimedia Cards (MMC), Secure Digital Cards (SD) and Secure Digital I/O and Combo Cards (SDIO). The i.MX31 device has two SDHC hardware modules. Each host controller supports connection to only one card. On the 3-Stack board, only host1 is connected to the SD card socket. The SDHC driver provides the interface between Microsoft's SD Bus driver and the SDHC hardware.

### 23.1 SDHC Driver Summary

Table 23-1 provides a summary of source code location, library dependencies and other BSP information.

**Table 23-1. SDHC Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 SOC Driver Path	..\PLATFORM\COMMON\SRC\SOC\FREESCALE\MXARM11_FSL_V1\SDHC
SOC Driver Path	N/A
SOC 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 > BSP > Freescale <TGTPLAT> > Device Drivers > SD Controller > SD Host Controller 1 Third Party > BSP > Freescale <TGTPLAT> > Device Drivers > SD Controller > SD Host Controller 2
SYSGEN Dependency	SYSGEN_SD_MEMORY=1
BSP Environment Variables	BSP_SDHC1=1 and BSP_SDHC2=1

### 23.2 Supported Functionality

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

- Supports the i.MX31 Secure Digital Host Controllers
- Supports two Host Controllers to be functional at the same time
- Supports SD cards
- Supports Power Management modes, full on and full off only

### 23.3 Hardware Operation

Refer to the chapter on the Secure Digital Host Controller (SDHC) in the hardware specification document for detailed operation and programming information.

### 23.3.1 Conflicts with Other Peripherals

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.

## 23.4 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://msdn2.microsoft.com/en-us/library/aa925967.aspx>

### 23.4.1 Required Catalog Items

#### 23.4.1.1 SD and MMC memory card support

Catalog > Device Drivers > SDIO > SD Memory.

### 23.4.2 SDHC Registry 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:FB
    "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:FB
    "Index"=dword:2
ENDIF ;BSP_SDHC2

```

### 23.4.3 DMA Support

SDHC driver supports DMA mode and non-DMA mode of data transfer. The driver defaults to DMA mode of transfer. 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 is not 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 from processor and SDMA access of the memory, the above criteria is further restricted 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

### 23.4.4 Power Management

The primary methods for limiting power in the 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 into 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 is detected and mounted.

### 23.4.4.1 PowerUp

This function is implemented to support resuming a memory card operation that was previously terminated by calling PowerDown() API. When using this function, power to the card is restored and clocks to the pertaining controller is restarted.

The 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 all the operations scheduled during a power down resume. SDIO cards is 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://msdn2.microsoft.com/en-us/library/aa910129.aspx>

Note that this function is intended to be called only by the Power Manager.

### 23.4.4.2 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.

### 23.4.4.3 IOCTL\_POWER\_CAPABILITIES

N/A

### 23.4.4.4 IOCTL\_POWER\_GET

N/A

### 23.4.4.5 IOCTL\_POWER\_SET

N/A

## 23.5 Unit Test

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

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

### 23.5.1 Unit Test Hardware

Table 23-2 lists the required hardware to run the unit tests.



**Table 23-2. Hardware Requirements**

Requirements	Description
SD Cards	SanDisk (128MB, 256MB, 512MB) Kingston (256MB) NCP (128MB, 256MB, 512MB) Transcend (128MB, 256MB, 512MB) Toshiba (1GB)
MMC Cards	Not supported on 3DS board.

## 23.5.2 Unit Test Software

Table 23-3 lists the required software to run the unit tests.

**Table 23-3. Software Requirements**

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
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
muparttest.dll	Partition Driver Test dll file
perflog.dll	Test dll for log performance data

## 23.5.3 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 in the following location:

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

## 23.5.4 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://msdn2.microsoft.com/en-us/library/aa934353.aspx>

### 23.5.4.1 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 only tests one card at a time.

### 23.5.4.2 Storage Device Block Driver API Tests

Use the command line **tux -o -d disktest -c "-z"** to run the tests. Note that this test only tests one card at a time. CETK cases #4006, #4007, #4012, #4013, and #4021 can be safely skipped

### 23.5.4.3 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 function tests all the cards inserted and requires the cards to be formatted prior to running the test. For higher capacity cards, the test takes a long time to complete. Therefore it is recommended that the system power management (from control panel) be configured so that the system does not enter suspend state during test execution. CETL case #50119 can be safely skipped

### 23.5.4.4 Partition Driver Test

Use command line **tux -o -d msparttest -c "-z"** to run the tests.

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

## 23.5.5 System Testing

The following system tests can be 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

## 23.6 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://msdn2.microsoft.com/en-us/library/aa912994.aspx>

# Chapter 24

## Serial Driver

### 24.1 Serial Driver Summary

The i.MX31 device has five internal UARTs (Universal Asynchronous Receiver Transmitters): UART1, UART2, UART3, UART4 and UART5. UART5 supports serial and 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 UART5 in addition supports slow infrared modes.

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 6.0 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 links with Microsoft provided public serial MDD library (com\_mdd2.lib) to form the complete serial port driver.

Table 24-1 provides a summary of source code location, library dependencies and other BSP information.

**Table 24-1. Serial Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 SOC Driver Path	..\PLATFORM\COMMON\SRC\SOC\freescall\mxarm11_fsl_v1\SERIAL
SOC Driver Path	..\PLATFORM\COMMON\SRC\SOC\freescall\mx31_fsl_v1\SERIAL
SOC Static Library	serial_mx31_fsl_v1.lib serial_mxarm11_fsl_v1.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\SERIAL
Import Library	com_mdd2.lib
Driver DLL	csp_serial.dll
Catalog Item for MGN	Third Party > BSPs > Freescale i.MX31 3DS: ARMV4I > Device Drivers > Serial > UART2 serial port Third Party > BSPs > Freescale i.MX31 3DS: ARMV4I > Device Drivers > Serial > UART3 serial port
SYSGEN Dependency	N/A
BSP Environment Variables for MGN	BSP_SERIAL_UART2=1 BSP_SERIAL_UART3=1

### 24.2 Supported Functionality

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

- Conforms to RS232 protocol standard
- Supports RTS/CTS hardware flow control function

- Supports up to 115200 BaudRate
- Supports internal UART controller
- Supports power management mode full on / full off

## 24.3 Hardware Operation

Refer to the chapter on the UART in the hardware specification document for detailed operation and programming information.

### 24.3.1 Conflicts with Other Peripherals

UART1 and UART2 do not have conflicts with any other module and are configured in functional mode. UART3 has conflicts with SOCI1 and SOCI3 modules and must be configured in alternate mode 1. UART4 has conflicts with ATA and USB OTG modules and must be configured in alternate mode 1. UART5 has conflicts with PCMCIA and USB modules and must be configured in alternate mode 2. [Table 24-2](#) shows pins to be configured for serial driver for different UARTs.

**Table 24-2. UART Setting for the Serial Driver**

UART Port	Pins To Be Configured				I/O MUX Settings	Comment
UART1	RXD1	TXD1	RTS1	CTS1	Functional Mode	No support
	DTR_DCE1	DSR_DCE1	RI_DCE1	DCD_DCE1		
UART2	RXD2	TXD2	RTS2	CTS2	Functional Mode	Connect to DMB
UART3	SOCI3_MOSI	SOCI3_MISO	SOCI3_SCLK	SOCI3_SPI_RDY	Functional Mode	No support
UART4	ATA_CS0	ATA_CS1	ATA_DIOR	ATA_DIOW	Alternate Mode1	No support
UART5	PC_VS2	PC_BVD1	PC_BVD2	PC_RST	Alternate Mode 2	Connect to GPS and SIR

## 24.4 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 EmbeddedDrivers > Serial Drivers > Serial Driver Development Concepts.**

### 24.4.1 Serial Registry Settings

The following registry keys are required to properly load the Serial driver.

```
IF BSP_SERIAL_UART3
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM3]
    "DeviceArrayIndex"=dword:0
    "IoBase"=dword:5000C000
    "IoLen"=dword:D4
```

```

"Prefix"="COM"
"Dll"="csp_serial.dll"
"Index"=dword:3
"Order"=dword:9
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM3\Unimodem]
"Tsp"="Unimodem.dll"
"DeviceType"=dword:0
"FriendlyName"="MGN COM3 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

IF BSP_SERIAL_UART2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2]
"DeviceArrayIndex"=dword:0
"IOBase"=dword:43F94000
"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"="MGN COM2 UNIMODEM"
"DevConfig"=hex: 10,00, 00,00, 05,00,00,00, 10,01,00,00, 00,4B,00,00, 00,00, 08, 00, 00,
00,00,00,00
ENDIF ; BSP_SERIAL_UART2

```

## 24.4.2 DMA Support

The serial driver uses the SDMA controller to transfer the data and minimize the processing that is required by the ARM core. The serial driver supports both DMA mode and Non-DMA mode of operation. DMA can be enabled/disabled using the Boolean variable present in the file `WINCE600\PLATFORM\<TGTPLAT>\SRC\INC\bsp_cfg.h`.

Individual UARTs can be configured for DMA using the variables and it is possible that some UARTs operate in one mode and the others in different mode (for example, 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. [Table 24-3](#) shows the variables used to enable/disable the DMA:

**Table 24-3. UART DMA Variables**

UART Port	Variable
UART2	BSP_SDMA_SUPPORT_UART2
UART3	BSP_SDMA_SUPPORT_UART3

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.

## 24.5 Unit Test

The serial driver is tested using the Serial Port Driver Test and Serial Communications Test included as part of the Windows CE 6.0 Test Kit (CETK). The Serial Port Test assesses whether the driver supports configurable device parameters, such as baud rate and data bits. The test also assesses additional functionality such as COM port events, escape functions and time-outs.

### 24.5.1 Unit Test Hardware

Table 24-4 lists the required hardware to run the unit tests.

**Table 24-4. Hardware Requirements**

Requirements	Description
<TGTSOC> PDK board with serial port to be tested	Serial ports can be attached as COM1 through COMX.

### 24.5.2 Unit Test Software

Table 24-5 lists the required software to run the unit tests.

**Table 24-5. Software Requirements**

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
SerDrvBvt.dll	Test .dll file for Serial Port Driver Test

### 24.5.3 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 with the other required CETK files in the following location:

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

### 24.5.4 Running the Serial Port Driver Test

To run the Serial Driver Test use the following command line:

**tux -o -d serdrvbt -c "-p COM3:".**

For detailed information on the Serial Port tests, see **Windows Embedded CE 6.0 > Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Serial Port Tests > Serial Port Driver Test** in the Windows CE 6.0 Help Documentation section.

### NOTE

Due to hardware connections with the Bluetooth chip, COM2 fails test cases 16 and 18. COM3 should pass all the serial port tests.

Serial port tests are designed to test that the serial port driver works properly and that the APIs behave correctly. It should pass all of the test cases.

Table 24-6 describes the Serial Port driver test cases.

**Table 24-6. Serial Port Driver Test Cases**

Test Case	Description
11	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.
12	Tests the <b>SetCommEvent</b> and <b>GetCommEvent</b> functions. This test fails if the driver does not properly support the <b>SetCommEvent</b> or <b>GetCommEvent</b> functions.
13	Tests the <b>EscapeCommFunction</b> function. This test fails if the driver does not support one of the Microsoft Win32 <b>EscapeCommFunction</b> functions.
14	Tests the <b>WaitCommEvent</b> 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 <b>WaitCommEvent</b> function behaves improperly or if the EV_TXEMPTY event does not signal appropriately.
15	Tests the <b>SetCommBreak</b> and <b>ClearCommBreak</b> functions. This test fails if the driver does not properly support the <b>SetCommBreak</b> or <b>ClearCommBreak</b> functions.
16	Makes the <b>WaitCommEvent</b> function return a value when the handle for the current COM port is cleared. This test fails if the <b>WaitCommEvent</b> function behaves improperly.
17	Makes the <b>WaitCommEvent</b> function return a value when the handle for the current COM port is closed. This test fails if the <b>WaitCommEvent</b> function behaves improperly.
18	Tests the <b>SetCommTimeouts</b> function and verifies that the <b>ReadFile</b> function properly times out when no data is received. This test fails if the COM timeouts do not function correctly.
19	Verifies that previous Device Control Block (DCB) settings are preserved when the <b>SetCommState</b> 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.
20	Tests <b>Open/Close</b> on port share. Calls the createfile for the COMX: port with sharedmode set to FILE_SHARE_READ and FILE_SHARE_WRITE.
21	Tests the power management abilities of a serial port. Verifies if the power management IOCTLs and function calls are supported.

## 24.6 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 Embedded Drivers > Serial Port Drivers > Serial Port Driver Reference

### 24.6.1 Serial PDD Functions

Table 24-7 shows a mapping of Serial PDD functions to the functions used in the Serial driver.

**Table 24-7. PDD and Serial Driver Function Mapping**

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



PDD Function Pointer	Serial Driver Function
HWSetsDCB	SL_SetDCB
HWSetsCommTimeouts	SL_SetCommTimeouts

## 24.6.2 Serial Driver Macros

N/A

## 24.6.3 Serial Driver Structures

### 24.6.3.1 UART\_INFO

This structure contains information about the UART Module.

```
typedef struct {
    volatile PSOC_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;
```

Table 24-8 shows the members of the UART module.

**Table 24-8. UART Module Members**

Member	Description
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
UIDiscard	This is used to discard the echo characters in IrDa Mode
UseIrDA	This Boolean value determines the driver is in IR mode or not
HwAddr	This value contains the hardware address of the UART Module
EventCallback	This is a callback to the Model Device Driver
pMDDContext	This contains the context of the UART, which is the first parameter to the callback function
dcb	This value contains the copy of Device Control Block
CommTimeouts	This contains the copy of CommTimeouts structure used to get and set the 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)

### 24.6.3.2 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;
    PSOC_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;

```

Table 24-9 shows the members of the Serial module.

**Table 24-9. Serial Module Members**

Member	Description
uart_info	This structure contains information about UART
fIRMode	This Boolean value determines the module is FIR or serial
dwDevIndex	This static value contains the device index value which is read from registry
dwIOBase	This static value contains the IO Base address of UART module which is read from registry
dwIOLen	This static value contains the IO length of UART Module which is 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

## Serial Driver

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 25

# Touch Panel Driver

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

### 25.1 Touch Panel Driver Summary

Table 25-1 provides a summary of source code location, library dependencies and other BSP information.

**Table 25-1. Touch Panel Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
MXARM11 CSP Driver Path	..\PLATFORM\common\src\soc\freescale\common_fsl_v1\touch
CSP Driver Path	N/A
CSP Static Library	touch_common_fsl_v1.lib
Platform Driver Path	..\PLATFORM\<TGTPLAT>\SRC\DRIVERS\TOUCH
Import Library	N/A
Driver DLL	touch.dll
Catalog Item	Third Party > BSP > Freescale i.MX31 3DS:ARMV4I > Device Drivers > TOUCH > MC13783 Touch Driver
SYSGEN Dependency	SYSGEN_Touch = 1
BSP Environment Variables	BSP_PMIC_MC13783=1,BSP_TOUCH_MC13783=1

### 25.2 Supported Functionality

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

- Conforms to the standards described in the Platform Builder documentation: **Developing a Device Driver > Windows Embedded CE Drivers > Touch Screen Drivers**

### 25.3 Hardware Operations

The hardware consists of a LCD Panel. Proper functioning requires an ADC module, which is used to generate the touch samples. After calculations are performed, the touch samples are converted to the x,y coordinates. The ADC module and the Touch Interrupt are part of the PMIC. For additional information, see [Chapter 21, “Power Management IC \(PMIC\)”](#).

### 25.3.1 Conflicts with Peripherals

The conflicts occur only with the GPIO Pin with which the PMIC Interrupt is routed. Therefore, those pins cannot be used as standard GPIO. For PMIC Interrupt routing and conflicts, see [Chapter 21, “Power Management IC \(PMIC\)”](#).

### 25.3.2 Conflicts with i.MX31 3-Stack

The touch driver requires a timer to obtain the time measurements among the different ADC samples. The EPIT2 timer is dedicated for use with the touch panel and cannot be used by any other module.

## 25.4 Software Operation

The touch screen driver reads user input from the 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 basic algorithm uses the following calls to sample and calibrate the screen with the touch screen driver:

- Calls the TouchPanelEnable function to start the screen sampling
- Calls the TouchPanelGetDeviceCaps function to request the number of sampling points

To test every calibration point, use the following steps:

1. Call the TouchPanelGetDeviceCaps to obtain a calibration coordinate. A crosshair appears on the screen. Touching the crosshair starts the calibration.
2. Call the TouchPanelReadCalibrationPoint function to obtain the calibration data
3. Call the TouchPanelSetCalibration function to calculate the calibration coefficients

### 25.4.1 Touch Driver Registry Settings

```
IF BSP_NOTOUCH !

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

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

```

    "Disable"=dword:FFFFFFFF
ENDIF ; BSP_PRECAL

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

[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]
    "CalibrationData"="517,526 788,249 785,801 261,799 250,255 "

; For launching the TouchPanel calibration application on boot.
[HKEY_LOCAL_MACHINE\init]
    "Launch80"="touchc.exe"
    "Depend80"=hex:14,00,1e,00 ; Wait for standard initialization
                                ; modules to load first (GWES.dll and
                                ; Device.exe).

ENDIF ; BSP_NOTOUCH !

```

## 25.5 Unit Tests

### 25.5.1 Unit Test Hardware

Table 25-2 lists the required hardware to run the unit tests.

**Table 25-2. Hardware Requirements**

Requirements	Description
EPSON L4F00242T03 VGA LCD Panel	Display panel required for display of graphics data.

### 25.5.2 Unit Test Software

Table 25-3 lists the required software to run the unit tests.

**Table 25-3. Software Requirements**

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
Touchtest.dll	Library containing the test
Touch.dll	Touch Panel Driver

The following errors are reported after the CETK Touch Panel test is performed:

- The Touch driver does not work after the CETK Touch Panel Test is performed. This is an MSFT CETK error. After the CETK Touch Panel Test is complete, the process shell.exe generates a "prefetch abort" exception on the touch.dll module and the touch panel driver does not work. This is the reason for this error are as follows:
  - GWES calls TouchPanelEnable to register a callback function when the OS is brought up

- When CETK runs, it also calls TouchPanelEnable to register its own callback function, and then the callback from the original GWES is lost
- When CETK ends, it does not call TouchPanelDisable. So the touch ISR is still running, and the CETK callback function memory has been free. When you click the panel, the ISR thread calls the callback function, which causes a "prefetch abort" exception.
- Case 8011, 9001-9003 fail. The touch panel displays lines even if you are drawing a circle or arc. This is also an MSFT CETK issue. The points are actually captured, but not painted in the allotted time.
- Case 8011 cannot draw on the right side of the screen after the screen is rotated 90 degrees. The executable ethca.exe works well after rotation. The CETK also works well when you run the case for the second time.

### 25.5.3 Building the Touch Panel Tests

To run the touch test cases, enter the following command:

**tux -o -n -d touchtest.dll -x <Test case id>**

This test must run in Kernel mode (-n option). To Run the CETK in kernel mode you must copy the ktux.dll from the CETK install directory to the release directory of your image.

For information about the test case IDs, see the Platform Builder Help:

**Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Touch Panel Tests > Touch Panel Test**

## 25.6 Touch Panel API Reference

To obtain the complete API reference, see the Platform Builder documentation:

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



## Chapter 26

# USB Boot and KITL

USB Boot and KITL are supported by implementing the RNDIS client device over USB on the target board. This feature configures the USB OTG port as a USB device and implements the RNDIS USB function driver. The USB RNDIS device acts as a standard Ethernet device and connects to the PC over a USB cable. Once connected, EBOOT and KITL can use the RNDIS Ethernet device.

### 26.1 USB Boot and KITL Summary

Table 26-1 identifies the source code location, library dependencies, and other BSP information.

**Table 26-1. USB Boot and KITL Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	IMX313DS
Target SoC (TGTSOC)	MX31_FSL_V1
MXARM11 SoC Driver Path	N/A
SoC Driver Path	N/A
SoC Static Library	N/A
Platform Driver Path	\\WINCE600\\PLATFORM\\<TGTPLAT>\\SRC\\COMMON\\USBFN \\WINCE600\\PLATFORM\\<TGTPLAT>\\SRC\\KITL
Import Library	fsl_usbfn_rndiskitl.lib
Driver DLL	kitl.dll
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	N/A

### 26.2 Supported Functionality

The USB Boot and KITL enables the 3-Stack board to provide the following software and hardware support:

- Supports image downloading over USB
- Supports KITL over USB
- Provides menu options to determine whether to enable USB Boot and/or USB KITL

## 26.3 Hardware Operation

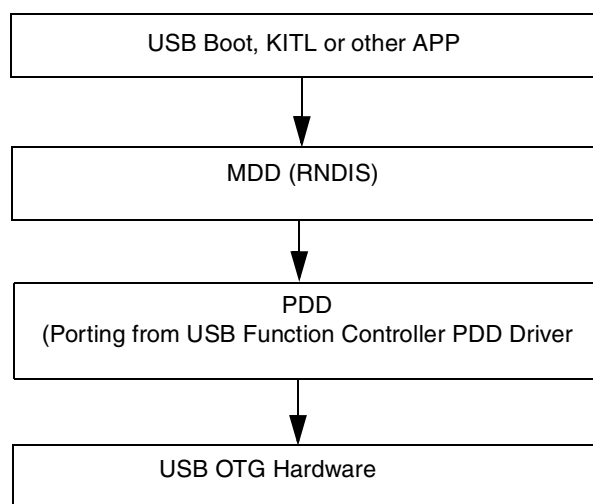
### 26.3.1 Conflicts with Other Peripherals

The USB Boot and KITL do not have conflicts with any other module. However, because the USB KITL and USB OTG drivers in the OS share the same USB OTG hardware, the USB OTG drivers should be disabled in the catalog item when USB KITL is enabled. The USB Boot does not have this limitation.

## 26.4 Software Operation

### 26.4.1 Software Architecture

USB Boot and KITL are part of the EBOOT and KITL subsystem. An RNDIS client device is implemented to support USB Boot and KITL. [Figure 26-1](#) illustrates the USB Boot and KITL software architecture.



**Figure 26-1. USB Boot and KITL Software Architecture Block Diagram**

Microsoft has implemented the RNDIS client MDD driver in Windows Embedded CE 6.0. The code is in following location:

```
%_WINCEROOT%\Public\Common\Oak\Drivers\Ethdbg\Rne_mdd
```

It generates static library `Rne_mdd.lib`.

The USB function controller PDD driver is ported to EBOOT and KITL to support the USB Boot and KITL. For details of the USB function controller PDD driver, see the Platform Builder for Microsoft Windows CE 6.0 Help:

**Developing a Device Driver > Windows CE Drivers > USB Function Drivers > USB Function Controller Drivers > USB Function Controller Driver Reference > USB Function Controller PDD Functions.**

Windows Embedded CE 6.0 provides an example of USB Boot in the following location:

```
%_WINCEROOT%\Platform\MainstoneIII\Src\Common\Usbfn
```

## 26.4.2 Source Code Layout

The following files have been modified or added to support USB Boot and KITL.

- The following files add the USB function controller and RNDIS PDD driver:
  - %\_WINCEROOT%\Platform\<Target Platform>\Src\Common\Usbfn
  - %\_WINCEROOT%\Platform\<Target Platform>\Src\Common\dirs
- The following file adds the RNDIS device to the EBOOT Ethernet initialization routines:
  - %\_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Common\ether.c
- The following file is used to set up the KITL device LogicalLoc and PhysicalLoc to USBOTG physical address if USB KITL option in EBOOT menu is selected by user:
  - %\_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Common\main.c
- The following file implements the private *NKCreateStaticMapping()* function. This function is defined in OS by Microsoft. It is not defined for EBOOT while USB Boot requires this function. So it is manually defined. This function calls only OALPatoUA().
  - %\_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Common\utils.c
- The following file adds the USB and KITL options to the EBOOT menu:
  - %\_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Eboot\menu.c
- The following file adds *fsl\_usbfn\_rndiskitl.lib* and *rne\_mdd.lib*:
  - %\_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Eboot\sources
- The following files add the USB RNDIS KITL device in the KITL initialization routines:
  - %\_WINCEROOT%\Platform\<Target Platform>\Src\Kitl\kitl.c
  - %\_WINCEROOT%\Platform\<Target Platform>\Src\Kitl\sources

## 26.4.3 IOMUX and Pinout

The i.MX31 3-Stack board system uses external ULPI PHY for USB OTG. There are IOMUX settings for USB OTG with external ULPI PHY. See the following file for information about the IOMUX settings of the external ULPI PHY:

```
%_WINCEROOT%\Platform\<Target Platform>\Src\Common\Usbfn\rndiskitl\hwinit.c
```

## 26.4.4 Power Management

Power management is not yet implemented in USB Boot and KITL.

## 26.4.5 Registry Settings

There are no related register settings for the USB Boot and KITL.

## 26.4.6 DMA Support

Physical contiguous memory is required to support USB DMA. This memory region is hard coded in the following:

```
%_WINCEROOT%\Platform\<Target Patform>\Src\Common\Usbfn\Rndiskitl\rndis_pdd.c
```

It uses the BSP reserved USB image region (Start from IMAGE\_SHARE\_USBKITL\_RAM\_OFFSET). This region is dedicated for USB Boot and KITL.

## 26.5 Unit Test

### 26.5.1 Building the USB Boot and KITL

There are no special configuration options for building USB Boot and USB KITL. Building the BSP with the default configuration includes the USB Boot and KITL support. There is one exception: USB OTG drivers should be de-selected from the catalog item view before building the NK image, in case the user wants to use USB KITL. The reason for this is that USB KITL and OS USB drivers share the same USB OTG hardware and they cannot exist simultaneously. As a result, USB KITL cannot be used to debug the USB OTG drivers.

### 26.5.2 Testing USB Boot and KITL

To test USB Boot and KITL, use these steps:

1. Connect the target board to the PC with a USB cable, and then power on the board
2. At the EBOOT menu, change the boot configuration to match the following:
  - 0) IP address: 192.168.0.2
  - 1) Subnet Mask: 255.255.255.0
  - 3) DHCP: Disabled
  - I) Kitl interrupt mode: Disable
  - P) Kitl passive mode: Disable
  - R) USB KITL: Enable
3. Press 'u' to download the image over USB. If this is the first time you have run the USB Boot or KITL with the PC, the "Found New Hardware Wizard" dialog is displayed, and you will be prompted to install the driver for Microsoft Windows CE RNDIS virtual adapter on the Windows PC. For instructions on installing the driver, refer to:
 

```
WINCE600\PUBLIC\COMMON\OAK\DRIVERS\ETHDBG\RNDISMINI\HOST\howto.txt
```
4. After the driver is installed successfully, the Microsoft Windows CE RNDIS virtual adapter should be displayed in the Network Connections on the PC. Configure this network connection properly. Use a static IP address (such as 192.168.0.3) in the same subnet as the target board.
5. Check the Platform Builder **Target > Connectivity** options to make sure the target device is selected. Now you should be able to download the image in the same way as the normal EBOOT
6. To test USB KITL, press 'r' in EBOOT menu to enable USB KITL. After NK starts up, the KITL works over the USB.

## Chapter 27

# 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 i.MX31. The OTG driver automatically selects either host or device functionality 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 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).

## 27.1 USB OTG Driver Summary

### 27.1.1 OTG Client Driver Summary

Table 27-1 provides a summary of source code location, library dependencies and other BSP information for the OTG Client Driver.

**Table 27-1. OTG Client Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	IMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
CSP Driver Path	..\SOC\Freescale\MX31_FSL_V1\USBD ..\SOC\Freescale\MX31_FSL_V1\USBFN
CSP Static Library	usbfn_mx31_fsl_v1.lib ufnmddbase_mx31_fsl_v1.lib
Platform Driver Path	\PLATFORM\IMX313DS\SRC\DRIVERS\USBD
Import Library	N/A
Driver DLL	usbfn.dll
Catalog Item	High Speed OTG: Third Party > BSP > Freescale i.MX31 3DS: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device To support only client/device mode, choose .. > High Speed OTG Port Pure Client Function
SYSGEN Dependency	SYSGEN_USBFN=1
BSP Environment Variable	BSP_USB=1 BSP_USB_HSOTG_CLIENT=1

**NOTE**

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 27.4.5, “Function Drivers”](#)) should be configured through drag and drop. If more than one is configured, the serial function is the default unless the registry is manually modified.

## 27.1.2 OTG Host Driver Summary

[Table 27-2](#) provides a summary of source code location, library dependencies and other BSP information for the OTG Host Driver.

**Table 27-2. OTG Host Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	IMX313DS
Target SOC (TGTSOC)	MX31
CSP Driver Path	..\SOC\Freescale\MX31_FSL_V1\USBH\EHCI ..\SOC\Freescale\MX31_FSL_V1\USBH\EHCIPDD ..\SOC\Freescale\MX31_FSL_V1\USBH\USB2COM
CSP Static Library	Ehcdmdd_mx31_fsl_v1.lib ehci_lib_mx31_fsl_v1.lib hcd2lib_mx31_fsl_v1.lib
Platform Driver Path	\PLATFORM\IMX313DS\SRC\DRIVERS\USBH\HSOTG
Import Library	N/A
Driver DLL	hcd_hsotg.dll
Catalog Item	Third Party > BSP > Freescale i.MX31 3DS: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device To support only host mode, choose .. > High Speed OTG Port Pure Host Function
SYSGEN Dependency	SYSGEN_USB=1
BSP Environment Variable	BSP_USB=1 BSP_USB_HSOTG_HOST=1

**NOTE**

The host driver requires a set of class drivers to be loaded. See [Section 27.4.6, “Class Drivers”](#) for more information.

## 27.1.3 OTG Transceiver Driver Summary (For HIGH-SPEED only)

[Table 27-3](#) provides a summary of source code location, library dependencies and other BSP information for the OTG Transceiver Driver.

Table 27-3. USB Transceiver Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	IMX313DS
Target SOC (TGTSOC)	MX31
CSP Driver Path	..\SOC\Freescale\MX31_FSL_V1\USBXVR
CSP Static Library	xvc_mx31_fsl_v1.lib
Platform Driver Path	\PLATFORM\IMX313DS\SRC\DRIVERS\USBXVR
Import Library	N/A
Driver DLL	imx_xvc.dll
Catalog Item	Third Party > BSPs > Freescale i.MX31 3DS: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device > High Speed OTG Port Full OTG Function
SYSGEN Dependency	SYSGEN_USBFN=1
BSP Environment Variable	BSP_USB=1 BSP_USB_HSOTG_CLIENT=1 BSP_USB_HSOTG_HOST=1 BSP_USB_HSOTG_XVC=1

## 27.2 Supported Functionality

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

- The High Speed OTG driver supports USB specification 2.0
- The driver is configured as client/peripheral by default, with one function driver defined as default. When nothing is connected to the OTG port, the port does not drive Vbus and waits attachment to a host by raising its D+ signal. On attachment of a mini-A plug, the driver switches to host mode.
- When a mini-B plug is connected to the OTG port, and the cable's opposite end is connected through mini-A plug to another OTG device, or through A-type plug to a host, then the OTG initiates operation as peripheral and responds to USB protocol from the host
- When a mini-A plug is connected to the OTG port and the cable's opposite end is connected through mini-B plug to another OTG device, then the OTG initializes/re-initializes itself into host mode and begins to act as a host. The OTG port remains in host mode whenever a mini-A plug is mated to the OTG socket connector.
- The OTG port as client/peripheral supports mass storage, RNDIS and serial clients
- The OTG port as host supports mass storage, printer, HID and RNDIS classes
- When nothing is attached to the OTG port, the driver configures the controller and transceiver into a low power state
- When the system is suspended with nothing attached to the OTG port, the system wakes upon attachment of the port to a host or attachment of a device with mini-A plug
- 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 remains suspended when the OTG port connection is unplugged

- When the system resumes after suspend, any attached devices are enumerated and their class drivers are loaded appropriately
- Data transfer rates on the client port exceed 40 Mbits/sec in mass storage client mode
- Mass storage client mode passes USBCV1.3, which is the software part of USB Logo test suite

## 27.3 Hardware Operation

There is an OTG mini socket on the 3-Stack board (J10). The i.MX31 device contains a USB 2.0 core for handling OTG, which is connected to the external transceivers through IO multiplexer (IOMUX). Options are possible on i.MX31 hardware (such as routing signals on H1 signal port to OTG transceiver, with both controllers “offline”). This allows external peripherals direct connection to a transceiver. The OTG driver currently supports only one hardware MUX configuration.

The ID Pin Detect is supported through the Transceiver Driver (for High Speed OTG), and is constructed as a stream interface driver. The sample reference implementation that is provided with WinCE 6.0 installation contains more detail on how the USB Host controller and USB Function controller driver are structured. The i.MX31 processor supports speed translation within the USB 2.0 controller, a non-standard EHCI implementation. As a result, the software does not currently support full/low speed devices (aside from those non-FS hub devices directly connected to the OTG port).

The 3-Stack 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 3-Stack. Self-powered hubs are required to expand the number of USB sockets and also to support devices that require greater than 100mA (for example: Mini HDD devices should be connected through self powered Hub).

### 27.3.1 Conflicts with Other Peripherals

The high speed OTG port conflicts with UART4. The USB controller drivers coordinate their management of the USB peripheral block clock and processor core voltage, as described in [Section 27.4.4.1, “Special i.MX31 Vcore Requirements](#) and [Section 27.4.4.2, “Clock Gating](#).

### 27.3.2 Signal Quality Requirement

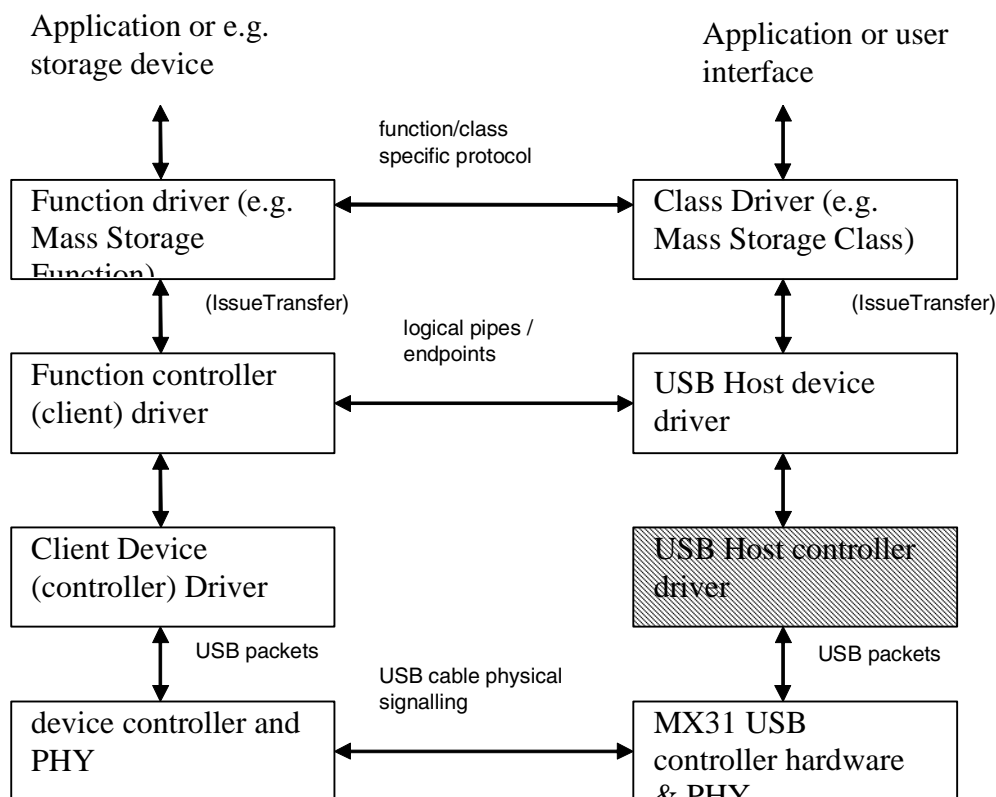
The USBCV test loads another USB host driver on the PC side. This one has more strict requirements for signal quality than the original one. Therefore, the platform must pass the USB signal quality test before the software test. The original board design has a defect. There is a 33  $\Omega$  resistor on both the DP and DM pins, which does not follow the standard and deteriorates the signal quality. They should be substituted with 0  $\Omega$  resistors.

## 27.4 Software Operation

### 27.4.1 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 27-1](#).





**Figure 27-1. Windows USB Driver Architecture**

For further details of the Windows CE USB driver architecture and usage, see Platform Builder Windows CE 6.0 help topic:

### Developing a Device Driver > Windows Embedded CE Drivers > USB Host Drivers

and

### Developing a Device Driver > Windows Embedded CE Drivers > USB Host Drivers > USB Host Controller Drivers > USB Host Controller Driver Development Concepts

When 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 3-Stack, the host controller driver enumerates and activates the appropriate class driver.

Windows CE 6.0 supports the following USB class drivers:

- Mass Storage – SD cards, MMC cards, CF cards, HDD drive, thumb drive (disk-on-key). Note that 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

#### 27.4.1.1 User Interface

As described above, user access to the USB host driver is through class drivers. For further details on these Host Client Drivers refer to Windows CE 6.0 Platform Builder help topic:

**Developing a Device Driver > Windows Embedded 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 (for example, IssueBulkTransfer) as documented in:

**Developing a Device Driver > Windows Embedded CE Drivers > USB Host Drivers > USB Host Controller Drivers > USB Host Client Drivers > Host Client Driver Reference.**

#### 27.4.1.2 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 [Section 27.4.1.5, “Registry Settings](#) for details on excluding OTG host support from the build.

The internal i.MX31 USB OTG signals can be multiplexed to a choice of pins on the device, as described for the IOMUX in the hardware reference manual.

#### 27.4.1.3 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 the registry setting PhysicalPageSize.

### 27.4.1.4 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 does not exceed the configured power limit.

This power limit is implemented through the platform-specific function `BSPUsbhCheckConfigPower()`, which is described in [Section 27.4.1.8.1, “BSPCheckConfigPower](#), and is located in the following directory:

```
\PLATFORM\IMX313DS\SRC\DRIVERS\USBH\Common\hwinit.c
```

This function must be modified to correspond with the platform hardware capabilities.

The i.MX31 3-Stack 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 3-Stack. Self-powered hubs are required to expand the number of USB sockets and also to support devices that require greater than 100mA.

### 27.4.1.5 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 through platform.reg. They do not normally require customization. Default values are contained in hshotg.reg. [Table 27-4](#) shows the USB OTG host controller registry values.

**Table 27-4. USB OTG Host Controller Registry Settings**

Value	Type	Content	Description
Dll	sz	hcd_hshotg.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.
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 i.MX31 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 <code>IssueTransfer()</code> .

### 27.4.1.6 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 i.MX31 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:

```
\\PLATFORM\\IMX313DS\\SRC\\DRIVERS\\USBH\\Common\\hwinit.c
```

The test mode is entered upon initialization and cannot be exited. Normal USB operation is disabled when test mode support is compiled into the image.

### 27.4.1.7 Unit Test

The USB driver has many devices to be tested. Tests are performed manually and include connecting the devices, 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.

#### 27.4.1.7.1 USB Host Controller Driver Test

Documentation for the Windows CE 6.0 CETK USB Host tests is normally found under Platform Builder Windows CE product documentation:

**Debugging and Testing > Windows CE Test Kit > CE Test Kit**

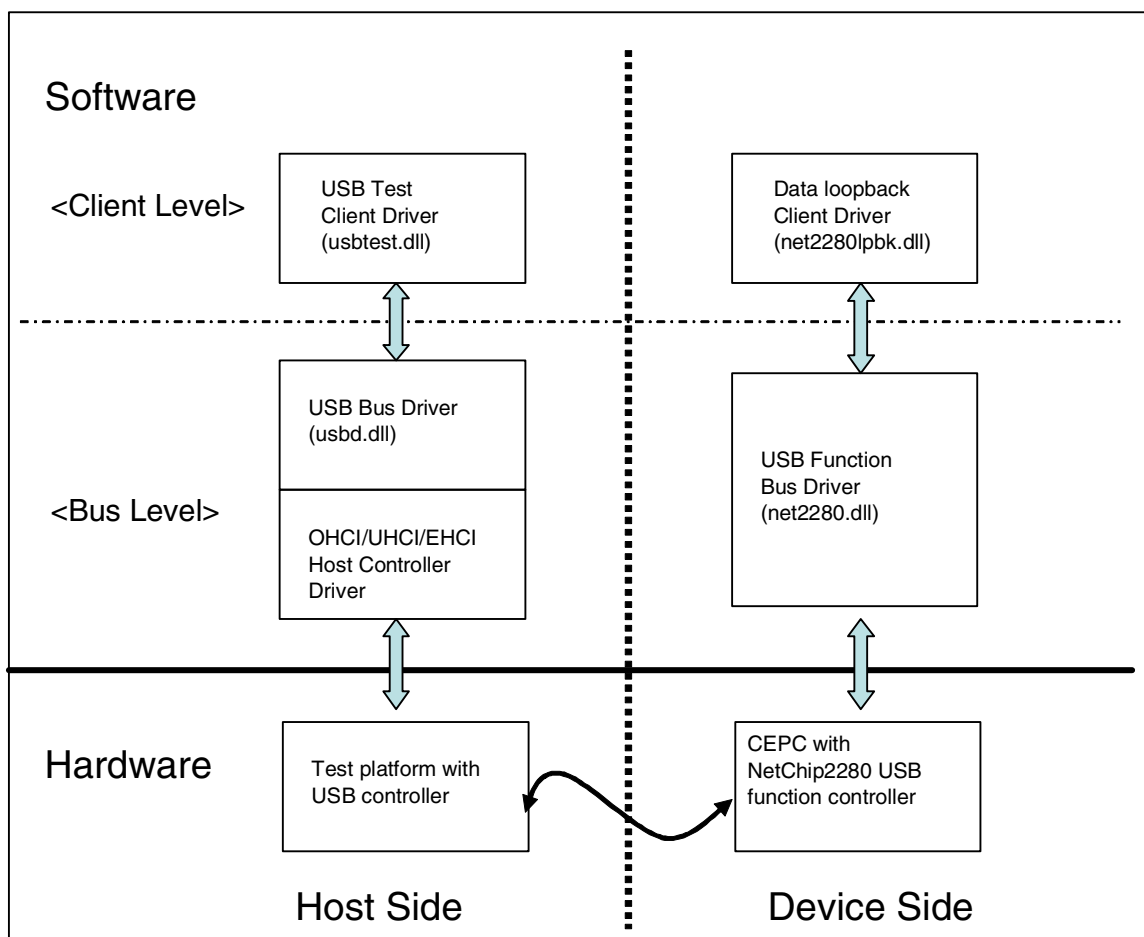
#### 27.4.1.7.2 Build the Image to be Tested

The following steps are used 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 i.MX31 3DS: ARMV4I – Device Drivers – USB Devices – USB High Speed Host 2
  - Core OS – Windows CE devices – Core OS Services – USB HOST Support and all the sub-components of this catalog item (for example, USB Storage Class Driver)
  - Core OS – Windows CE devices – File Systems And Data store – Storage Manager (Sub-Components: FAT File System, Partition Driver, Storage Manager control panel applet)
  - Device Drivers – USB Function – USB Function Clients – Serial

### 27.4.1.7.3 Abstract

This test suite can be used to test USB host controller drivers that provide the same interface as the Window CE USB host controller driver. The test setup and scenario is shown in [Figure 27-2](#).



**Figure 27-2. Test Setup**

This test suite acts as a client driver above USB bus driver (usbd.dll). It is loaded when the test device is connected to the host through USB cable. The test device is a CEPC with a NetChip2280 USB function controller card in it. After this CEPC is booted up and net2280lpbk.dll is loaded, the whole CEPC acts as a generic USB data loopback device. The USB test suite (the test client driver on the host side) can then stream 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 are exercised.

The NetChip2280 USB function PCI controller card is a USB2.0 compatible USB function device. It can be used to test both USB2.0 and USB1.1 host controllers (EHCI/OHCI/UHCI) and corresponding drivers.

Netchip2280 controller has six endpoints besides endpoint 0. The data loopback driver (net2280lback.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, receiving it back through IN pipe, and then verifying the data.

#### 27.4.1.7.4 Hardware Requirements

- Test platform
- Host Controller Card (if not onboard logic)
- CEPC
- Netchip2280 Card
- USB cable

#### 27.4.1.7.5 Software Requirements

Host side:

- Tux.exe
- Ddlx.dll
- Usbtest.dll
- Kato.dll
- USB component (usbd.dll, EHCI/OHCI/UHCI host controller driver(s)) must be included in the run time image.

Device side:

- Lufldr.exe
- Net2280lpbk.dll
- NetChip2280 USB function support (net2280.dll) must be included in the CEPC run time image.

#### 27.4.1.7.6 Running the Test

The test procedure is as follows:

1. Download runtime image to CEPC with Netchip2280 card on it
2. After the system has booted up, run command **s lufldr**, the tester should verify that net2280lpbk.dll is loaded
3. Download the runtime image to test platform with USB host controller on it
4. After the system has booted 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 indicates that there should be no connection between host and device side
6. After seven seconds, the test asks to connect the two sides with USB cable and the main test body starts to run
7. After the test(s) is(are) done and another test is to be run, do not disconnect the two sides of the USB cable. Just type the next test command, and the test will start directly. If the USB connection was disconnected before the next test, the test will ask to make the connection again before the test begins.

### 27.4.1.7.7 Test Cases

Table 27-5 shows the tests cases contained in the test suite.

**Table 27-5. USB Host Controller Driver Test Cases**

Test Case ID	Test Description
1001-1315, 1501-1515	<p>Data loopback tests.</p> <p>Concerning the transfer type, there are five categories:</p> <ol style="list-style-type: none"> <li>1) Bulk pipe loopback tests (tests with ID end with 1, like xxx1),</li> <li>2) Interrupt pipe loopback tests (tests with ID end with 2, xxx2),</li> <li>3) Isochronous pipe loopback tests (tests with ID end with 3, xxx3),</li> <li>4) All pipe transfer simultaneously (tests with ID end with 4, xxx4),</li> <li>5) All three types of transfer carry on simultaneously (tests with ID end with 5, xxx5)<sup>1</sup>.</li> </ol> <p>Concerning about how data is being transferred, there are also five categories:</p> <ol style="list-style-type: none"> <li>1) Normal loopback tests (tests with ID start with 10, like 10),</li> <li>2) loopback tests using physical memory (tests with ID start with 11, 11xx),</li> <li>3) loopback tests using a part of allocated physical memory (tests with ID start with 12, 12xx),</li> <li>4) Normal short transfer loopback tests ((tests with ID start with 13, 13xx),</li> <li>5) Stress short transfer loopback tests ((tests with ID start with 15, 15xx),</li> </ol> <p>Also, both synchronous and asynchronous transfer methods (test cases like xx1x using asynchronous transfer method, test cases like xx0x using synchronous method) are exercised.</p> <p>Therefore, there are <math>5 \times 5 \times 2 = 50</math> test cases.</p>
1401-1413	Some additional data loopback tests. They mainly focus on testing APIs like GetTransferStatus(), AbortTransfer() and CloseTransfer().
2001-2013	Test related with Device requests.
9001-9004	These are some special tests that test APIs like SuspendDevice(), ResumeDevice() and DisableDevice().
9005	This is a test that stresses EP0 transfer (Vendor Transfer)

<sup>1</sup> 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. Having all the tests listed above pass under this configuration is more than sufficient for a BVT-type test pass. However, testers can change the packet sizes (these values are hard-coded in the source code for net2280lpbk.dll) for each endpoint and run the test cases again for more comprehensive testing.

This test suite provides a way to change packet sizes of on NetChip2280 device on the fly as follows:

- Test case 3001: Using some very small packet sizes in NetChip2280 device's full speed configuration
- Test case 3002: Using some very small packet sizes in NetChip2280 device's high speed configuration
- Test case 3003: Using some irregular packet sizes (like non DWORD-aligned size) in NetChip2280 device's full speed configuration

- Test case 3004: Using some irregular packet sizes (like non DWORD-aligned size) in NetChip2280 device's high speed configuration
- Test case 3005 (High Speed only): Using some very large packet sizes (like  $2 \times 1024$  for Isochronous endpoints) in NetChip2280 device's full speed configuration. Note that in the real world, Netchip2280 can not handle transfers using such large packet size because its onboard FIFO buffer is small.

Run one of the test case above like running those normal tests, then after 15~20 seconds, through PB `usbtest.dll` will be seen unloading and loading again automatically. This means the packets sizes on netchip2280 side have already been changed. Then 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 performs split transfers.

NetChip2280 can be forced to be a full speed device. In the test setup stage, instead of run `s lufdrv` to load loopback driver, run `s lufdrv -f`. This forces Netchip2280 to be configured as a full speed device.

Also testers are encouraged to do some manual tests. Following are some examples:

- Plug in a USB device, suspend system, and then resume. USB devices should still be there.
- Plug in a USB device, suspend system, unplug it, plug in another device, then resume. System should enumerate the new device properly.
- Run one of the data transfer tests, in the middle of transfer stage, suspend the system (host side), then resume. Tests may fail, but system should not crash.
- Run one of the data transfer tests, in the middle of transfer stage, disconnect the USB connection. Tests should fail, but system should not crash.

## 27.4.1.8 Platform-Specific API

### 27.4.1.8.1 BSPCheckConfigPower

This function is used to evaluate whether a device can be supported on the specified USB port.

#### Parameters:

<i>UCHAR bPort</i>	[in] Unused. Each USB controller has only one port
<i>DWORD dwCfgPower</i>	[in] Power requirement (number of milliamps) requested by the device being evaluated for attachment support on this port
<i>DWORD dwTotalPower</i>	[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



### 27.4.1.8.2 BSPUsbSetWakeUp

This function does 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

### 27.4.1.8.3 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

### 27.4.1.8.4 SetPHYPowerMgmt

This function is called by the USB driver when transitioning to or from the suspended state (for example, during system suspend). The function does what is necessary to place the transceiver hardware into a suspended (*fSuspend* == TRUE) or running (*fSuspend* == FALSE) state.

The standard implementation for i.MX31/3-Stack uses a ULPI-bus based ISP1504 transceiver for the HS OTG port, and this function configures the ULPI-bus for sleep state. If the platform hardware uses other transceivers, this function needs to be modified appropriately.

**Parameters:**

*BOOL fSuspend* [in] TRUE: system/controller is going to suspend mode. FALSE: resuming

## 27.4.2 USB Client Driver

This driver enables the USB device functionality for the i.MX31 device. It is activated when a USB Mini B connector is connected to the Mini USB OTG socket. When the i.MX31 3-Stack board is connected to a USB host system (ex: high speed or full speed port of PC), it is enumerated according to the current configuration settings and the appropriate class driver is loaded on the PC. By default the 3-Stack board is configured for USB serial class. The 3-Stack 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

### 27.4.2.1 User Interface

The USB client driver provides a standard Windows CE USB driver implementation. For an overview see:

**Developing a Device Driver > Windows CE Drivers > USB Function Drivers > USB Function Controller Drivers.**

User access to the USB client driver is through function drivers such as Mass Storage or RNDIS. For further details on these USB Function drivers, refer to Windows CE 6.0 Platform Builder help topic:

**Developing a Device Driver > Windows Embedded CE Drivers > USB Function Client Drivers.**

Where new function driver code is to be developed, refer to the Function controller driver interface functions (for example, IssueTransfer) as documented in:

**Developing a Device Driver > Windows Embedded CE Drivers > USB Function Controller Drivers > USB Function Controller Driver Reference.**

### 27.4.2.2 Client Driver Configuration

The OTG client driver is configured into the BSP build by dragging & dropping the appropriate catalog item (See [Table 27-1](#)). 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).

### 27.4.2.3 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 through platform.reg. They do not normally require customization. [Table 27-6](#) shows the USB OTG client registry values.

**Table 27-6. USB OTG Client Registry Settings**

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 MX31 hardware.

### 27.4.2.4 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 cannot 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:

```
SOC\FREESCALE\MX31_FSL_V1\DRIVERS\USBFN\CONTROLLER\MDD
```

In addition, USBFN\_TEST\_MODE\_SUPPORT must be defined during compilation of the CSP USB device driver library.

### 27.4.2.5 Unit Test

There is no CETK test case for USB client (function) drivers. The USB Function is tested by configuring the i.MX31 3-Stack 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 to [Section 27.4.1.7.2, “Build the Image to be Tested](#) for more information.

#### 27.4.2.5.1 Unit Test Hardware

[Table 27-7](#) lists the required hardware to run the unit tests.

**Table 27-7. Hardware Requirements**

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.

#### 27.4.2.5.2 Unit Test Software

[Table 27-8](#) shows the software requirements for the USB Function controller driver test.

**Table 27-8. Software Requirements**

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.
USBCV1.3.	This is the host side software that is required for software part of USB Logo Test.

#### 27.4.2.5.3 Running the USB Function Controller Driver Tests

[Table 27-9](#) lists USB Function controller driver tests.

**Table 27-9. 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.	<p>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.</p> <p>Procedure:</p> <ol style="list-style-type: none"> <li>1. Connect the mini USB OTG plug B to the mini USB OTG socket.</li> <li>2. Observe that the ActiveSync on the host side gets connected and is synchronized.</li> <li>3. Copy files from Host system to the Mobile Device. Files are copied.</li> <li>4. Copy files from the Mobile Device to the Host system. Files gets copied.</li> <li>5. Unplug the mini USB OTG plug B from the i.MX31 mini USB OTG socket to unload the Serial class driver.</li> </ol> <p>Expected Result: ActiveSync should get synchronized and copying of files should happen between the Host and the 3-Stack board.</p>
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.	<p>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.</p> <p>Procedure:</p> <ol style="list-style-type: none"> <li>1. Connect the mini USB OTG plug B to the mini USB OTG socket.</li> <li>2. Observe that a new disk in My Computer having as Removable Disk appearing in it.</li> <li>3. Copy files from Host system to the new disk drive. Files are copied.</li> <li>4. Copy files from the new disk drive to the Host system. Files gets copied.</li> <li>5. Unplug the mini USB OTG plug B from the 3-Stack mini USB OTG socket to unload the mass storage class driver.</li> </ol> <p>Expected Result: Files copied into mass storage client device match those copied out (when compared on Windows XP PC using file compare utility). Note that files are not be visible from within the 3-Stack system until the system has been reset. The file system should not be used inside the 3-Stack when it is being accessed through USB as a mass storage client.</p>

Test Cases	Entry Criteria/Procedure/Expected Results
Board configured as USB RNDIS client and connected to a host system after the board boots up completely. Browsing the Internet.	<p>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.</p> <p>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, and so on. 4. On the 3-Stack 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, and so on. 5. In the Internet explorer on the 3-Stack board, configure the LAN settings as per the local area settings.</p> <p>Expected Result: Browsing the Internet should be possible.</p>
Board configured as USB Mass storage client, with SD drive as DSK1 mounted, and connected to a host system after the board boots up completely.	<p>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.</p> <p>Procedure: 1. Run USBCV1.3 on PC side 2. Plug the mini USB OTG plug B to the mini USB OTG socket, connect it with PC. 3. Run Chap-9 Test on USBCV1.3, select our board as test target. 4. Run MSC Test on USBCV1.3, select our platform as test target. 5. Unplug the mini USB OTG plug B from the 3-Stack mini USB OTG socket to unload the mass storage class driver.</p> <p>Expected Result: all test item should get passed.</p>

## 27.4.2.6 Platform-Specific API

### 27.4.2.6.1 InitializeMux

This function is called to initialize the IOMUX connection within i.MX31, 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

### 27.4.2.6.2 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

## 27.4.3 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 3-Stack. Upon detection the driver activates the USB host controller driver or USB function controller driver.

### 27.4.3.1 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 through its platform-specific routines to provide different behavior for power management (wake-up, D+ soft connect, and so on).

### 27.4.3.2 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.

### 27.4.3.3 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 through platform.reg. They do not normally require customization. [Table 27-10](#) shows the USB OTG transceiver registry values.

**Table 27-10. USB OTG Transceiver Registry Settings**

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 MX31 hardware.

### 27.4.3.4 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 3-Stack board. The test verifies that the USB host or Function controller driver is activated on cable plug-in.

#### 27.4.3.4.1 Unit Test Hardware

Table 27-11 lists the required hardware to run the unit tests.

**Table 27-11. USB Client Driver Hardware Requirements**

Requirements	Description
3-Stack board to act as a device.	3-Stack 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.

#### 27.4.3.4.2 Running the Transceiver Test

Table 27-12 lists Transceiver tests.

**Table 27-12. Transceiver Tests**

Test Cases	Entry Criteria/Procedure/Expected Results
Idle case when no cable plugged in	<p>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</p> <p>Procedure: When the board is powered and completely booted-up, the board should be idle (and as mass storage client, not verifiable)</p> <p>Expected Result: Device boots up and is stable</p>
Mass storage client visible from PC	<p>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</p> <p>Procedure: When the board is powered and completely booted-up, verify that board responds as a mass storage client when plugged into PC</p> <p>Expected Result: New storage must be visible on PC</p>
Mini USB OTG plug-A connected to the mini USB OTG socket of 3-Stack board and mouse plugged into OTG port through this cable	<p>Entry Criteria: Unplug board from PC (in previous step)</p> <p>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. For example, if a USB mouse is connected, on movement of the mouse, the pointer in the LCD screen is seen moving.</p> <p>Expected Result: Device should start functioning</p>

### 27.4.3.5 Platform-Specific API

The transceiver driver library code contains all i.MX31 chip-specific implementation, and is located in:

```
SOC\Freescale\MX31_FSL_V1\Drivers\USBXVR
```

The transceiver driver operation can be customized through the platform-specific code located in:

```
PLATFORM\IMX313DS\SRC\Drivers\USBXVR
```

The standard implementation located in hwinit.c is provided for the 3-Stack 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 through 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 3-Stack hardware platform.

#### 27.4.3.5.1 Structure BSP\_USB\_CALLBACK\_FNS

Structure BSP\_USB\_CALLBACK\_FNS is defined in MX31\_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() (Section 27.4.3.6.2, “RegisterCallback”).

```
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;
```

#### 27.4.3.5.2 pfnUSBPowerDown

This callback function is called during the Windows Embedded CE 6.0 power down sequence. The actual platform specific power down routine should be registered as this callback function. 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 automatically suspends 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 i.MX31, 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

### 27.4.3.6 pfnUSBPowerUp

Similar to `pfnUSBPowerDown`, this is called during the Windows Embedded CE 6.0 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 automatically suspends 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 [Section 27.4.3.5.2, “pfnUSBPowerDown](#).

#### 27.4.3.6.1 pfnUSBSetPhyPowerMode

This function is called when the system is in USB transceiver mode, with no USB activity. With standard implementation on 3-Stack, if the system is in transceiver mode and there is no activity in USB port for one second, the transceiver driver suspends 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 (for example, device attach), the transceiver driver sets the system to panic mode (requiring core voltage to stay high using `DDKClockEnablePanicMode()`, supported for i.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 i.MX31, 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

#### 27.4.3.6.2 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 transceiver of the 3-Stack. 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 function inside this structure is used by the CSP transceiver code.

## 27.4.4 Power Management

The following are the aspects of power management for the USB device drivers:

- Special i.MX31 Vcore requirements
- Clock gating to the USB peripheral block within the i.MX31
- Setting the transceiver to a lower power mode or suspend
- 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 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.

#### 27.4.4.1 Special i.MX31 Vcore Requirements

When ULPI-bus transceivers are used with the USB controller (for example, ISP1504 transceivers for High Speed OTG port and High Speed Host 2 port on i.MX31/3-Stack), normal DVFS scaling of the i.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 by 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.

#### 27.4.4.2 Clock Gating

The USB driver(s) for the various USB ports automatically manage clock gating to the i.MX31 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)
```

#### 27.4.4.3 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 [Section 27.4.4.4, “Transceiver Power Mode”](#)).

#### 27.4.4.4 Transceiver Power Mode

Software timeout is used to establish whether the transceivers and their related bus (for example, ULPI-bus for ISP1504 connection to i.MX31) 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.

#### 27.4.4.5 PowerUp

Each of the OTG client, host and transceiver drivers have PowerUp routine associated. For the host driver, this is referenced through the MDD to a function `PowerMgmtCallback()`.

For the host, the routine does the following:

- Verify the wake-up conditions through 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.

#### 27.4.4.6 PowerDown

As with the PowerUp routine, OTG client, host and transceiver drivers have PowerDown routine associated. For the host driver, this is referenced through the MDD to a function PowerMgmtCallback().

For the host, the routine does the following:

- Verify the wake-up conditions through 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 be made, the transceiver would wake through its wake-up mechanism and launch the appropriate (client or host) driver.

### 27.4.4.7 Suspend / Resume Operations

- Mass Storage Host/Client: Device is mounted automatically, but any unfinished browse/copy is terminated
- ActiveSync Client: Once browsing into the content of device. A system suspend/resume causes device to not be mounted until unplug and plug cable again
- HID Host: client is recognized again automatically

### 27.4.5 Function Drivers

The function drivers can be configured into the image through the Windows CE 6.0 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 applies):

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers]
"DefaultClientDriver"=- ; erase previous default
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers]
"DefaultClientDriver"="Mass_Storage_Class"
```

or

```
"DefaultClientDriver"="RNDIS"
```

or

```
"DefaultClientDriver"="Serial_Class"
```

Unless the BSP is configured with persistent registry storage, it only makes sense to configure a single function driver into the image and this one becomes 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 are also included in the image through logic in:  
PUBLIC\CEBASE\OAK\Misc\winceos.bat

### 27.4.5.1 Mass Storage Function

Table 27-13. Mass Storage Function

Driver Attribute	Definition
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	\PLATFORM\IMX313DS\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 uses 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 through 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
```

For commercial products, this default registry entry must be copied into platform.reg and modified to override the defaults. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Mass_Storage_Class]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
"IdVendor"=dword:045E
"Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
"IdProduct"=dword:FFFF
"Product"="Generic Mass Storage (PROTOTYPE--Remember to change idVendor)"
"bcdDevice"=dword:0
```

## 27.4.5.2 Serial Function

The primary use for Serial function is ActiveSync.

**Table 27-14. Serial Function**

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 ufncntlib.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. See microsoft.com to download the latest ActiveSync software for the PC. In some cases, DEBUGCHK may be triggered during attachment to ActiveSync in DEBUG builds.

When SYSGEN\_USBFN\_SERIAL is defined, the default registry entry is automatically included from:

```
PUBLIC\Common\OAK\FILES\common.reg
```

For commercial products, this registry entry must be copied into platform.reg and modified to over-ride the defaults. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Serial_Class]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
"IdVendor"=dword:045E
"Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
"IdProduct"=dword:00ce
"Product"="Generic Serial (PROTOTYPE--Remember to change idVendor)"
"bcdDevice"=dword:0
```

**27.4.5.3 RNDIS Function**

The RNDIS function allows communication over USB to be supplied to Ethernet NDIS interface of protocol stack.

**Table 27-15. RNDIS Function**

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
```

For commercial products, this registry entry must be copied into platform.reg and modified to over-ride the defaults. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\RNDIS]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
"idVendor"=dword:045E
"Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
"idProduct"=dword:0301
"Product"="Generic RNDIS (PROTOTYPE--Remember to change idVendor)"
"bcdDevice"=dword:0
```

## 27.4.6 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. 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.

[Table 27-16](#) shows the standard Microsoft-supplied drivers that are available by drag & drop from the catalog.

**Table 27-16. Class Drivers**

Class Driver	Configuration Flag	Catalog Item
HID	SYSGEN_USB_HID	Core OS—>Windows CE devices —>Core OS Services —> USB Host Support —> USB Human Input Device (HID) Class Driver
Printer	SYSGEN_USB_PRINTER	.. —> USB Printer Class Driver (and see additional configuration in <a href="#">Section 27.4.6.1, “Printer”</a> )
Keyboard	SYSGEN_USB_HID_KEYBOARD	.. —> Keyboard HID Device (and see additional configuration in <a href="#">Section 27.4.6.3, “HID Keyboard”</a> )
Mouse	SYSGEN_USB_HID_MOUSE	.. —> Mouse HID Device (and see additional configuration in <a href="#">Section 27.4.6.2, “HID Mouse”</a> )
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 are included by default from logic in: PUBLIC\CEBASE\OAK\Misc\winceos.bat



### 27.4.6.1 Printer

For printer support, printer driver/protocol support is required. For example, include PCL shown in [Table 27-17](#).

**Table 27-17. Printer Class Driver**

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**

### 27.4.6.2 HID Mouse

For mouse support, the cursor is required in order to test/use the mouse.

**Table 27-18. HID Mouse Class Driver**

Catalog Item	Configuration Flag	Catalog Item
HID	SYSGEN_CURSOR	Core OS → Shell and User Interface → User Interface → Customizable UI → Mouse

### 27.4.6.3 HID Keyboard

The 3-Stack Keyboard key mapping conflicts with that used for the HID keyboard. When USB keyboard support is included, remove the 3-Stack Keyboard ([Table 27-19](#)) and include the appropriate stub keyboard and keyboard .dll ([Table 27-20](#)).

**Table 27-19. HID Keyboard Driver to Remove**

Remove Item	Remove Catalog Item
Keyboard	Third Party → Freescale 3DS: ARMV4I → Device Drivers → Input Devices → Keyboard/Mouse → EVB Keypad

**Table 27-20. HID Keyboard Driver to Include**

Catalog Item	Configuration Flag	Catalog Item
NOP Stub Keyboard	BSP_KEYBD_NOP	Device Drivers → Input Devices → Keyboard/Mouse → NOP (Stub) Keyboard/Mouse English

Also include the appropriate keyboard .dll. For example, define SYSGEN\_KBD\_US and add the following lines in your platform.bib (immediately before the FILES section):

```
IF BSP_KEYBD_NOP
    kbdmouse.dll    $( _FLATRELEASEDIR )\KbdnopUs.dll           NK    SH
ENDIF; BSP_KEYBD_NOP
```

## 27.5 IRAM Patch

The USB link in the i.MX31 device uses very specific data structures, that is QH and TD. Data memory is also prepared before a transfer is primed. In the original design, memory is allocated in DRAM. HC locks the EMI AHB of DRAM when it performs the transfer. The lock prevents another module from accessing the DRAM. This causes problem for IPU module, which has strict real-time requirements. If HC occupies the DRAM AHB for too long, underflow occurs for the IPU module, which results in LCD display flicker. The problem becomes serious when a complex codec is used, which gives the IPU very high loadings. The IC should have wider bandwidth or a smarter arbiter to avoid such problem in the root.

A software workaround is to move all the USB related data structures and data memory to the IRAM area. This way, USB and IPU do not create such conflicts. The trade-off is that, as IRAM areas total size is 16K, all the data needed cannot be pulled into the IRAM simultaneously. Large transfers need to be split into smaller ones and primed one by one. The limited size of the IRAM memory also restricts the number of USB devices that can be used .

The detailed implementation of IRAM patch not in the scope of this document. To enable the patch, simply add an environment variable “BSP\_USB\_IRAM\_PATCH” = “1” in the project settings. As described, attach two more devices in one USB port to weaken the effect of this patch.

## 27.6 Basic Elements for Driver Development

This section provides the details of the basic elements for driver development in the 3-Stack BSP.

### 27.6.1 BSP Environment Variables

**Table 27-21. 3-Stack 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_HSOTG_HOST	Set to include USB host functionality on High Speed OTG port.

Pin conflicts between default driver implementations for the i.MX31 pin muxing (platform-specific implementation) mean certain configurations are mutually exclusive, as listed in the following table.

Table 27-22. Mutual Exclusive Driver Summary

Functionality <sup>1</sup>	BSP_ATA	BSP_CSPIBUS	BSP_USB	BSP_USB_HSOTG_XVC	BSP_USB_HSOTG_CLIENT	BSP_USB_HSOTG_HOST
ATA disk drive	yes	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				
High Speed Host (H2)	no	no				

<sup>1</sup> yes = Required, no = Not permitted, — = Don't care

## 27.6.2 Dependencies of Drivers

Table 27-23 summarizes the Microsoft defined environment variables used in the BSP.

Table 27-23. . 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

**Table 27-23. . USB Driver(Continued)**

<b>Names</b>	<b>Definition</b>
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 28

# WLAN Driver

The WLAN driver is used to drive the APM6628 module to implement Wi-Fi functionality. The WLAN module exchanges data with the i.MX31 device through the SDHC 2 port. The APM6628 module adopts Unifi V5 solution of Cambridge Silicon Radio company.

### 28.1 WLAN Driver Summary

WLAN driver is provided in binary form instead of source codes. [Table 28-1](#) provides a summary of the source code location, library dependencies, and other BSP information.

**Table 28-1. WLAN Client Driver Summary**

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX313DS
Target SOC (TGTSOC)	MX31_FSL_V1
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	..\PLATFORM\<TGTPLAT > \SRC\DRIVERS\wifi\csr
Import Library	N.A
Driver DLL	LoadDriveriMX31.exe loader.xbv sta.xbv ufmib.dat ufmp.dll ufsdio.dll
Catalog Item	Third Party > BSPs -> Freescale i.MX31 3DS: ARMV4I > Device Drivers > WiFi. > CSR > CSR APM6628 WiFi Core OS > CE BASE > Communication Services and Networking > Networking -Local Area Network [LAN] > Wireless LAN (802.11) STA - Automatic Configuration and 802.1x Core OS > CE BASE > Communication Services and Networking > Networking-General > Extensible Authentication Protocol Core OS > CE BASE > Security > Authentication Services (SSPI) > NTLM Core OS > CE BASE > Security > Authentication Services (SSPI) > Schannel (SSL/TLS) Core OS > CE BASE > Security > Microsoft Certificate Enrollment Tool Sample Core OS > CE BASE > Internet Client Services > Internet Explorer 6 for Windows CE Embedded Components > Windows Internet Services Core OS > CE BASE > Communication Services and Networking > Networking-General > Network Utilities (IpConfig, Ping, Route)
SYSGEN Dependency	SYSGEN_ETH_80211 SYSGEN_EAP SYSGEN_AUTH_NTLM SYSGEN_AUTH_SCHANNEL SYSGEN_ENROLL SYSGEN_WININET
BSP Environment Variable	BSP_CSR_WIFI=1 BSP_SDHC2 =1

The Recommended Catalog Items listed in [Table 28-1](#) should be included in the OS design in order to provide Wi-Fi functionality.

## 28.2 Supported Functionality

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

- Drives wifi module in APM6628 chip
- Supports scanning and connection to 802.11b AP with open security
- Supports scanning and connection to 802.11b/g AP with open security
- Supports scanning and connection to 802.11b/g AP with WEP(64/128/256) security
- Supports scanning and connection to 802.11b/g AP with WPA-PSK security
- Supports scanning and connection to adhoc laptop with open security
- Supports scanning and connection to adhoc laptop with WEP security
- Supports scanning and connection to 802.11g-only AP with open security

## 28.3 Hardware Operation

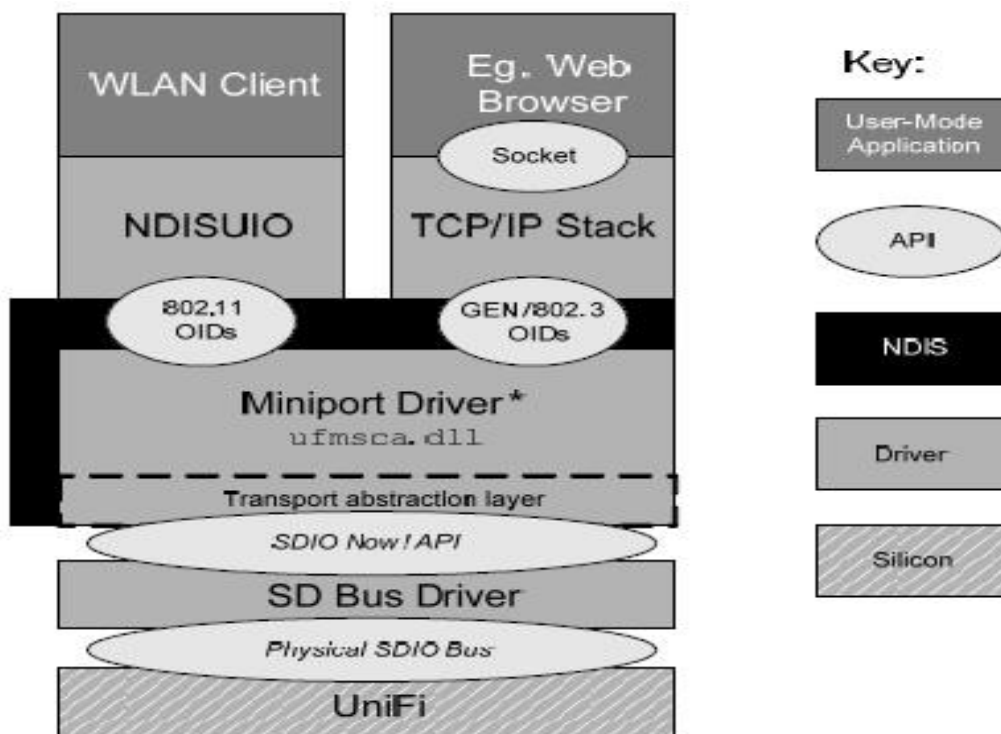
The Wi-Fi client driver exchanges data and commands between the SD stack and the Wi-Fi hardware through SDIO port.

### 28.3.1 Conflicts with Other Peripherals

Wi-Fi shares one reset pin with the Bluetooth module in the 3-Stack board.

## 28.4 Software Operation

The overall software architecture with WLAN Unifi driver for APM 6628 is depicted in [Figure 28-1](#).



**Figure 28-1. Software Architecture with WLAN Unifi Driver**

The main component is the miniport driver. The driver provides the means to configure the UniFi device for connecting to a wireless network to send and receive data. A suitable wireless client, such as Microsoft Wireless Zero Configuration can:

- Actively scan for wireless networks in the local area
- Connect to an unsecured or WEP-enabled infrastructure or ad-hoc network
- Connect to WPA-enabled networks using pre-shared key (PSK)
- Start an unsecured or WEP-enabled ad-hoc network (IBSS)

The device driver conforms to the following:

- The NDIS 5.1 specification (defined by the IEEE 802.11 Network Adapter Design Guidelines for Windows XP) for integration into the Windows operating system
- The UniFi Host Interface Protocol Specification for the exchange of signal primitives with the UniFi WLAN card.
- Network connections are set up using a wireless LAN client. The client issues a set of NDIS defined 802.11 OIDs to the miniport driver so that it can configure the UniFi device appropriately

### 28.4.1 Wi-Fi Registry setting

The following registry keys are required to properly load and configure WLAN driver

```
[HKEY_LOCAL_MACHINE\Comm\ufmp]
"DisplayName"="CSR UniFi v5.0"
```

```

"Group"="NDIS"
"ImagePath"="ufmp.dll"
"Dll"="ufmp.dll"
"Prefix"="NDL"

[HKEY_LOCAL_MACHINE\Comm\ufmp\Linkage]
"Route"=multi_sz:" "

[HKEY_LOCAL_MACHINE\Comm\ufmp1]
"DisplayName"="CSR UniFi v5.0"
"Group"="NDIS"
"ImagePath"="ufmp.dll"

[HKEY_LOCAL_MACHINE\Comm\ufmp1\Parms\TcpIp]
"EnabledDHCP"=dword:1
"IpAddress"="0.0.0.0"
"DefaultGateway"="0.0.0.0"
"UseZeroBroadcast"=dword:0
"Subnetmask"="0.0.0.0"

[HKEY_LOCAL_MACHINE\Comm\ufmp1\Parms]
"BusType"=dword:0
"BusNumber"=dword:0
"PowerMobileMode"=dword:0
"PollingModeEnabled"=dword:0
"SdioBusWidth"=dword:4
"SmeDebug"=dword:0
"CoreDebug"=dword:0
"DrvDebug"=dword:0

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-032A-CARDID-0001-FUNC-1]
"Dll"="ufmp.dll"
"Prefix"="NDL"
"Instance0"="ufmp:ufmp1"
"DbgLevel"="1"

```

## 28.5 Unit Test

WLAN test includes CETK test and manual WLAN connection without protection.

### 28.5.1 Unit Test Hardware

Table 28-2 lists the required hardware on the 3-Stack board to run the unit tests.

**Table 28-2. Hardware Requirements**

Requirement	Description
3 access points	The Access point supports open/wep/wpa-psk.
laptop	Used to setup adhoc network.
3-Stack board	Test board.



## 28.5.2 Unit Test Software

Table 28-3 lists the required software on the 3-Stack board to run the unit tests.

**Table 28-3. Software Requirements**

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

## 28.5.3 Running the WLAN Driver Tests

The Wi-Fi test suite requires three Wi-Fi access points to be present simultaneously, each configured for different encryption schemes. In case there is only one access point available, the tests can be split into three parts, depending upon encryption disabled: WEP 40-bit or WPA-PSK. Follow the steps below:

1. Create an ad-hoc Wi-Fi link (SSID: CE-ADHOC1) on a WLAN supported laptop or other WLAN-supporting device
2. Power On 3DS board, and click LoadDriveriMX31.exe to load the wifi. If the Wireless driver can be successfully loaded, Windows CE 6.0 displays a Window listing the available wireless networks
3. Select CE-ADHOC1 to connect, config the IPaddress
4. Use Ping tool to test the connection
5. Create an Open security AP, SSID: CE-OPEN, DHCP enable
6. Power On 3DS board, and click LoadDriveriMX31.exe to load the wifi. If the Wireless driver can be successfully loaded, Windows CE 6.0 displays a Window listing the available wireless networks
7. Select CE-ADHOC1 to connect
8. Use Ping tool to test the connection
9. Create an WEP 40-bit security AP, SSID:CE-OPEN,DHCP enable, key 0x1234567890
10. Power On 3DS board, and click LoadDriveriMX31.exe to load the wifi. If the Wireless driver can be successfully loaded, Windows CE 6.0 displays a Window listing the available wireless networks
11. Select CE-ADHOC1 to connect, set WEP , KEY 0x1234567890
12. Use Ping tool to test the connection
13. Create an WPA-PSK security AP, SSID:CE-OPEN,DHCP enable, key 12345678
14. Power On 3DS board, and click LoadDriveriMX31.exe to load the wifi. If the Wireless driver can be successfully loaded, Windows CE 6.0 displays a Window listing the available wireless networks
15. Select CE-ADHOC1 to connect, set WEP , KEY 12345678
16. Use Ping tool to test the connection.

## 28.5.4 Test the WLAN Communication without Protection

This test covers the practical functionality of the Wireless LAN driver to connect to any public wireless network for internet access. For this test it is required to have a 3-Stack board and any wireless access point (maybe a wireless router) without any protection to the internet access. The test is considered passed if the user can access <http://www.google.com> and view its contents. To run the test:

1. Turn on the board
2. If Windows CE 6.0 was correctly configured
3. Click LoadDriveriMX31.exe to load the wifi. If the Wireless driver can be successfully loaded, Windows CE 6.0 displays a Window listing the available wireless networks
4. Select the wireless network without protection that you can use to navigate through the Internet
5. Open Internet Explorer from the desktop icon
6. Navigate through the Internet to <http://www.google.com>

## Appendix A

### Frequently Asked Questions

#### A.1 How to Deal with Different Resolutions of the Display Panel?

The DirectDraw display driver does not support dynamic resolution change. You can specify the screen resolution in either of two locations during boot up:

- The `sd.c` file specifies the hardware-level register settings. Set the width and height using the variables `PANEL_INFO.width` and `PANEL_INFO.height`; these parameters identify to the SDC the size of the LCD panel. For more information about the register settings, see the chapter on the Image Processing Unit (IPU) in the *MCIMX31 and MCIMX31L Applications Processors Reference Manual*.
- The `sd.c.h` file specifies the actual panel size in the software settings. Set the size values using `SCREEN_PIX_WIDTH` and `SCREEN_PIX_HEIGHT`.

#### NOTE

If the user selects a large resolution, such as VGA (640x480), remember to adjust the video memory size, which is set in the registry. For details, see [Section 10.4.2.2, “Display Registry Settings.”](#)

#### A.2 How to Deal with Different Display Interface Formats?

The current driver setting uses the RGB565 interface format. It is recommended that you use the RGB565 interface, as it uses much less memory than the RGB888 interface, and it also provides a high quality color representation.

Currently, the display driver supports only RGB565 and YUV422 surface. Better performance cannot be obtained using the RGB666 and RGB888 display interfaces.

If the display panel does not support the RGB565 format, use other RGB format. Map RGB565 to RGBXXX by changing three registers: `DI_DISP3_B0_MAP`, `DI_DISP3_B1_MAP`, `DI_DISP3_B2_MAP`. These control the DI bus mapping unit. The setting of these registers is in the function `InitializeSDC` in the `sd.c` file. For further information about settings, see Section 44.4.6.4, “Bus Mapping Unit,” in *MCIMX31 and MCIMX31L Applications Processors Reference Manual*.

