i.MX35 3-Stack Windows Embedded CE 6.0

Reference Manual

Part Number: 924-76370
Rev. 2009.12
02/2010

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

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

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

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

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

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 010 5879 8000
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

freescale™
semiconductor

Contents

Chapter 4
Audio Driver

Chapter 5
Backlight Driver

Chapter 6

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

Chapter 7
Camera Driver

Chapter 8
Configurable Serial Peripheral Interface (CSPI) Driver

Chapter 9
Controller Area Network (CAN) Driver

Chapter 10
Chip Support Package Driver Development Kit (CSPDDK)

Chapter 11
Display Driver for IPUv1

Chapter 12
Dynamic Voltage and Frequency Control (DVFC) Driver

Chapter 13
Enhanced Secure Digital Host Controller (eSDHC) Driver

Chapter 14
Enhanced Serial Audio Interface (ESAI) Driver

Chapter 15
Fast Ethernet Controller (FEC) Driver

Chapter 16
FM Radio Driver

Chapter 17
General Purpose Timer (GPT) Driver

Chapter 18
Global Positioning System (GPS) Driver

Chapter 19
Graphics Processing Unit (GPU)

Chapter 20
Inter-Integrated Circuit (I^2C) Driver

Chapter 21
MediaLB Device Module (MLB)

Chapter 22
Micro Controller Unit (MCU) Driver

Chapter 23
NAND Flash Driver

Chapter 24
Power Management IC (PMIC)

Chapter 25
Serial Driver

Chapter 26
Sony/Philips Digital Interface (SPDIF) Driver

Chapter 27
Touch Panel Driver

Chapter 28
Universal Serial Bus (USB) OTG Driver

Chapter 29
USB Boot and KITL

About This Book

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

Audience

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

Suggested Reading

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

- i.MX35 Applications Processor Reference Manual
- i.MX35 3-Stack Release Notes for Windows Embedded CE 6.0
- i.MX35 3-Stack User's Guide for Windows Embedded CE 6.0
- Microsoft Platform Builder for Windows Embedded CE 6.0 Help

Conventions

This document uses the following notational conventions:

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

Definitions, Acronyms, and Abbreviations

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

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
CSPI	Configurable serial peripheral interface
D3DM	Direct 3D Mobile
DHCP	Dynamic host configuration protocol
DPTC	Dynamic power and temperature control
DVFC	Dynamic voltage and frequency control
DVFS	Dynamic voltage and frequency scaling
EBOOT	Ethernet bootloader
EVB	Platform evaluation board
FAL	Flash abstraction layer
FIR	Fast infrared
FMD	Flash media driver
GDI	Graphics display interface
GPT	General purpose timer
I^2C	Inter-integrated circuit
IDE	Integrated development environment
IST	Interrupt service thread
IPU	Image processing unit
KITL	Kernel independent transport layer
LVDS	Low-voltage differential signaling
MAC	Media access control
MMC	Multimedia cards
OAL	OEM adaptation layer
OEM	Original equipment manufacturer
OS	Operating system
OTG	On-the-go
PMIC	Power management IC
PQOAL	Production quality OEM adaptation layer
PWM	Pulse-width modulator
SD	Secure digital cards
SDC	Synchronous display controller
SDHC	Secure digital host controller
SDIO	Secure digital I/O and combo cards
SDRAM	Synchronous dynamic random access memory

Table i. Acronyms and Abbreviated Terms (continued)

Term	Meaning
SDK	Software development kit
SIM	Subscriber identification module
SOC	System on a chip
UART	Universal asynchronous receiver transmitter
USB	Universal serial bus

Chapter 1
Introduction

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

- i.MX35 3-Stack Development System

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

NOTE

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

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

1.1 Getting Started

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

1.2 Windows Embedded CE 6.0 Architecture

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

Welcome to Windows Embedded CE 6.0 > Windows Embedded CE Architecture

Chapter 2
Asynchronous Sample Rate Converter (ASRC) Driver

The Asynchronous Sample Rate Converter (ASRC) converts the sampling rate of as signal associated to an input clock into a signal associated to a different output clock. The ASRC supports concurrent sample rate conversion of up to 10 channels. The ASRC supports up to three sampling rate pairs, each pair should only have even number of channels.

2.1 ASRC Driver Summary

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

Table 2-1. ASRC Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\ASRC
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ASRC
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\ASRC
Driver DLL	asrc.dll
SDK Library	asrcbase_common_fsl_v2_PDK1_7.lib, asrc_common_fsl_v2_PDK1_7.lib, asrcbase_<*Target SOC*>.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > ASRC
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOAUDIO= BSP_ASRC=1

2.2 Supported Functionality

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

1. Supports standard stream interface for application usage.
2. For hardware limitation, supports only 24-bit wave format for both input and output. 24-bit audio data packed in 32-bit with LSB aligned, bit0–bit 23 are valid and bit24–bit31 are ignored by hardware.
3. Supports input sample rate range: 8K–96K
4. Supports output sample rate range: 32K–96K

5. One conversion pair (with two channels) is available for application usage (only for stereo wave conversion), other pairs are reserved for further audio driver usage.

2.3 Hardware Operation

ASRC is a 24-bit hardware module. Refer to the chapter on the Asynchronous Sample Rate Converter (ASRC) in the hardware specification document for detailed operation and programming information.

2.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

2.4 Software Operation

The ASRC driver follows the Microsoft standard stream interface driver architecture.

2.4.1 Required Catalog Items

N/A

2.4.2 ASRC Registry Settings

N/A

2.4.3 DMA Support

2.4.3.1 DMA Support

For the stream interface driver, two SDMA channels are allocated for data transfer: one for data transfer from memory to ASRC input fifo, and the other for data transfer from ASRC output fifo to memory. For both the input and output DMA, dual-buffer is used for chain operation.

2.4.4 Power Management

No power management is implemented yet in the ASRC driver.

2.4.4.1 PowerUp

This function is not implemented

2.4.4.2 PowerDown

This function is not implemented

2.4.4.3 IOCTL_POWER_CAPABILITIES

N/A

2.4.4.4 IOCTL_POWER_GET

N/A

2.4.4.5 IOCTL_POWER_SET

N/A

2.5 Unit Test

Because the supported wave format by ASRC is different from general wave file, the wave file used for ASRC test must be converted to the specified format (24-bit packed in 32-bit package, bit0–bit23 valid). The ASRC driver function can be tested by converting the wave file through the ASRC stream interface, and the output wave file can be verified by stereo audio playback function.

2.5.1 Building the Unit Tests

The source code for the ASRC test case be found under the directory:

```
\WINCE600\SUPPORT_PDK1_7\TEST\ASRC\
```

And there are three sub-directory in this directory:

```
\WINCE600\SUPPORT_PDK1_7\TEST\ASRC\FILE_CONVERT
\WINCE600\SUPPORT_PDK1_7\TEST\ASRC\ASRC_TEST
\WINCE600\SUPPORT_PDK1_7\TEST\ASRC\ASRC_PLAYER
```

To build each application, select “Open Release Directory in Build Window” in the IDE menu, enter the source code directory in the command prompt window, and type “build -c” to build the program.

2.5.2 Running the Unit Tests

Three simple applications are available for ASRC unit test: file_convert.exe, asrc_test.exe, asrc_player.exe.

- File_convert.exe can be used to convert general 16-bit wave file to the specific 24-bit wave file supported by ASRC.

  Example: `file_convert temp\input_16bit.wav temp\output_24bit.wav`
- Asrc_test.exe is used for the ASRC function test.

  Example: `asrc_test temp\input.wav temp\output.wav 48000`

  In this case, the test program reads data from the file input.wav, sends the audio data to ASRC module, reads back the data processed by ASRC and writes the output data to file output.wav. The sample rate is converted to 48K.
- Asrc_player is used for output wave file verification. This application directly plays back 24-bit wave file through stereo audio codec.

Example: `asrc_player temp\output.wav`

NOTE

These three applications are mainly used for simple function test and API demo usage. Users might encounter wave file format related failures (like wave format chunk length and fact chunk is not well handled). Editing the source code can resolve these problems.

2.6 ASRC Driver API Reference

The API follows the standard stream interface API. This section lists the SDK function for ASRC application interface.

2.6.1 ASRC SDK Functions

```
HANDLE ASRCOpenHandle(DWORD* pPairIndex);
BOOL ASRCCloseHandle(HANDLE hASRC, DWORD dwPairIndex);
BOOL ASRCOpenPair(HANDLE hASRC,PASRC_OPEN_PARAM pOpenParam );
BOOL ASRCGetCapability(HANDLE hASRC,PASRC_CAP_PARAM pCapParam);
BOOL ASRCClosePair(HANDLE hASRC,DWORD dwPairIndex );
BOOL ASRCConfig(HANDLE hASRC, PASRC_CONFIG_PARAM pConfigParam);
BOOL ASRCAddInputBuffer(HANDLE hASRC, PASRCHDR pHdrIn);
BOOL ASRCAddOutputBuffer(HANDLE hASRC, PASRCHDR pHdrOut);
BOOL ASRCStart(HANDLE hASRC, DWORD dwPairIndex);
BOOL ASRCStop(HANDLE hASRC, DWORD dwPairIndex);
```

Important note for using the SDK functions:

- Both input and output buffer length (number of bytes) must be a multiple of the internal ASRC DMA buffer size (which can be attained by ASRCGetCapability, ASRC_CAP_PARAM.dwInputBlockSize and ASRC_CAP_PARAM.dwOutputBlockSize), or driver failure may occur.
- Do not call ASRCStop until the entire wave file has been processed. Because the ASRC internal memory might not be cleared, stopping the ASRC and re-starting it introduces noise.
- The ASRC hardware module continues procession after it is started. So input buffer under-run causes noise and more output data numbers than expected.

2.6.2 Example for Using SDK Functions

Below is some sample code for using the SDK functions, refer to the demo test application and design document for more details.

```
#include “asrc_sdk.h”
......
// request the asrc pair first
g_hASRC = ASRCOpenHandle(&g_dwPairIndex);

//qurery the capability
ASRCGetCapability(g_hASRC,&capParam);
// the input buffer size should be multiple of capParam.dwInputBlockSize, same for
output buffer.
```

```
// open the pair for operation
openParam.inputChnNum = 2; // for application usage, set this value as 2 now
openParam.outputChnNum = 2; //for application usage, set this value as 2 now
openParam.pairIndex = (ASRC_PAIR_INDEX)g_dwPairIndex;
openParam.hEventInputDone = g_hInputEvent;
openParam.hEventOutputDone = g_hOutputEvent;
ASRCOpenPair(g_hASRC,&openParam );

// config the pair for conversion
configParam.clkMode = ASRC_CLK_NONE_SRC;
configParam.inputBitClkRate = g_dwInputSampleRate*2*24;
configParam.outputBitClkRate= g_dwOutputSampleRate*2*24;
configParam.inputSampleRate = g_dwInputSampleRate;
configParam.outputSampleRate = g_dwOutputSampleRate;
ASRCConfig(g_hASRC,&configParam );
.......
//add input buffers
for(i=0;i<INPUT_BUF_NUM;i++){
        ASRCAddInputBuffer(g_hASRC, &g_hdrInput[i]);
}
......
//add output buffers
for(i=0;i<OUTPUT_BUF_NUM;i++){
        ASRCAddOutputBuffer(g_hASRC, &g_hdrOutput[i]);
}
//start conversion
ASRCStart(g_hASRC,g_dwPairIndex);
......
// wait for the input event
WaitForSingleObject(g_hInputEvent, INFINITE);
// handle the input buffer here
......
//wait for the output event
WaitForSingleObject(g_hOutputEvent, INFINITE);
//handle the output buffer here
......
//when all the input data is processed, and output data has been received as expected,
stop it
ASRCStop(g_hASRC,g_dwPairIndex);
// close pair
ASRCClosePair(g_hASRC,g_dwPairIndex );
// release the pair
ASRCCloseHandle(g_hASRC, g_dwPairIndex);
```

2.6.3 Memory->ASRC->Memory->SSI Mode

In the general mode, ASRC is used for Memory->ASRC->Memory audio data transfer, which means the user data from memory buffer (audio file) is send to ASRC, converted and then put back into memory (audio file). In this mode, ASRC will chose the fast working clock for transfer. But in quite a lot application cases, users may want to send the data converted by asrc directly to the waveform audio device for playback instead of store them in files. To do this ,users need to use the Memory->ASRC->Memory->SSI mode. In this mode ,the ASRC working clock is synchronous to the wave device clock, so during the same interval, the audio data produced by ASRC can be just comsumed by wave device, and it will be easy for users to manager the data buffers.

To use this mode, users need to set different clkMode while config converstion pair, and outputSampleRate must be set correctly according to the wave device:

```
...
configParam.clkMode = ASRC_CLK_ONE_SRC_OUTPUT__AUTO_SEL;
...
configParam.outputSampleRate = g_dwOutputSampleRate;
ASRCConfig(g_hASRC,&configParam );
...
```

In this mode, clk Mode is set as ASRC_CLK_ONE_SRC_OUTPUT__AUTO_SEL, while in general mode it is set as ASRC_CLK_NONE_SRC. The others are same.

Also, another two SDK functions are provided to support this working mode:

```
BOOL ASRCSuspend(HANDLE hASRC, DWORD dwPairIndex);
BOOL ASRCResume(HANDLE hASRC, DWORD dwPairIndex);
```

The suspend function can be used to halt the conversion when there is the risk that the buffers used to keep the data produced by ASRC might be overrunned. And the resume function is then called to continue the converstion when the buffer level becomes normal.

Chapter 3
ATA/ATAPI Driver

ATA/ATAPI driver in WinCE 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, DISK_IOCTL_XXX, IOCTL_CDROM_XXX, IOCTL_DVD_XXX). The file system uses these I/O control codes to access the ATA/ATAPI devices.

ATAPI driver uses the ATA bus and interface to send command packets to ATAPI device.

3.1 ATA/ATAPI Driver Summary

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

Table 3-1. ATA/ATAPI Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\ATA
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ATA
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\BLOCK\ATA (ATA driver) ..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\BLOCK\ATAPI (ATAPI driver)
Driver DLL	ata.dll (ATA driver) mxatapi.dll (ATAPI driver)
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX35 3DS PDK1_7: ARMV4I > Storage Drivers > ATA (ATA driver) Third Party > BSP > Freescale i.MX35 3DS PDK1_7: ARMV4I > Storage Drivers > ATAPI (ATAPI driver)
SYSGEN Dependency	SYSGEN_ATADISK,SYSGEN_STOREMGR_CPL,SYSGEN_MSPART,SYSGEN_FATFS,SYSGEN_EXFAT (ATA driver) SYSGEN_UDFS (ATAPI driver)
BSP Environment Variable	BSP_NOATA= (for ATA driver) BSP_NOATAPI= (for ATAPI driver)

3.2 Supported Functionality

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

1. Provides standard Microsoft Block Storage Device API, including disk information management, formatting, block data read/write with full scatter-gather buffer support

2. Supports two power management modes, full on and full off
3. Driver reuse buffers allocated by upper layer by using DMA scatter/gather list to improve performance by reducing data copies
4. Supports FAT file system
5. Supports exFAT file system
6. Supports TFAT file system
7. Supports standard bus timing mode for UDMA mode 5 (optional support other modes such as PIO modes 0-4, MDMA modes 0-2, and UDMA modes 0-4)
8. Supports full sustained (media) data throughput capacity of Hitachi TravelStar C4K40 (or equivalent) at UDMA mode 5

NOTE

UDMA5 mode requires 80MHz bus clock or above.

This mode can apply to i.MX35 (133MHz bus clock).

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

1. Provides standard Microsoft Block Storage Device API, including disk information management, block data read with full scatter-gather buffer support
2. Supports two power management modes, full on and full off
3. Supports standard bus timing mode for PIO mode 0-4 (currently DMA mode is not supported).
4. Supports full sustained (media) data throughput capacity of SAMSUNG DVD-ROM DRIVE SH-D162D(TS-H352D) (or equivalent) at PIO mode 4.

3.3 Hardware Operation

The i.MX SOC contains an on-chip ATA controller. Refer to the chapter on the ATA in the hardware specification document for detailed operation and programming information. Data transfers on the ATA bus can take place through:

- CPU programmed data transfers via ATA controller registers. (Programmed I/O (PIO) modes modes 0-4)
- Multi-word DMA (MDMA modes 0-2)
- Ultra DMA (UDMA modes 0-5)

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

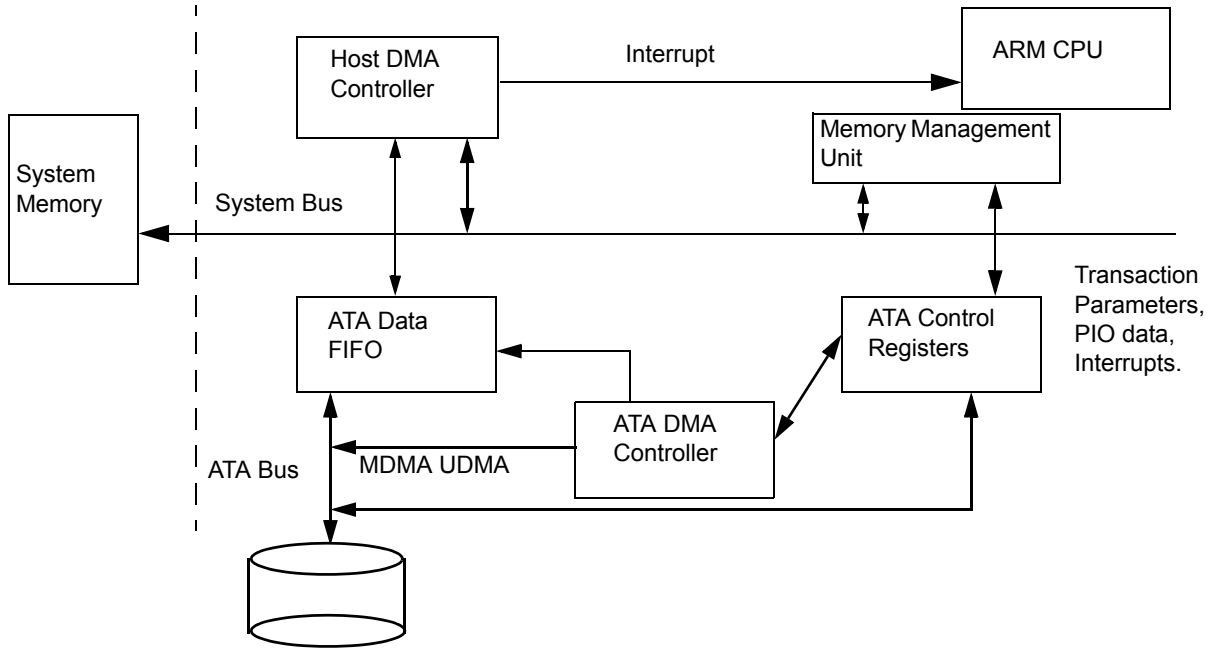

Figure 3-1. ATA Hardware Block Diagram

The i.MX SOC also contains a Host DMA controller which acts as a third-party bus mastering DMA, for transporting data between the ATA data FIFO and system memory. Host DMA controller 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.

NOTE

For i.MX35, the DMA controller can be Smart DMA controller (SDMA) or Advanced DMA controller (ADMA).

3.3.1 Conflicts with Other Peripherals and Catalog Options

3.3.1.1 Conflicts with SoC Peripherals

3.3.1.1.1 i.MX35 Peripheral Conflicts

- The ATA driver conflicts with ATAPI driver and can not be used together.
- When the ATA driver works in UDMA5 by ADMA controller in MX35 TO1, screen flickers due to memory bus bandwidth.

3.3.1.2 Conflicts with 3-Stack Peripherals

3.3.1.2.1 i.MX35 3-Stack Peripheral Conflicts

An ATAPI daughter board must be used to connect to CD/DVD ATAPI device.

3.3.2 Cabling

The ATA/ATAPI specification requires an 80-conductor cable when used in UDMA modes 3 or greater, otherwise read/write failures may occur. This requirement 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/ATAPI drives

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

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

3.4.2 ATA/ATAPI Driver Configuration

The driver is configured into the BSP build by check the catalog item listed in Table 3-1. This defines the environment variable/configuration option: BSP_NOATA for ATA driver, BSP_NOATAPI for ATAPI driver. Configuration for the ATA/ATAPI 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”.

3.4.2.1.1 i.MX35 Timing

The ATA bus timings are based on the AHB bus clock, as defined in the hardware reference manual. The ATA/ATAPI driver requires a clock period of 7.5 ns (133 MHz).

3.4.2.2 Prefix and Index

The default device prefix is “DSK”.

When no Index is configured for the ATA/ATAPI block device, the bus enumerator assigns 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 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

3.4.2.4.1 i.MX35 Defaults

The following defaults are selected by the default platform.reg file supplied for the build

- The default mode for the ATA driver is transfer mode UDMA mode 5 by ADMA controller.
- The default mode for the ATAPI driver is transfer mode PIO mode 0.

3.4.3 Power Management

The ATA/ATAPI supports two power management modes, ON (D0) and OFF (D4). These modes are managed via the standard Windows Power Manager. Power Manager uses IOCTL_POWER_SET to switch the disk 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/ATAPI 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/ATAPI driver turns ON the clock and enables the ATA/ATAPI module before processing any data transfer. After the block of data has been processed, the ATA/ATAPI module is disabled and the clock is turned OFF.

3.4.3.1 PowerUp

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

3.4.3.2 PowerDown

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

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]
```

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

Table 3-2. ATA Driver Registry Setting Values

Value	Type	Content	Description
Dll	sz	ata.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
		...	...
		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 shown in Table 3-3.

Table 3-3. ATA Driver Registry Setting Values

Value	Type	Content	Description
Prefix	sz	“DSK”	Device identifier (combined with Index for DSK1 for example)
Index	dword	1	Instance of ATA drive (if not configured in the registry, autoumatically assigned when driver loads)
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
```

3.4.4.1 i.MX35 Registry Settings

There are more registry settings for ATA driver SDMA/ADMA selection and for ATAPI driver:

3.4.4.1.1 ATA Driver SDMA/ADMA Selection

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ATA]
```

Table 3-4. i.MX35 ATA Registry Settings

Value	Type	Content	Description
DMAMode	dword	0	SDMA
		1	ADMA

3.4.4.1.2 ATAPI Driver

The ATAPI 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\ATAPI]
```

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

Table 3-5. i.MX35 ATAPI Registry Settings

Value	Type	Content	Description
Dll	sz	mxatapi.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
InterruptDriven	dword	01 (00)	enable interrupt driven I/O use for PIO or MDMA/UDMA modes (disable interrupt; not used normally)
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
"TransferMode"=dword:0c         ; 08-PIO mode 0, ..., 0C-PIO mode 4
"IORDYEnable"=dword:01          ; enable Host IORDY for PIO mode 3, 4
```

Standard registry entries also to be included for the ATAPI device under the above key as shown in Table 3-6.

Table 3-6. ATAPI Registry Settings

Value	Type	Content	Description
Prefix	sz	“DSK”	Device identifier (combined with Index for DSK1 for example)
Index	dword	1	Instance of ATAPI drive (if not configured in the registry, autoumatically assigned when driver loads)
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
CDProfile	sz	“CDProfile”	Storage Manager profile to be used in GetDeviceInfo (see below)

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

```
[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\CDProfile]
"Name"="IDE CDROM/DVD Drive"
"Folder"="CDROM Drive"

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\CDProfile\CDRom]
    "UseLegacyReadIOCTL"=dword:1
```

3.4.5 DMA Support

ATA driver supports DMA mode and non-DMA mode of transfer. The driver defaults to DMA mode of transfer. ATA supports three transfer-types: UDMA, MDMA and PIO mode. 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 ATA driver operates by host DMA controller, it always uses the scatter gather method.

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

3.4.5.1 i.MX35 DMA Support

- For the ATA driver, the Host DMA controller can be SDMA controller or ADMA controller.
- For the ATAPI driver, DMA mode is not supported.

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 ATA driver. The Storage Device test cases used to test ATA driver include:

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

The ATAPI 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 ATAPI driver. The Storage Device test cases used to test ATAPI driver include:

- Audio CD Driver Test cases
- CD/DVD-ROM Block Driver Test cases
- CD/DVD-ROM File System Driver Test cases

3.5.1 Unit Test Hardware

Table 3-7 lists the required hardware to run the ATA driver unit tests.

Table 3-7. ATA Driver Hardware Requirements

Requirement	Description
i.MX SOC and attached HITACHI hard disk C4K40.	Other drives supporting up to UDMA mode 3 may be used.

Table 3-8 lists the required hardware to run the ATAPI driver unit tests.

Table 3-8. ATAPI Driver Hardware Requirements

Requirement	Description
i.MX SOC and attached SAMSUNG DVD-ROM DRIVE SH-D162D(TS-H352D).	Other drives supporting up to PIO mode 4 may be used.

3.5.2 Unit Test Software

Table 3-9 lists the required software to run the Storage Device Tests.

Table 3-9. Software Requirements

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

Table 3-10. ATA Driver Software Requirements

Requirement	Description
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.

Table 3-11. ATAPI Driver Software Requirements

Requirement	Description
cddatest.dll	Test .dll file used to perform Audio CD Driver Test cases.
cdromtest.dll	Test .dll file used to perform CD/DVD-ROM Block Driver Test cases.
udftest.dll	Test .dll file used to perform CD/DVD-ROM File System Driver Test cases.

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. All the test .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

The tests can be launched from command line or CE Target Control window in Platform Builder.

3.5.4.1 ATA Driver

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

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

Format and create partition on disk before test. The command line option "-zorch" is case sensitive (help message within the test .dll is not correct) and is used to confirm over-writing of all information on the hard disk. Test cases 5019, 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 -sectors 65536"
```

NOTE

The free program memory to be adjusted to be larger than 64 Mbytes 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"
```

NOTE

Do not include NANDFlash driver or SD driver in the image, the CETK open DSK1 as default to test which may be NANDFlash or SD card instead of hard disk.

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

```
tux -o -d disktest_perf -c  "-profile 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"
```

NOTE

Do not include NANDFlash driver or SD driver in the image, the CETK open DSK1 as default to test which may be NANDFlash or SD card instead of hard disk.

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 allows the drive to provide maximum sustained data rate from the media, to ensure ATA driver supports the same. It is not necessary for all drive sectors to be tested, but the pre-compiled test does not have options to limit the portion tested, and all components are not publicly available for test customization. The test takes approximately four hours to execute on a 40 Gbyte drive. Tests using smaller contiguous chunks take even longer, and are not necessary for driver characterization.

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

- **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**

3.5.4.2 ATAPI Driver

The command line for running the Audio CD Driver Test is:

```
tux -o -d cddatest
```

Assesses the functionality of a CD-ROM block driver that supports the audio CD format

NOTE

Put audio CD into the CDROM drive

The command line for running the CD/DVD-ROM Block Driver Test is:

```
tux -o -d cdromtest
```

NOTE

A complete image of the CD or DVD media needs to be used for testing. The image is stored on the development workstation in a file named Cdsector.dat. To create Cdsector.dat for media in the CD-ROM drive or CD/DVD-ROM drive, run test case 6101.

The command line for running the CD/DVD-ROM File System Driver Test is:

```
tux -o -d udftest
```

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

- **Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Other Tests > Audio CD Driver Test**
- **Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Other Tests > CD/DVD-ROM Block Driver Test**
- **Windows Embedded CE Test Kit > CETK Tests and Test Tools > CETK Tests > Other Tests > CD/DVD-ROM File System Driver 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 3-12. BSP Environment Variables

Name	Definition
BSP_NOATA	Set to disable ATA device driver
BSP_NOATAPI	Set to disable ATAPI device driver

3.6.2 Mutual Exclusive Drivers

3.6.2.1 i.MX35 Mutual Exclusive Drivers

The ATA driver conflicts with ATAPI driver and they cannot be used together.

3.6.3 Dependencies of Drivers

Table 3-13 summarizes the Microsoft defined environment variables used in the BSP.

Table 3-13. Microsoft Defined Environment Variables

Names	Definition
SYSGEN_STOREMGR	Set to support storage manager

Table 3-13. Microsoft Defined Environment Variables

SYSGEN_STOREMGR_CPL	Set to support storage manager in control panel
SYSGEN_MSPART	Set to support partition driver.

Table 3-14. ATA Driver Environment Variables

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

Table 3-15. ATAPI Driver Environment Variables

Names	Definition
SYSGEN_UDFS	Set to support CDFS/UDFS file system

3.7 Block Device API Reference

The primary interface to the ATA/ATAPI block device is through the standard Windows CE Block Device IOCTLs as described in the following sections. Application-level access to ATA/ATAPI 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 Device Manager and the bus enumerator to load the driver. When the registry settings for ATA/ATAPI 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

lpInBuffer [in] Pointer to a STORAGEDEVICEINFO structure.

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

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

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.

3.7.8 IOCTL_DISK_FLUSH_CACHE

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

Parameters [No parameters]

3.7.9 IOCTL_CDROM_DISC_INFO

This IOCTL retrieves disk information to fill the CDROM_DISCINFO structure.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_DISC_INFO to retrieve disk information and fill the CDROM_DISCINFO structure.
lpInBuf	[in] Set to NULL.
nInBufSize	[in] Set to zero.
lpOutBuf	[in, out] On input, set to the address of an allocated CDROM_DISCINFO structure. This is the memory needed for the structure and information storage. On output, a filled CDROM_DISCINFO structure.
nOutBufSize	[in] Set to the size of the CDROM_DISCINFO.
lpBytesReturned	[in, out] On input, address of a DWORD that receives the size in bytes of the data returned. On output, set to the number of bytes written to the supplied buffer.

3.7.10 IOCTL_CDROM_EJECT_MEDIA

The IOCTL ejects the CD-ROM.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_EJECT_MEDIA to eject the CD-ROM.
lpInBuf	[in] Set to NULL.
nInBufSize	[in] Set to zero.
lpOutBuf	[in] Set to NULL.
nOutBufSize	[in] Set to zero.
lpBytesReturned	[in] Set to NULL.

3.7.11 IOCTL_CDROM_GET_SENSE_DATA

This IOCTL specifies retrieval of CD-ROM sense information contained in a CD_SENSE_DATA structure.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_GET_SENSE_DATA to retrieve CD-ROM sense information and fill the CD_SENSE_DATA structure.
lpInBuf	[in] Set to NULL.
nInBufSize	[in] Set to zero.
lpOutBuf	[in, out] On input, set to the address of an allocated CD_SENSE_DATA structure. On output, a filled CD_SENSE_DATA structure.
nOutBufSize	[in] Set to the size of the CD_SENSE_DATA.
lpBytesReturned	[in, out] On input, address of a DWORD that receives the size in bytes of the data returned. On output, set to the number of bytes written to the supplied buffer.

3.7.12 IOCTL_CDROM_ISSUE_INQUIRY

This IOCTL retrieves information used in the INQUIRY_DATA structure.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_ISSUE_INQUIRY to retrieve information and fill the INQUIRY_DATA structure.
lpInBuf	[in] Set to NULL.
nInBufSize	[in] Set to zero.
lpOutBuf	[in, out] On input, set to the address of an allocated INQUIRY_DATA structure. On output, a filled INQUIRY_DATA structure.
nOutBufSize	[in] Set to the size of the INQUIRY_DATA.
lpBytesReturned	[in, out] On input, address of a DWORD that receives the size in bytes of the data returned. On output, set to the number of bytes written to the supplied buffer.

3.7.13 IOCTL_CDROM_PAUSE_AUDIO

This IOCTL suspends audio play.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_PAUSE_AUDIO to pause audio playback if it was playing.
lpInBuf	[in] Set to NULL.
nInBufSize	[in] Set to zero.
lpOutBuf	[in] Set to NULL.
nOutBufSize	[in] Set to zero.
lpBytesReturned	[in] Set to NULL.

3.7.14 IOCTL_CDROM_PLAY_AUDIO_MSF

This IOCTL plays audio from the specified range of the medium.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_PLAY_AUDIO_MSF to play audio based on the information in the CDROM_PLAY_AUDIO_MSF structure.
lpInBuf	[in] Set to the address of an allocated CDROM_PLAY_AUDIO_MSF structure.
nInBufSize	[in] Set to the size of the CDROM_PLAY_AUDIO_MSF structure.
lpOutBuf	[in] Set to NULL.
nOutBufSize	[in] Set to zero.
lpBytesReturned	[in, out] On input, address of a DWORD that receives the size in bytes of the data sent. On output, set to the number of bytes written from the supplied buffer.

3.7.15 IOCTL_CDROM_READ_SG

This IOCTL reads scatter buffers from the CD-ROM and the information is stored in the CDROM_READ structure.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_READ_SG to read scatter buffers from the CD-ROM and store the information in the CDROM_READ structure.
lpInBuf	[in] Set to the address of an allocated SGX_BUF structure.
nInBufSize	[in] Set to the size of the SGX_BUF.
lpOutBuf	[in, out] On input, set to the address of an allocated CDROM_READ structure. This is the memory needed for the structure and info storage. On output, a filled CDROM_READ structure.
nOutBufSize	[in] Set to the size of the CDROM_READ.
lpBytesReturned	[in, out] On input, address of a DWORD that receives the size in bytes of the data returned. On output, set to the number of bytes written to the supplied buffer.

3.7.16 IOCTL_CDROM_READ_TOC

This I/O control returns the table of contents of the medium.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_READ_TOC to retrieve the table of contents information and store it into the CDROM_TOC structure.
lpInBuf	[in] Set to NULL.
nInBufSize	[in] Set to zero.
lpOutBuf	[in, out] On input, set to the address of an allocated CDROM_TOC structure. On output, a filled CDROM_TOC structure.
nOutBufSize	[in] Set to the size of the CDROM_TOC.
lpBytesReturned	[in, out] On input, address of a DWORD that receives the size in bytes of the data returned. On output, set to the number of bytes written to the supplied buffer.

3.7.17 IOCTL_CDROM_RESUME_AUDIO

This IOCTL resumes a suspended audio operation.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_RESUME_AUDIO to resume audio playback if it was paused.
lpInBuf	[in] Set to NULL.
nInBufSize	[in] Set to zero.
lpOutBuf	[in] Set to NULL.
nOutBufSize	[in] Set to zero.
lpBytesReturned	[in] Set to NULL.

3.7.18 IOCTL_CDROM_SEEK_AUDIO_MSF

This IOCTL moves the heads to the specified minutes, seconds, and frames on the medium.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_SEEK_AUDIO_MSF.
lpInBuf	[in] Set to NULL.
nInBufSize	[in] Set to zero.
lpOutBuf	[in] Set to NULL.
nOutBufSize	[in] Set to zero.
lpBytesReturned	[in] Set to NULL.

3.7.19 IOCTL_CDROM_STOP_AUDIO

This IOCTL stops audio play.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_STOP_AUDIO to stop audio playback.
lpInBuf	[in] Set to NULL.
nInBufSize	[in] Set to zero.
lpOutBuf	[in] Set to NULL.
nOutBufSize	[in] Set to zero.
lpBytesReturned	[in] Set to NULL.

3.7.20 IOCTL_CDROM_TEST_UNIT_READY

This IOCTL retrieves disc ready information and fills the CDROM_TESTUNITREADY structure.

Parameters

dwIoControlCode	[in] Set to IOCTL_CDROM_TEST_UNIT_READY to retrieve disc ready information and fill the CDROM_TESTUNITREADY structure.
lpInBuf	[in] Set to NULL.
nInBufSize	[in] Set to zero.
lpOutBuf	[in, out] On input, set to the address of an allocated CDROM_TESTUNITREADY structure. This is the memory needed for the structure and info storage. On output, a filled CDROM_TESTUNITREADY structure.
nOutBufSize	[in] Set to the size of the CDROM_TESTUNITREADY.
lpBytesReturned	[in, out] On input, address of a DWORD that receives the size in bytes of the data returned. On output, set to the number of bytes written to the supplied buffer.

3.7.21 IOCTL_DVD_GET_REGION

This IOCTL returns DVD disk and drive regions.

Parameters

hDevice	[in] Set to a handle to a block device.
dwIoControlCode	[in] Specifies this IOCTL.
lpInBuffer	Not used.
nInBufferSize	Not used.
lpOutBuffer	[out] Pointer to a DVD_REGIONCE structure.
nOutBufferSize	Not used.
lpBytesReturned	Not used.
lpOverlapped	Not used.

Chapter 4
Audio Driver

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

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

4.1 Audio Driver Summary

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

Table 4-1. Audio Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\WAVEDEV2
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\WAVEDEV2
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\WAVEDEV2\SGTL5000
Driver DLL	wavedev2_sgtl5000.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX35-3DS:ARMV4I > Device Drivers > Audio > SGTL5000 Stereo Audio
SYSGEN Dependency	SYSGEN_AUDIO
BSP Environment Variables	BSP_NOAUDIO= BSP_AUDIO_SGTL5000=1

NOTE

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

4.2 Supported Functionality

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

1. Conforms to the audio driver architecture as defined for Windows Embedded CE 6.0 and all related operating systems
2. Double-buffered DMA operations to transfer audio data between memory and the hardware FIFO
3. Two power management modes: full on and full off
4. Full duplex playback and record
5. Minimizes power consumption at all times by using clock gating and by disabling all audio-related hardware components that are not actively being used
6. 8–96 KHz for both recording and playback
7. Mono and stereo 16-bit sample, and stereo 24-bit sample

4.3 Hardware Operation

4.3.1 Audio Hardware Design

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

4.3.1.1 i.MX35 3-Stack Audio Hardware Design

As the Synchronous Serial Interface is a full-duplex serial port, i.MX35 SoC uses instance 2 (SSI2) for both audio playback and recording. The external stereo codec SGTL5000 is connected to AUDMUX port 4 (external) while SSI2 is internally connected to AUDMUX port 2 (internal) by i.MX35 SoC design. Both ports are configured to operate in synchronous 4-wire mode.

The i.MX35 uses the I^2C bus interface to access SGTL5000 control registers, so that SGTL5000 can be configured by i.MX35 as per hardware design and software configuration.

The stereo codec SGTL5000 on i.MX35 3-Stack supports output to Headphone or Line Out, input from Microphone or Line In. The Line Out is designed to support speaker output, but there is currently no speaker device available on board, so only Headphone output is supported.

For operation and programming, see the chapters in the i.*MX35 Reference Manual* for the SSI, SDMA, AUDMUX, and IOMUX components, and see SGTL5000 *Datasheet* for Stereo Audio Codec SGTL5000.

4.3.2 Audio Playback

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

- The audio driver is configured to use SSI2 for I^2S mode and a sampling rate of 44.1 KHz
  - The first two time slots 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
  - The SSI2 transmitter watermark level is set to support SDMA transfers during audio playback
- The stereo codec is also configured for I^2S mode using a 44.1 KHz sample rate in master mode
- The Digital Audio MUX is configured to connect internal port 2 (which is assigned to SSI2) with one external port, which is used to communicate with the Stereo DAC. At the same time, the appropriate IOMUX pins are configured so that the Audio MUX external port signals can be routed off-chip to the Stereo Codec. The external port 4 is used to connect the Stereo Codec on the i.MX35 System.
- The SDMA channel supports 16-bit data transfers between the application memory buffers and the SSI2 TX FIFO0. The SSI2 TX FIFO0 is pre-filled 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 SSI2 issues a new DMA request when the transmitter FIFO0 level reaches the empty watermark level. The SDMA controller then refills FIFO0 using data from the DMA buffers, until the DMA buffer is empty.
- An interrupt is generated when a DMA buffer is empty and this interrupt is handled by the audio driver. The audio driver refills the DMA buffer and returns it to the SDMA controller for processing.
- Due to the double-buffering scheme, the SDMA controller simply uses the other DMA buffer to continue refilling the SS12 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 is no more data available to refill the empty DMA buffers. Therefore, the output DMA channel is disabled when both output DMA buffers are empty and there is no additional data available to refill them.
- The 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, gate SSI2 is disabled and clocked if receiver is not working.

4.3.3 Audio Recording

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

- As SSI2 is used in both playback and recording path, the audio recording shares the SSI configuration with playback configuration.
- The SDMA channel is fully configured to support 16-bit data transfers between the application memory buffers and the SSI2 RX FIFO0.
- The SSI2 receiver is enabled and ready to receive data from the stereo codec.

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

- The SSI2 issues a new DMA request whenever the receive FIFO0 level reaches the full watermark level. The SDMA controller then transfers the data from the receiver FIFO0 to an input DMA buffer until the DMA buffer is full.
- The SDMA controller generates an interrupt that is handled by the audio driver. The audio driver is responsible for copying the data from the full input DMA buffer into application-supplied buffers and then returning the empty input DMA buffer back to the SDMA controller. Any data which cannot be transferred to an application-supplied buffer (for example, due to insufficient space) is simply discarded.
- Since a double-buffering scheme is being used, the SDMA controller simply uses the other DMA buffer to continue recording the data from the SSI2 receiver FIFO0 while the previous DMA buffer is being copied to application-supplied buffers.

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

- Terminate the recording process by having the application close the audio input stream. At this point, disable audio components that were used for recording to minimize power consumption.
- Disable and clock gate SSI2, if transmitter is not working.
- Disable the input DMA channel to completely terminate the audio recording operation.

4.3.4 Required SoC Peripherals

Table 4-2 shows the SoC hardware components required by the audio driver.

Table 4-2. Required SoC Peripherals

Component	Use
SSI2	Playback and recording
Digital Audio MUX	Connects the SSI2 to the IO MUX to access off-chip peripherals
IO MUX Pins	Connects the Digital Audio MUX external port to the external stereo codec
SDMA Controller	Manages the DMA channels that are used for playback and recording

4.3.5 Conflicts with SoC Peripherals

No conflicts.

4.3.6 Conflicts with Board Peripherals

4.3.6.1 i.MX35 3-Stack Peripherals Conflicts

The TVIN and FM modules use SGTL5000 loopback mode to support audio output through headphone. Thus SGTL5000 stereo audio driver stops working when either TVIN or FM application is running.

4.3.7 Known Issues

4.3.7.1 i.MX35 Known Issues

If both the SGTL5000 stereo audio driver and S/PDIF driver occurs, the default audio device might be SPDIF. The default audio device may be chosen by AudioRouting application.

4.4 Software Operation

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

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

4.4.1 Audio Playback

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

4.4.2 Audio Recording

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

NOTE

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

4.4.3.1 i.MX35 Audio Driver Configuration Options

Table 4-3 gives the compile-time configuration options of i.MX35 stereo audio driver.

Table 4-3. i.MX35 Audio Driver Configuration Options (oemsettings.h)

Configuration Setting Name	Description
INCHANNELS	Defines the number of input/recording channels that are available. Can be set to either 1 or 2. Default is 2.
OUTCHANNELS	Defines the number of output/playback channels that are available. Can be set to either 1 or 2. Default is 2.
HWSAMPLE	A typedef that defines the size of each audio data word. This must match the BITSPERSAMPLE and AUDIO_SAMPLE_MAX/AUDIO_SAMPLE_MIN values. Default is 16.
USE_MIX_SATURATE	Enable a check in the software mixer code to guard against saturation. Default is 1.
AUDIO_SAMPLE_MAX and AUDIO_SAMPLE_MIN	The valid range of each audio data word. Values that are outside of this range is clipped to the max/min value by the saturation protection code if USE_MIX_SATURATE is set to 1. Default is 32767 and -32768.
ENABLE_MIDI	If set to 1, MIDI code is included in the driver (~4k).
USE_OS_MIXER	If set to 1, the driver does not do any internal mixing and relies on the OS mixer.
BITSPERSAMPLE	The number of data bits per audio sample. If set to 16, support 16bit sample; If set to 24, support 24bit sample (in sgtl5000codec.h)

4.4.4 DMA Support

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

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

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

The DMA data buffers can be allocated from either internal memory (which is provided by on-chip internal RAM) or external memory (which is provided by off-chip external DRAM). Table 4-4 describes the issues and considerations for the type of memory to use 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) • Less power is required to access the internal RAM • The total size of the internal memory region is limited • The limited amount of internal memory may have to be shared by multiple device drivers • The entire internal memory region must be manually managed with predefined addressed ranges being reserved for each specific use
External	• The total size of the external memory is typically much greater than the size of the internal memory. This provides much greater flexibility in selecting the size of the DMA data buffers. • There is typically no need to worry about the possible impact and memory requirements of any other device driver. • Memory allocation is handled using the standard Windows Embedded CE 6.0 system calls • The external memory cannot be placed into a low power mode while the DMA is active

Table 4-5 describes how to configure the build so that the audio driver allocates 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, the choice is made automatically through the use of the CSPDDK APIs, specifically DDKSdmaAllocChain(). See Chapter 10, "Chip Support Package Driver Development Kit (CSPDDK)," for additional information about the DDKSdmaAllocChain() API.

Table 4-5. Configuration Options 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. Set BSP_AUDIO_DMA_BUF_SIZE to the total size (in bytes) for all DMA data buffers that is allocated.
External	Make sure that the BSP_AUDIO_DMA_BUF_ADDR macro is commented out in bsp_cfg.h

4.4.4.1 i.MX35 Audio DMA Buffer Use

The i.MX35 audio driver supports both playback and recording. Playback function always uses internal memory as DMA buffer, while recording function allocates DMA buffer from external memory.

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

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

4.4.5.1 PowerUp

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

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

4.4.5.2 PowerDown

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

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

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

4.4.5.3 IOCTL_POWER_SET

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

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

This registry entry is required for proper power management functionality.

4.4.6 Audio Driver Registry Settings

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

4.4.6.1 i.MX35 Audio Driver Registry Settings

The following registry keys are required in order for the Device Manager to properly load the i.MX35 audio device driver during the device 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 and all audio functions are disabled.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio]
"Prefix"="WAV"
    "Dll"="wavedev2_sgtl5000.dll"
    "Index"=dword:1
    "Order"=dword:4
    "Priority256"=dword:95
    "IClass"=multi_sz:"{A32942B7-920C-486b-B0E6-92A702A99B35}",
         "{37168569-61C4-45fd-BD54-9442C7DBA46F}"

; Override wave API load order to follow audio driver
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\WAPIMAN]
   "Order"=dword:5
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\WAPIMAN_ACM]
   "Order"=dword:5
```

4.5 Unit Test

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

4.5.1 Unit Test Hardware

Table 4-6 identifies the hardware needed to run the unit tests.

Table 4-6. Hardware Requirements

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

4.5.2 Unit Test Software

Table 4-7 lists the software required to run the unit tests.

Table 4-7. Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data

Table 4-7. Software Requirements (continued)

Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
wavetest.dll	Test .dll file

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 is included with the CETK files in the following location:

[Drive]:\Program Files\Microsoft Platform Builder\6.00\cepb\wcetk\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, use the CETK interface in the Platform Builder. If the full-duplex operation is not supported, the command line is:

```
tux -o -d wavetest -c "-e"
```

NOTE

Some test cases may fail with default parameters or settings, such as

- Playback Mixing Test (Test ID 6000). The thread number should be limited or the case fails. This is a known issue from MSFT. Using the command : tux -o -d wavetest -x 6000 -c "-t 5"
- Capture Mixing Test (Test ID 6001). The thread number should be limited or the case fails. This is a known issue from MSFT. Using the command : tux -o -d wavetest -x 6001 -c "-t 5"
- Verify Device ID Validation Test (Test ID 8001). If SPDIF or ESAI driver is in the system, the case may fail as this case is not well supported by these drivers. Test with the image without SPDIF and ESAI driver.

For detailed information about the audio driver tests, see the Platform Builder 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, various system-level tests that involve the use of the audio driver can be performed. The following sections describe how to test the audio driver without using the CETK.

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 touch panel calibration screen. At this point, the audio driver should already have successfully loaded and the tune should be heard if a headset is attached to the stereo output jack.

4.6.2 Confirming Touchpanel Taps and Keypad Key Presses

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

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 is to include the software codecs in the OS image that may be needed to decode the media file. The Media Player includes controls for pausing, resuming, and stopping playback, and advancing playback to a specific point. Volume and muting controls are also provided.

4.6.4 Using the SDK Sample Audio Applications for Testing

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

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

4.7 Mixer Driver Tests

The Stereo Codec SGTL5000 on i.MX35 3-Stack supports loopback mode and input mux between Microphone and Line In. The loopback mode is used by the TVIN and FM applications. The selection of these modes is implemented as mixer interface within audio driver. An example application (unit test program) for the mixer interface use can be found in WINCE600\SUPPORT\TEST\MIXER.

4.8 Audio Driver API Reference

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

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

4.9 Audio Driver Troubleshooting Guide

The following sections describe techniques to identify and fix the most common problems involving the audio driver.

4.9.1 Checking Build-Time Configuration Options

Compile- or link-time errors are probably due to incorrect or invalid configuration settings defined in `hwctxt.h` or `hwctxt.cpp`. See Section 4.4.3.1, "i.MX35 Audio Driver Configuration Options for information about the device driver build configuration options. Follow the build procedure documented in the Release Notes to compile and link the audio driver. Confirm that the required Platform Builder catalog items are included in the OS design. See Table 4-1 for a list of the required and recommended audio driver-related catalog items.

4.9.2 Media Player Application Not Found

Make sure that the Media Player catalog item is included in the OS design. The Media Player application is not included in the final system image if the catalog item is not selected. For information about these items, see the Platform Builder Help topic:

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

4.9.3 Media Player Fails to Load and Play an Audio File

This problem is typically caused by failing to include the appropriate software codec that is required to handle the audio file format. See the list of recommended audio driver catalog items in Table 4-1 and make sure that support for the desired audio file format is included.

Chapter 5
Backlight Driver

The backlight driver uses the hardware provided by the display module on the device 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 is power-manageable, and it meets the requirements of a power-manageable device by implementing the required IOCTLs. The backlight driver uses its own defined timer to set the backlight power states.

5.1 Backlight Driver Summary

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

Table 5-1. Backlight Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\BACKLIGHT
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\\<*Target Platform*>\SRC\DRIVERS\BACKLIKGHT
Driver DLL	backlight.dll
SDK Library	N\A
Catalog Item	Third Party > BSP > Freescale i.MX35 3DS PDK1_7: ARMV4I > Device Drivers > Smart Backlight Control
SYSGEN Dependency	SYSGEN_BATTERY=1
BSP Environment Variables	BSP_NOBACKLIGHT= BSP_BACKLIGHT_IPU=1

5.2 Supported Functionality

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

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

5.3 Hardware Operation

5.3.1 i.MX35-3DS Hardware Operation

The hardware consists of a PWM implemented by the IPU. This PWM is usually dedicated to the contrast but on the iMX35-3DS it used to drive the backlight. It can be configured by writing the SDC_CUR_BLINK_PWM_CTRL register.

5.4 Software Operation

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

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

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

5.4.1 Backlight Driver Registry Settings

5.4.1.1 i.MX35-3DS Backlight Driver Registry Setting

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

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

5.4.2 Power Management

The backlight driver consumes power primarily through the operation of the Liquid Crystal Display Panel backlight. To facilitate management of this module, the backlight driver implements the power management I/O Control (IOCTL) code IOCTL_POWER_SET.

5.4.2.1 PowerUp

This function is not implemented for the backlight driver.

5.4.2.2 PowerDown

This function is not implemented for the backlight driver.

5.4.2.3 IOCTL_POWER_SET

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

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

5.5 Unit Test

The backlight driver is tested by the application test.

5.5.1 Unit Test Hardware

5.5.1.1 i.MX35-3DS Unit Test Hardware

Table 5-2 lists the required hardware to run backlight application test.

Table 5-2. Hardware Requirements

Requirement	Description
CHUNGHWA 7" WVGA Display With Touch Screen (CLAA070VC01)	Display panel required for display of graphics data.

5.5.2 Unit Test Software

Table 5-3 lists the required software to run the backlight application test.

Table 5-3. Software Requirements

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

5.5.3 Running the Backlight Application Test

Table 5-4 lists the backlight application test.

Table 5-4. Backlight Application Test

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

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

Boot support from SD/MMC includes the following components:

- Xloader (XLDR)
- EBOOT (may also be referred to as bootloader in this document)
- Storage for OS binary image (NK)

Xloader, which executes from Internal RAM (IRAM), is a initial loader whose responsibility is to copy the bootloader from the SD/MMC memory to external RAM (SDRAM) and then pass the execution to EBOOT.

NOTE

XLDR and EBOOT only support boot from ESDHC1. Boot ROM supports booting from all ESDHC ports; therefore, XLDR and EBOOT can be extended to boot from other ports. SD/MMC boot requires a card that is at least 64 Mbytes.

6.1 Boot from SD/MMC Summary

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

Table 6-1. Boot from SD/MMC Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX35-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\\<*Target Platform*>\SRC\BOOTLOADER ..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\BOOT\FMD\SDMMC
Driver DLL	N/A
SDK Library	N/A
Catalog Item(s)	N/A
SYSGEN Dependency	N/A
BSP Environment Variable(s)	N/A

6.2 Supported Functionality

The boot support from SD/MMC includes:

1. Boot from low or high capacity SD/MMC card at least 64 Mbytes in size on ESDHC1
2. Storing bootloader and SD/MMC Xloader images to SD/MMC flash
3. Storing OS images to SD/MMC flash
4. Loading OS image from SD/MMC flash to RAM
5. File system on bootable SD/MMC card
6. Internal boot (BMOD = 00) from SD/MMC
7. eSD2.1 and eMMC 4.3 boot from boot partition if boot partition can be configured to be at least 64 Mbytes in size; otherwise, boot from user partition on these devices is supported

6.3 Hardware Operation

6.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts for eSDHC1 with other on-chip peripherals.

6.4 Software Operation

Only ESDHC1 is supported by XLDR and EBOOT as the boot port.

On startup, when booting from SD/MMC, the boot ROM is responsible for initializing and bringing the SD/MMC memory to a proper working state. It configures the memory only in 1-bit mode and brings it to transfer state where read/write operation can be done from the memory. The boot ROM then copies the entire XLDR from the SD/MMC memory to internal RAM and passes the control to the Xloader. The Xloader initializes the SDRAM, copies the bootloader from a predefined memory location of the SD/MMC memory to SDRAM, and passes control to the bootloader which in turn brings up the OS. Xloader reads data in 1-bit mode only. It checks the addressing mode for the card used by the boot ROM (which is stored in the IRAM at a fixed location), and decides whether to address the card in sector mode (high capacity) or byte mode (low capacity).

SD/MMC boot does not use any form of DMA. Whether it is the boot ROM, XLDR, or EBOOT, all the components involved in the boot process utilize the PIO mode. SD/MMC boot supports both secure (internal boot mode is required for enabling security checks) as well as non-secure boot.

To store and load a boot image to SD/MMC cards using EBOOT, the SDFMD (SD Flash Media Driver) library is used which exposes functions to perform erase, read and write operations on SD/MMC flash. The FMD layer provides support for all types of cards (high as well as low capacity SD/MMC cards). It also supports 1 and 4-bit modes for data transfer that is configurable through the BSP_MMC4BitSupported() function found in the BSP portion of EBOOT.

For preparing and downloading the SD/MMC bootloader and for usage of the SD/MMC bootloader, refer to the *BSP User's Guide*.

6.4.1 Card Memory Layout

SD cards that do not meet the v2.1 spec and MMC cards that do not meet the v4.3 spec have only one physical partition. To allow storage of boot images as well as file system on these card, EBOOT can add a partition table (MBR) to the card that reserves the initial 64 Mbytes for boot images (XLDR, EBOOT, NK) and the remaining portion of the card for the file system. The card must then be inserted into a PC to format the file system partition. Subsequently, it can be used as a boot device as well as to store and load user files once the OS has loaded. Refer to the *BSP User's Guide* for details.

eSD v2.1 and eMMC v4.3 both provide the capability of having more than one physical partition, thus eliminating the need to put an MBR on the device. Reading, writing, and erasing one partition has no effect on the other partitions. Starting with TO1, the ROM is able to boot from the boot partition on these devices. During boot, the ROM code selects the boot partition #1 on the eSD v2.1 device and either boot partition #1 or #2 on the eMMC v4.3 device (depending on which partition is enabled in the EXT_CSD register), and subsequently reads out the data that is flashed to the boot partition and executes it. EBOOT provides menu options to create and enable/disable boot partitions on both devices using the MMC and SD Utilities sub-menu. Refer to the *BSP User's Guide* for details.

Before the NK OS image is launched, EBOOT disables the boot partition, and the user partition, where the file system can be stored, is activated. As soon as system is reset, the ROM code re-enables the boot partition and reads out and executes the boot images. The Windows CE 6 R2 SDBus2 Driver, although capable of supporting high capacity SD cards, is not capable of supporting high capacity MMC cards. Therefore, high capacity eMMC v4.3 devices are not usable on Windows CE 6 for file system storage.

6.4.1.1 i.MX35 Card Memory Layout

Figure 6-1 shows the card memory layout for the i.MX35.

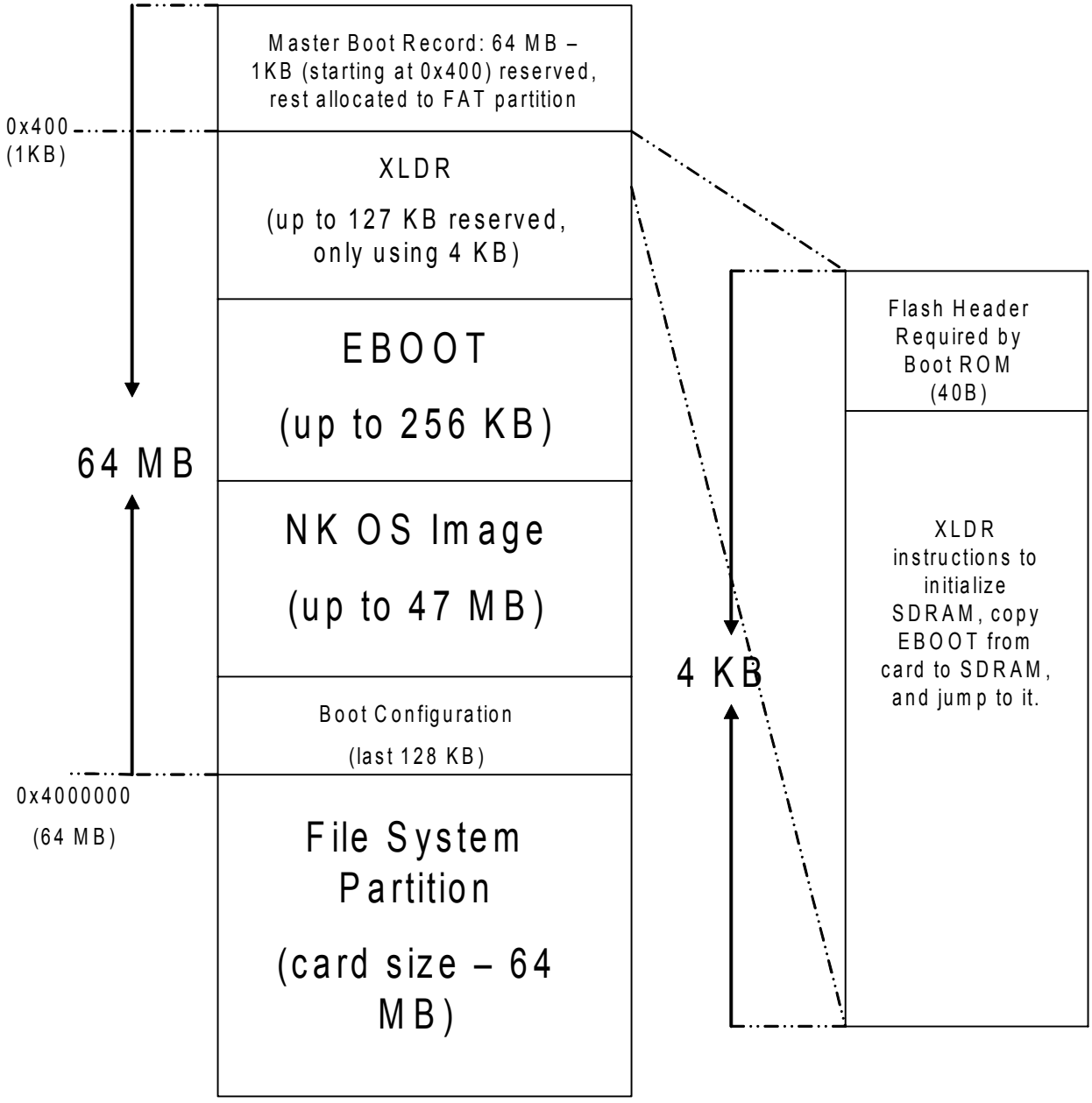

Figure 6-1. Card Memory Layout

A Master Boot Record (MBR) is placed by EBOOT (this functionality can be accessed using the EBOOT menu) at sector 0 of the card to reserve the first 96 Mbytes of the card for boot images, and allocate the remaining portion to the file system. The XLDR is saved at 0x400 (1 Kbyte) offset, which is sector 2 in the card. The Boot ROM calculates the entry point of the XLDR from the flash header structure found in the XLDR.

The MBR is only required on cards that are older than eSD v2.1 and eMMC v4.3 because these newer devices can have multiple physical partitions. On these devices, the first 96 Mbytes shown above are flashed on a separate boot partition (without an MBR at sector 0), and the file system partition referenced above is another separate physical partition, which should only be active while OS is running.

Chapter 7
Camera Driver

The camera driver interfaces the low level camera sensor to the Windows CE camera subsystem.

7.1 Camera Driver Summary

The camera driver is based on the Windows CE 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. At the lower layer, the camera driver performs several tasks:

- Communicating with and configuring the camera device or other input device through the I^2C interface
- Interfacing with the Image Processing Unit (IPU) to perform pre-processing tasks on captured images
- Configuring the IPU Synchronous Display Controller (SDC) for display of video preview data

For i.MX35 BSP, this module is compatible with the camera sensor OV2640. And it is also supported TVIN device ADV7180.

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

Table 7-1. Camera Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\IPU\CAMERA
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\CAMERA
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\CAMERA
Driver DLL	camera.dll
SDK Library	N/A
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*>*:ARMV4I* > Device Drivers > Camera

Table 7-1. Camera Driver Summary

SYSGEN Dependency	SYSGEN_IMAGING_BMP_ENCODE SYSGEN_IMAGING_JPG_ENCODE SYSGEN_IMAGING_BMP_DECODE SYSGEN_IMAGING_JPG_DECODE SYSGEN_DSHOW_DISPLAY SYSGEN_DSHOW_CAPTURE SYSGEN_DSHOW_DMO SYSGEN_DSHOW_VIDREND
BSP Environment Variables	BSP_NOCAMERATVIN= BSP_CAMERA=1 for camera used BSP_TVIN=1 for TVIN used

7.2 Supported Functionality

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

1. Supports the Windows CE Camera Device Driver Interface
2. Supports Preview, Capture, and Still pins for camera application
3. Supports Preview pin for TVIN application.
4. Supports the OV2640 camera sensors for camera used for i.MX35.
5. Supports the analog device ADV7180 for TVIN used for i.MX35.

7.3 Hardware Operation

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

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

7.3.1 Conflicts with Other Peripherals and Catalog Items

7.3.1.1 Conflicts with SoC Peripherals

For i.MX35 platform, the camera sensor shares the CSI interface with the TVIN device and they use the same camera driver. Only one module can be used on the i.MX35 3-Stack board at one time.

7.3.2 Conflicts with 3-Stack Peripherals

No conflicts.

7.4 Software Operation

The development concepts for camera driver is described in the Windows CE 6.0 Help Documentation section under the topic

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

7.4.1 Communicating with the Camera

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

7.4.1.1 Using the Windows CE Video Camera Device Driver Interface

The Windows CE Video Camera Device Driver Interface provides basic support for video and still image capture devices. For information about using camera APIs, see the Windows Embedded CE 6.0 Help topic:

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

7.4.1.2 Using DirectShow for Video Capture

DirectShow provides support in its architecture for the creation of filter graphs for video capture. For information about using DirectShow for video capture, see the Windows Embedded CE 6.0 Help:

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

7.4.2 Registry Settings

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

This section describes the registry keys used to select the camera sensor used on the SoC.

7.4.2.1 i.MX35 Registry Settings

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

```
#if (defined BSP_CAMERA ||  defined BSP_TVIN)
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Camera]
   "Prefix"="CAM"
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:1
   IF BSP_CAMERA
```

```
        "CameraId"=dword:3
   ELSE
        "CameraId"=dword:4
   ENDIF ;BSP_CAMERA
      ;See csi.h,In WinCE6.0,default is 2. CameraInUse: 0=iMagic8803, 1=iMagic8201,
2=Magna521DA, 3=ov2640, 4=Adv7180
   "IClass"=multi_sz:
"{CB998A05-122C-4166-846A-933E4D7E3C86}","{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

The CameraId registry key identifies the available camera sensor modules. Valid values:

- 0 to indicate that the camera sensor in use is the iMagic IM8803
- 1 to indicate that the camera sensor in use is the iMagic IM8201.
- 2 to indicate that the camera sensor in use is the Magna521DA.
- 3 to indicate that the camera sensor in use is the OV2640.
- 4 to indicate that the TVIN in use is the ADV7180.

```
[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}"
#endif ; (defined BSP_CAMERA ||  defined BSP_TVIN)
```

7.5 Power Management

The camera driver consumes power primarily through the operation of various IPU sub-modules, such as the CSI (which synchronizes and receives image data from the camera sensor) and the IC (which performs pre-processing operations on captured image data). The CSI and IC modules are enabled when the camera or TVIN device is set to a running state.

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

7.5.1 Power Up

This function is not implemented for the camera driver.

7.5.2 Power Down

This function is not implemented for the camera driver.

7.5.3 IOCTL_POWER_SET

7.5.3.1 i.MX35 IOCTL_POWER_SET Support

The camera driver implements the IOCTL_POWER_SET IOCTL API with support for the D0 (Full on) and D4 (Off) power states.

These states are handled in the following manner:

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

7.6 Unit Test

Because the Camera Driver API was introduced with Windows Embedded CE 6.0, there are CETK tests written and provided by Microsoft.

The Camera CETK tests include:

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

Additionally, for Windows Embedded CE 6.0, a Camera Application may be used to preview and capture images. A TVIN Application is only used to preview images.

7.6.1 Unit Test Hardware

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

Table 7-2. Hardware Requirements

Requirement	Description
Camera Sensor or TVIN device	OV2640 CMOS camera sensor

The OV2640 camera sensor is used to run the Windows Embedded CE 6.0 Camera CETK test and the camera application. The TVIN device ADV7180 is used to run the Windows Embedded CE 6.0 Camera CETK test (only need test preview parts) and the TVIN application.

7.6.2 Unit Test Software

7.6.2.1 CETK Test

Table 7-3 lists the required software to run the Camera Test.

Table 7-3. Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
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 options to track and output information about media samples
camera.dll	Driver.dll file

The configuration file `capconfig.ini` is required for `CameraPerfTests.dll`.

7.6.2.2 Custom Camera and TVIN Application Test

The `camapp.exe` executable file is needed to run the custom camera application. The `tvinapp.exe` executable file is needed to run the custom TVIN application.

7.6.2.3 Camera and TVIN Application Test

No additional actions are required to include the Windows CE 6.0 Camera/TVIN Application in an OS image beyond the required registry keys.

7.6.3 Building the Unit Tests

7.6.3.1 CETK Test

The camera tests come pre-built as part of the CETK so no steps are required to build these tests. These test files 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

7.6.3.2 Custom Camera and TVIN Application Test

To build the custom Camera/TVIN application, complete the following steps.

Build an OS image for the desired Camera/TVIN configuration:

1. Add a new folder named `APP` under the folder ..\PLATFORM\<Target Platform>\SRC
2. Create an empty dirs file under the folder ..\PLATFORM\<Target Platform>\SRC\APP
3. Copy the folder of `CAMAPP/TVIN` under the folder `APP`
4. Select the Solution Explorer of the Platform Builder Workspace window
5. Expand Platform > <Target Platform> > Src > App > CAMAPP`/TVIN`
6. Right-click on the CAMAPP/TVIN folder and select `Rebuild`

The CAMAPP/TVIN execution file (`camapp.exe/tvinapp.exe`) is created in the `obj\release` or `obj\debug` folder under the `CAMAPP/TVIN` folder. The camapp.exe/tvinapp.exe file is copied to the workspace release directory.

Complete the following steps to build the custom Camera/TVIN application:

1. Select the `Solution Explorer` of the Platform Builder Workspace window
2. Select `Subprojects` in `Solution Explorer`
3. Right-click `Subprojects` and select `Add Existing Subproject`
4. Add the CAMAPP/TVIN project by selecting the sources file located in \WINCE600\SUPPORT\APP\CAMAPP folder
5. Right-click on the CAMAPP/TVIN project and select `Rebuild`

The CAMAPP/TVIN execution file (camapp.exe/tvinapp.exe) is created in the workspace release directory.

7.6.4 Running the Unit Tests

7.6.4.1 Running the Camera Unit Tests

7.6.4.1.1 Running the Camera CETK Test

For detailed information about the tests in this section, see the Windows Embedded CE 6.0 Help topic:

Windows Embedded CE Test Kit > CETK Tests and Test Tools > CECETK Tests > Camera Tests

Use this command line to run the Camera and DirectShow integration test:

```
tux –o –d CameraGraphTests.dll
```

Use this command line to run the Camera Driver Data Structure Verification test:

```
tux –o –d CamTestProperties.dll
```

Use this command line to run the Camera Driver I/O test:

```
tux –o –d CamIOTests.dll
```

Use this command line to run the Camera Performance test:

```
tux -o -d cameraperftests.dll -c "-p \release\capresults.csv -c
\release\capconfig.ini"
```

NOTE

The Camera Performance test requires the configuration file `capconfig.ini` which specifies what to test, by 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
```

For i.MX35 Case #508 and #510 may fail for CameraGraphTests.dll.

7.6.4.1.2 Running the Custom Camera Application Test

The following command executes the Custom Camera Application: `camapp.exe`

7.6.4.2 Running the TVIN Unit Tests

7.6.4.2.1 Running the TVIN CETK Test

TVIN only supports preview, so only test related preview CETK. Use this command line to run the Camera and DirectShow integration test:

```
tux –o –d CameraGraphTests.dll -x !400-607
```

Use this command line to run the Camera Driver Data Structure Verification test:

```
tux –o –d CamTestProperties.dll
```

Use this command line to run the Camera Driver I/O test:

```
tux –o –d CamIOTests.dll
```

Use this command line to run the Camera Performance test:

```
tux -o -d cameraperftests.dll -c "-p \release\capresults.csv -c
\release\capconfig.ini" -x !300-401
```

NOTE

TVIN CETK does not support these cases related to Capture Pin and Still Pin.

7.6.4.2.2 Running the Custom TVIN Application Test

The following command executes the Custom TVIN Application: `tvinapp.exe`

7.7 Camera Driver API Reference

For the camera driver API reference, see the Windows Embedded CE 6.0 documentation. There is one additional custom API provided to allow applications to enable direct display of video preview data. For reference information on basic camera driver functions, methods, and structures, see the Windows Embedded CE 6.0 Help:

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

Chapter 8
Configurable Serial Peripheral Interface (CSPI) Driver

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

8.1 CSPI Driver Summary

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

Table 8-1. CSPI Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\CSPIBUSV2
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\CSPIBUS
Platform Driver Path	..\PLATFORM\<*Target Platform*>\DRIVERS\CSPIBUS
Import Library	cspisdk.lib
Driver DLL	cspi.dll
Catalog Item	Third Party > BSP > Freescale <*TGTPLAT*> > Device Drivers > CSPI Bus
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_CSPIBUS1=1 or BSP_CSPIBUS2=1

8.2 Supported Functionality

The CSPI driver supports the following features:

1. Supports the CSPI master mode of operation
2. Supports CSPI configurable bus feature
3. Supports CSPI multiple channel method
4. Supports configurable access method of polling method and interrupt method
5. Supports DMA exchange mode for 32-bit interface SPI access
6. Supports buffering exchange for asynchronous SPI access
7. Supports stream interface
8. Supports two power management modes, full on and full off

8.2.1 Conflicts with Other Peripherals and Catalog Items

8.2.1.1 Conflicts with SoC Peripherals

The i.MX35 platform contains two CSPI modules, but only the CSPI1 module is used on the i.MX35 3DS board. No pad is provided to CSPI2 module.

8.2.2 Conflicts with 3-Stack Peripherals

No conflicts

8.3 Software Operation

8.3.1 Registry Settings

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

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

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

8.3.2 Communicating with the CSPI

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

8.3.3 Creating a Handle to the CSPI

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

To open a handle to the CSPI:

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

   For example, specify SPI1: as the CSPI port
2. Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to an CSPI port are supported by the driver.
3. Specify OPEN_EXISTING in the *dwCreationDisposition* parameter. This flag is required.
4. Specify FILE_FLAG_RANDOM_ACCESS in the *dwFlagsAndAttributes* parameter.

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

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

8.3.4 Data Transfer Operations

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

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

To perform an CSPI transfer:

1. Create a CSPI_XCH_PKT object and initialize the fields of the packet as follows:
   a) Initialize a CSPI_BUSCONFIG datum to specify the bus parameters as CHANNEL SELECT, DATA RATE, BURST LENGTH, SSPOL, SSCTL, POL, PHA, DRCTL, and specify the method parameters for use/not use DMA, use/not use POLLING.
   b) Set the *pTxBuf* field to the user buffer which the data is written.
   c) Set the *pRxBuf* field to the user buffer which receives data, if does not care of the Rx data, set the field to NULL.
   d) Set the *xchCnt* field, for the 1-8 bit XCH, the *xchCnt* = *bytes*, for the 9-16 bit XCH, the *xchCnt* = *words*, for the 17-32 bit XCH, the *xchCnt* = *dwords*.
   e) If give the *xchEvent* parameter, also give the *xchEventlength* and include the tail Zero character. Otherwise, set *xchEvent* to NULL, and *xchEventlength* to 0. When use *xchEvent* the XCH data is queued inside driver.
2. Set the *hDevice* parameter to the previously acquired CSPI port handle.
3. Set the *dwIoControlCode* to the SPI_IOCTL_EXCHANGE IOCTL code.

4. Set the *lpInBuffer* to point to the CSPI_XCH_PKT object created in step 1. Set *nInBufferSize* to the size of that packet object.
5. Set *lpOutBuffer*, *lpBytesReturned*, and *lpOverlapped* to NULL. Set *nOutBufferSize* to 0.

The following code example demonstrates how to perform a XCH transfer.

```
CSPI_BUSCONFIG_T buscnfg =
{
    0,                                  //use channel 0
    16000000,                           //XCH speed 16M
    32,                                 //data rate = 32bit
    FALSE,                              // SSCTL: Only one SPI burst will be transmitted.
    TRUE,                               // SSPOL: Active HIGH
    FALSE,                              // POL: Active high polarity
    FALSE,                              // PHA: Phase 0 operation
    0,                                  // DRCTL: Don't care SPI_RDY
    FALSE,                              //Don't use DMA
    FALSE                               //Don't polling
};

DWORD TxData[11];
DWORD RxData[11];

CSPI_XCH_PKT_T xchPkt =
{
    &buscnfg,
    TxData,
    RxData,
    11,                                 // XCH to target SPI device 11 times
    NULL,
    0
};                                      // optional asynchronous event, recommended
hEvent = CreateEvent(0, FALSE, FALSE, L"RX_EVENT");
xchpkt.xchEvent = L"RX_EVENT";
xchpkt.xchEventLength = sizeof(L"RX_EVENT");
                                        // Transfer data via CSPI
DeviceIoControl(hCSPI,                  // file handle to the driver
 CSPI_IOCTL_EXCHANGE,                   // I/O control code
                 (PBYTE) &xchPkt,       // in buffer
                 sizeof(xchPkt),        // in buffer size
                 NULL,                  // out buffer
                 0,                     // out buffer size
                 NULL,                  // number of bytes returned
                 NULL);                 // ignored (=NULL)
```

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

```
CSPIExchange(hCSPI, &xchPkt);                   // optional
WaitForSingleObject(hEvent, INFINITE);          // Code for dealing received DATA
```

8.3.5 Closing the Handle to the CSPI

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

8.3.6 Power Management

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

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

8.3.6.1 PowerUp

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

8.3.6.2 PowerDown

This function is not implemented for the CSPI driver.

8.3.6.3 IOCTL_POWER_SET

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

8.4 Restrictions

The following restrictions apply to the DMA XCH:

- The DMA XCH in CSPI only supports 32-bit data size that is DWORD aligned
- The DMA XCH size is restricted to maximum DMA buffer size, up to 5 Kbytes

8.5 Unit Test

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

8.5.1 Building the Unit Tests

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

1. Within the Platform Builder, choose **Build OS** > **Open Release Directory**.
   A DOS prompt is displayed.
2. Change to the CSPI Test directory: `\WINCE600\SUPPORT`_PDK1_7`\TEST\CSPI`
3. Enter **set WINCEREL=1** on the command prompt and press return.

This copies the EXE to the flat release directory.

4. Input **build -c** to build CSPI test.

After the build completes, the CSPIAPP.EXE file is located in the $(_FLATRELEASEDIR) directory. To run the application within VS2005, go to the Target menu option and select the Run Programs menu option. This gives a list of applications that can be run on the OS. Select CSPIAPP.EXE from this list and click on Run to run this application.

8.6 CSPI Driver API Reference

8.6.1 CSPI Driver IOCTLS

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

8.6.1.1 CSPI_IOCTL_EXCHANGE

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

Parameters

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

8.6.1.2 CSPI_IOCTL_ENABLE_LOOPBACK

This **DeviceIoControl** request sets the LOOPBACK flag in the CSPI hardware.

8.6.1.3 CSPI_IOCTL_DISABLE_LOOPBACK

This **DeviceIoControl** request clears the LOOPBACK flag in the CSPI hardware.

8.6.2 CSPI Driver SDK Wrapper

8.6.2.1 CSPIOpenHandle

This function retrieves the CSPI device handle.

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

Parameters

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

8.6.2.2 CSPICloseHandle

This function closes a handle of the CSPI stream driver.

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

Parameters

hDev The CSPI device handle retrieved from CreateFile()

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

8.6.2.3 CSPIEnableLoopback

This function sets the CSPI controller in loopback mode to inspect if data value during XCH is correct.

```
BOOL CSPIEnbaleLoopback(
        HANDLE hDev
);
```

Parameters

hDev The CSPI device handle retrieved from CreateFile()

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

8.6.2.4 CSPIExchange

This function performs XCH operations.

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

Parameters

hDev The CSPI device handle retrieved from CreateFile()

pCspiXchPkt [in] Pointer to XCH packet with bus configuration parameters

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

8.6.3 CSPI Driver Structures

8.6.3.1 CSPI_BUSCONFIG_T

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

```
// CSPI bus configuration
typedef struct
{
    UINT8         chipselect;
    UINT32        freq;
    UINT8         bitcount;
    BOOL          sspol;
    BOOL          ssctl;
    BOOL          pol;
    BOOL          pha;
```

```
    UINT8         drctl;
    BOOL          usedma;
    BOOL          usepolling;
} CSPI_BUSCONFIG_T, *PCSPI_BUSCONFIG_T;
```

Table 8-2. CSPI_BUSCONFIG_T Structure Members

Member	Description
chipselect	Select XCH channel, range 0-3
freq	DATA band rate
bitcount	Define bits used in a single XCH, range 1-32
sspol	SPI SS Polarity Select. If FALSE, active low, if TURE, active high
ssctl	SPI SS Wave Form Select. This bit controls the output wave form of SS signal FALSE: Only one SPI bursts is transmitted TRUE: Negate SS between SPI bursts. Multiple SPI bursts is transmitted
pol	SPI Clock Polarity Control FALSE: Active high polarity (0 = Idle) TRUE: Active low polarity (1 = Idle)
pha	SPI Clock/Data Phase Control FALSE: Phase 0 operation TRUE: Phase 1 operation
drctl	DRCTL of CSPI XCH operation 00: Do not care SPI_RDY 01: Burst is triggered by failing edge of SPI_RDY 10: Burst is triggered by low level of SPI_RDY 11: RSV
usedma	If TRUE, uses DMA mode
usepolling	If TRUE, uses polling mode

8.6.3.2 CSPI_XCH_PKT_T

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

```
// CSPI exchange packet
typedef struct
{
    PCSPI_BUSCONFIG_T pBusCnfg;
    LPVOID pTxBuf;
    LPVOID pRxBuf;
    UINT32 xchCnt;
    LPWSTR xchEvent;
    UINT32 xchEventLength;
} CSPI_XCH_PKT_T, *PCSPI_XCH_PKT_T;
```

Table 8-3. CSPI_XCH_PKT_T Structure Members

Member	Description
pBusCnfg	A pointer to CSPI bus configuration object
pTxBuf	A pointer to Tx data buffer
pRxBuf	A pointer to Rx data buffer

Table 8-3. CSPI_XCH_PKT_T Structure Members

Member	Description
xchCnt	Amount of XCH operation to SPI device
xchEvent	Asynchronous access using the internal exchange queue
xchEventLength	Event name length including tailing Zero

Chapter 9
Controller Area Network (CAN) Driver

The CAN module provides the low level functionality of a CAN protocol according to the CAN 2.0B protocol spec. The CAN module only supports Message Buffer mode.

9.1 CAN Driver Summary

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

Table 9-1. CAN Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\CANBUS
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\CANBUS
Driver DLL	can.dll
SDK Library	cansdk.lib
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*> > Device Drivers > CAN Bus
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_CANBUS1=1 BSP_CANBUS2=1

9.2 Supported Functionality

The CAN driver enables the 3-Stack System to provide the following software and hardware support:

1. Supports the CAN communication protocol
2. Provides a stream interface driver implementing the programming interface defined in this document
3. Supports two power management modes, full on and full off

9.3 Hardware Operation

Refer to the chapter on CAN in the *Multimedia Applications Processor Reference Manual* for detailed operation and programming information.

9.3.1 Conflicts with Other Peripherals and Catalog Items

9.3.1.1 Conflicts with SoC Peripherals

No conflicts.

9.3.1.2 Conflicts with 3-Stack Peripherals

On the 3-Stack board, CAN1 pin conflicts with the USB high speed device. Do not enable the USB host device driver when CAN1 driver is enabled.

9.4 Software Operation

9.4.1 Communicating with the CAN

The CAN driver is a stream interface driver, and is thus accessed through the file system APIs. To communicate using the CAN, 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.

9.4.2 Creating a Handle to the CAN

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

To open a handle to the CAN:

1. Insert a colon after the CAN port for the first parameter, *lpFileName*. For example, specify CAN1: as the CAN port.
2. Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to an CAN 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 CAN1 port.

```
// Open the CAN port.
hCAN = CreateFile (CAN1_FID,                          // 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 CAN port, the port must be configured. When an application opens an CAN port, it uses the default configuration settings, which might not be suitable for the device at the other end of the connection.

9.4.3 Configuring the CAN

Configuring the CAN port for communications involves one main operation: setting the CAN for transmit or receiver mode. Before this action can be taken, a handle to the CAN port must already be opened. Each of these steps requires a call to the **DeviceIoControl** function. As parameters, the CAN port handle, appropriate IOCTL code, and other input and output parameters are required.

To configure an CAN port:

1. Set the *hDevice* parameter to the previously acquired CAN port handle.
2. Set the *dwIoControlCode* to the following IOCTL code: CAN_IOCTL_SET_CAN_MODE
3. Set the *lpInBuffer* to point to the variable to use for the CAN 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 CAN port.

```
// Set CAN mode
    DeviceIoControl(hCAN,                   // file handle to the driver
         CAN_IOCTL_SET_CAN_MODE,            // I/O control code
         &ChangedMode,                      // in buffer
         sizeof(DWORD)                      // 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, sdk may be used to simplify the code. The following code shows an example:

```
CANSetMode(HANDLE hCAN,DWORD index,CAN_MODE mode);
```

9.4.4 Data Transfer Operations

The CAN driver provides one command, Transfer, that facilitates performing both reads and writes through the CAN. The basic unit of data transfer in the CAN driver is the CAN_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 CAN_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 CAN.

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

To perform an CAN transfer:

1. Create an array of CAN_PACKET objects and initialize the fields of each packet as follows:
   a) Set the *byIndex* field to the message buffer index for exchange data, the maximun value is 64.
   b) Set the *byRW* field to CAN_RW_WRITE to specify that the CAN operation is a Write, or CAN_RW_READ to specify that the CAN operation is a Read.
   c) Set the *format* field to CAN_STANDARD to specify that the CAN frame format is a standard, or CAN_EXTENDED to specify that the CAN frame format is a extended.
   d) Set the *frame* field to CAN_DATA to specify that the CAN RTR format is a data, or CAN_REMOTE to specify that the CAN RTR frame format is a remote.
   e) Set the *ID* field to the message buffer ID for exchange data, for standard frame only supports 11 bit frame identification, extended frame can support 29 bit frame identification.
   f) Set the *wLen* field to size, in bytes, of the read or write buffer. This indicates the number of bytes to write or read.
   g) Set the *pbybuf* field to the read or write buffer.
   h) Set the *lpiResult* field to point to an integer that holds the return value from the write operation.
2. Set the *hDevice* parameter to the previously acquired CAN port handle.
3. Set the *dwIoControlCode* to the CAN_IOCTL_TRANSFER IOCTL code.
4. Set the *lpInBuffer* to point to the CAN_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 CAN_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.

```
CAN_PACKET cp = {0};
CAN_TRANSFER_BLOCK ctb = {0};

cp.byIndex=(DWORD)lpParameter;
cp.byRW=CAN_RW_READ;
cp.fromat=CAN_EXTENDED;
cp.frame =CAN_DATA;
cp.ID=0x1234456;
cp.wLen=8;
cp.pbyBuf=(PBYTE)data;
cp.lpiResult=&ret;
ctb.pCANPackets=&cp;
ctb.iNumPackets=1;

                                    // Transfer data via CAN
if (!DeviceIoControl(hCAN,          // file handle to the driver
        CAN_IOCTL_TRANSFER,         // I/O control code
        pCANTransferBlock,          // in buffer
        sizeof(CAN_TRANSFER_BLOCK), // in buffer size
        NULL,                       // out buffer
        0,                           // out buffer size
        NULL,                       // pointer to number of bytes returned
```

```
        NULL))                      // ignored (=NULL)
    {
        DEBUGMSG(ZONE_ERROR,
            (TEXT("%s: CAN_IOCTL_TRANSFER failed!\r\n"), __WFUNCTION__));
        return FALSE;
    }
```

As a substitute for the **DeviceIoControl** call above, the SDK function as following:

```
CANTransfer(g_hReader, &ctb);
```

9.4.5 Closing the Handle to the CAN

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

9.4.6 Power Management

9.4.6.1 PowerUp

This function is not implemented for the CAN driver.

9.4.6.2 PowerDown

This function is not implemented for the CAN driver.

9.4.6.3 IOCTL_POWER_CAPABILITIES

The power management capabilities are handled with the Power Manager through this IOCTL. The CA N module supports only two power states: D0 and D4.

9.4.6.4 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** in the CAN driver. Any request that is not D0 is changed to a D4 request and results in the system entering into suspend state, while for a value of D0 the system is resumed. For all platforms, the following registry entry must be defined:

```
"IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

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

9.4.7 CAN Registry Settings

The following registry keys are required to properly load the CAN1 and CAN2 module.

```
IF BSP_CANBUS1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\CAN1]
   "Prefix"="CAN"
   "Dll"="can.dll"
   "Index"=dword:1
   "Order"=dword:9
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"
ENDIF ; BSP_CANBUS1

IF BSP_CANBUS2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\CAN2]
   "Prefix"="CAN"
   "Dll"="can.dll"
   "Index"=dword:2
   "Order"=dword:9
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"
ENDIF ; BSP_CANBUS2
```

9.5 Unit Test

The CAN CETK test cases verify the functionality of the CAN driver with the CAN controller. The CAN driver can also be used to verify the functionality of the CAN driver.

9.5.1 Unit Test Hardware

The CANBUS1 controller should be connected to the CANBUS2 controller and loopback is tested between the two CANBUS controllers. The CANBUS controllers can be connected directly; you do not need to connect an external transceiver between the two controllers. The four CANBUS controller pins should be connected together directly.

9.5.2 Unit Test Software

Table 9-2 lists the required software to run the unit tests.

Table 9-2. Software Requirements

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

9.5.3 Building the Unit Tests

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

1. Within the Platform Builder, choose **Build OS** > **Open Release Directory**.

A DOS prompt is displayed.

2. Change to the CAN Tests directory: `\WINCE600\SUPPORT_PDK1_7\TEST\CANBUS\CANFunc`
3. Enter **set WINCEREL=1** on the command prompt and press return.
   This copies the file to the flat release directory.
4. Input **build -c** to build the CAN test.

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

9.5.4 Running the Unit Tests

The command line for running the tests is:

```
tux -o -n -d cantest
```

CAN tests do not contain any test specific command line options.

Chapter 10
Chip Support Package Driver Development Kit (CSPDDK)

The Chip Support Package Driver Development Kit (CSPDDK) provides an interface to access peripheral features and SOC configurations 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)

10.1 CSPDDK Driver Summary

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

Table 10-1. CSPDDK Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFROM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\CSPDDK
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\CSPDDK
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\CSPDDK
Driver DLL	cspddk.dll
SDK Library	N/A
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOCSPDDK=

10.2 Supported Functionality

The CSPDDK meets the following requirements:

1. Supports an interface that allows synchronized inter-process access to the following set of shared SoC resources:
   - GPIO (DDK_GPIO)
   - SDMA (DDK_SDMA)
   - IOMUX (DDK_IOMUX)

— CCM (DDK_CLK)

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

10.3 Hardware Operation

Refer to the *Applications Processor Reference Manual* for detailed operation and programming information.

10.3.1 Conflicts with Other Peripherals and Catalog Items

10.3.1.1 Conflicts with SoC Peripherals

Refer to the i.MX35 hardware specification document for possible conflicts.

10.3.1.2 Conflicts with Board Peripherals

No conflicts.

10.4 Software Operation

10.4.1 Communicating with the CSPDDK

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.

10.4.2 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 10-2 describes the available CSPDDK compile options.

Table 10-2. CSPDDK Compile Options

Compilation Variable	Header File	Description
IMAGE_WINCE_DDKSDMA_IRAM_PA_START	image_cfg.h	Physical starting address in internal RAM (IRAM) where the shared SDMA data structures are 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_CSPDDK_RAM_PA_START	image_cfg.h	Physical starting address in external RAM where the shared CSPDDK data structures are located. The DDK_CLK and DDK_SDMA uses space from this region. This address must correspond to the region reserved in config.bib.

Table 10-2. CSPDDK Compile Options (continued)

IMAGE_WINCE_CSPDDK_RAM_OFFSET	image_cfg.h	Offset in bytes from the base of external RAM for the shared CSPDDK data structures.
IMAGE_WINCE_CSPDDK_RAM_SIZE	image_cfg.h	Size in bytes of the external RAM reserved for CSPDDK data structures. This size must correspond to the region reserved in config.bib.
IMAGE_WINCE_DDKSDMA_RAM_PA_START	image_cfg.h	Physical starting address in external RAM where the shared DDK_SDMA data structures are located. This starting address must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions.
IMAGE_WINCE_DDKSDMA_RAM_SIZE	image_cfg.h	Size in bytes of the external RAM reserved for DDK_SDMA data structures. This size must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions.
IMAGE_WINCE_DDKCLK_RAM_PA_START	image_cfg.h	Physical starting address in external RAM where the shared DDK_CLK data structures are located. This starting address must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions.
IMAGE_WINCE_DDKCLK_RAM_SIZE	image_cfg.h	Size in bytes of the external RAM reserved for DDK_CLK data structures. This size must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions.
BSP_SDMA_MC0PTR	bsp_cfg.h	Starting address for the shared SDMA data structures. Can only be set to IMAGE_WINCE_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.

10.4.3 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.

10.4.4 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.

10.5 Unit Test

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

10.5.1 Unit Test Hardware

Table 10-3 lists the required hardware to run the unit tests.

Table 10-3. Hardware Requirements

Requirement	Description
No additional hardware required	

10.5.2 Unit Test Software

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

Table 10-4. Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Ktux.dll	Required to run tests in kernel mode
Kato.dll	Kato logging engine, which is required for logging test data
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
SDMATEST.dll	Test .dll file

10.5.3 Building the Unit Tests

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

1. Within the Platform Builder, choose **Build OS** > **Open Release Directory**.
   A DOS prompt is displayed.
2. Change to the SDMA Tests directory: `\WINCE600\SUPPORT_PDK1_7\TEST\SDMA`
3. Enter **set WINCEREL=1** on the command prompt and press return.
   This copies the DLL to the flat release directory.
4. Input **build -c** to build the CSPDDK test.

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

10.5.4 Running the Unit Tests

The command line for running the DDK_SDMA tests is `tux -o -d SDMATEST -n`. The CSPDDK_SDMA tests do not contain any test-specific command line options. Table 10-5 describes the test cases contained in the DDK_SDMA tests.

Table 10-5. DDK_SDMA Test Cases

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

10.6 CSPDDK DLL Reference

10.6.1 CSPDDK DLL System Clocking (DDK_CLK) Reference

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

10.6.1.1 DDK_CLK Enumerations

Table 10-6. 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 in 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 one (CKO) signal selections
DDK_CLOCK_ACM	Module name of Audio Clock Mux (ACM) configuration
DDK_CLOCK_ACM_SRC	Clock source selections of Audio Clock Mux (ACM) configuration
DDK_DVFC_SETPOINT	Frequency/voltage setpoints supported by the DVFC driver

10.6.1.2 DDK_CLK Functions

10.6.1.2.1 DDKClockSetGatingMode

This function sets the clock gating mode of the peripheral.

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

Parameters

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

mode	[in] Requested clock gating mode for the peripheral
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.1.2.2 DDKClockGetGatingMode

This function retrieves the clock gating mode of the peripheral.

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

Parameters

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

10.6.1.2.3 DDKClockGetFreq

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

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

Parameters

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

10.6.1.2.4 DDKClockConfigBaud

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

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

Parameters

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

10.6.1.2.5 DDKClockConfigACM

This function configures the Audio Clock Mux (ACM) source selections for the specified Audio module.

```
BOOL DDKClockConfigACM(
        DDK_CLOCK_ACM module,
        DDK_CLOCK_ACM_SRC src)
```

Parameters

module	[in] Audio module to configure
src	[in] Selects the ACM source
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.1.2.6 DDKClockSetCKO

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

```
BOOL DDKClockSetCKO(
        BOOL bEnable,
        DDK_CLOCK_CKO_SRC src,
        UINT32 div1,
        UINT32 preDiv,
        UINT32 postDiv)
```

Parameters

bEnable	[in] Set to TRUE to enable CKO output. Set to FALSE to disable CKO output
src	[in] Selects the CKO source signal
div1	[in] Specifies the CKO_DIV1, either 0 or 1
preDiv	[in] Specifies the CKO_DIV[5:3]
postDiv	[in] Specifies the CKO_DIV[2:0]
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.1.2.7 DDKClockSetpointRequest

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

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

Parameters

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

10.6.1.2.8 DDKClockSetpointRelease

This function releases a setpoint previously requested using DDKClockSetpointRequest.

```
BOOL DDKClockSetpointRelease(
        DDK_DVFC_SETPOINT setpoint,
        DDK_DVFC_DOMAIN domain)
```

Parameters

setpoint	[in] Specifies the setpoint to be released
domain	[in] Specifies DVFC domain for which the setpoint is requested
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.1.3 DDK_CLK Examples

Example 10-1. CSPDDK Clock Gating

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

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

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

Example 10-2. CSPDDK Clock Rate Query

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

UINT32 freq;

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

10.6.2 CSPDDK DLL GPIO (DDK_GPIO) Reference

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

10.6.2.1 DDK_GPIO Enumerations

Table 10-7. DDK_GPIO Enumerations

Programming Element	Description
DDK_GPIO_PORT	GPIO module instance
DDK_GPIO_DIR	Direction the GPIO pins
DDK_GPIO_INTR	Detection logic used for generating GPIO interrupts

10.6.2.2 DDK_GPIO Functions

10.6.2.2.1 DDKGpioSetConfig

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

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

Parameters

port	[in] GPIO module instance
pin	[in] GPIO pin [0-31]
dir	[in] Direction for the pin
intr	[in] Interrupt configuration for the pin
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.2.2.2 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

10.6.2.2.3 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

10.6.2.2.4 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

10.6.2.2.5 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

10.6.2.2.6 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

10.6.2.2.7 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

10.6.2.2.8 DDKGpioClearIntrPin

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

```
BOOL DDKGpioClearIntrPin(
        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

10.6.2.3 DDK_GPIO Example

Example 10-3. 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);
```

10.6.3 CSPDDK DLL IOMUX (DDK_IOMUX) Reference

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

10.6.3.1 DDK_IOMUX Enumerations

Table 10-8. DDK_IOMUX Enumerations

Programming Element	Description
DDK_IOMUX_PIN	Functional pin name used to configure the IOMUX. The enum value corresponds to the index to the SW_MUX_CTL registers
DDK_IOMUX_PIN_MUXMODE	Mux mode for a signal

Table 10-8. DDK_IOMUX Enumerations (continued)

Programming Element	Description
DDK_IOMUX_PIN_SION	Configuration on Software Input On Field to force the selected mux mode Input path no matter of mux mode functionality. If no SION bit for a PIN, the DDK_IOMUX_PIN_SION_NULL should be set
DDK_IOMUX_PAD	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	Slew rate for a pad; if no SLEW bit for a PAD, the DDK_IOMUX_PAD_SLEW_NULL should be set
DDK_IOMUX_PAD_DRIVE	Drive strength for a pad; if no DRIVE bit for a PAD, the DDK_IOMUX_PAD_DRIVE_NULL should be set.
DDK_IOMUX_PAD_OPENDRAIN	Open drain for a pad; if no ODE bit for a PAD, the DDK_IOMUX_PAD_OPENDRAIN_NULL should be set
DDK_IOMUX_PAD_HYSTERESIS	Hysteresis mode for a pad; if no HYS bit for a PAD, the DDK_IOMUX_PAD_HYSTERESIS_NULL should be set
DDK_IOMUX_PAD_VOLTAGE	Specifies the driver voltage for a pad, either 1.8 V or 3.3 V
DDK_IOMUX_PAD_PULL	Pull-up/pull-down/keeper configuration for a pad
DDK_IOMUX_SELECT_INPUT	Functional pad name to be selected and involved in Daisy Chain

10.6.3.2 DDK_IOMUX Functions

10.6.3.2.1 DDKIomuxSetPinMux

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

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

Parameters

pin [in] Functional pin name used to select the pin that is configured

muxmode [in] Mux mode configuration

sion [in] Sion configuration

Return Values Returns TRUE if successful, otherwise returns FALSE

10.6.3.2.2 DDKIomuxGetPinMux

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

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

Parameters

pin [in] Functional pin name used to select the pin that is returned

pMuxmode	[out] Mux mode configuration
pSion	[out] Sion configuration
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.3.2.3 DDKIomuxSetPadConfig

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

```
BOOL DDKIomuxSetPadConfig(
        DDK_IOMUX_PAD pad,
        DDK_IOMUX_PAD_SLEW slew,
        DDK_IOMUX_PAD_DRIVE drive,
        DDK_IOMUX_PAD_OPENDRAIN openDrain,
        DDK_IOMUX_PAD_PULL pull,
        DDK_IOMUX_PAD_HYSTERESIS hysteresis,
        DDK_IOMUX_PAD_VOLTAGE voltage)
```

Parameters

pad	[in] Functional pad name used to select the pad that is configured
slew	[in] Slew rate configuration
drive	[in] Drive strength configuration
openDrain	[in] Open drain configuration
pull	[in] Pull-up/pull-down/keeper configuration
hysteresis	[in] Hysteresis configuration
voltage	[in] Drive voltage configuration
Return Values	Returns TRUE if successful, otherwise returns FALSE.

10.6.3.2.4 DDKIomuxGetPadConfig

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

```
BOOL DDKIomuxSetPadConfig(
        DDK_IOMUX_PAD pad,
        DDK_IOMUX_PAD_SLEW *pSlew,
        DDK_IOMUX_PAD_DRIVE *pDrive,
        DDK_IOMUX_PAD_OPENDRAIN *pOpenDrain,
        DDK_IOMUX_PAD_PULL *pPull,
        DDK_IOMUX_PAD_HYSTERESIS *pHysteresis,
        DDK_IOMUX_PAD_VOLTAGE *pVoltage)
```

Parameters

pad	[in] Functional pad name used to select the pad that is configured
pSlew	[out] Slew rate configuration
pDrive	[out] Drive strength configuration
pOpenDrain	[out] Open drain configuration
pPull	[out] Pull-up/pull-down/keeper configuration
pHysteresis	[out] Hysteresis configuration
pVoltage	[out] Drive voltage configuration

Return Values Returns TRUE if successful, otherwise returns FALSE.

10.6.3.2.5 DDKIomuxSelectInput

This function writes a daisy value into the IOMUX SELECT_INPUT register to select the pad that is the input to the port.

```
BOOL DDKIomuxSelectInput(
        DDK_IOMUX_SELEIN port,
        UINT32 daisy)
```

Parameters

port [in] Port to select input

daisy [in] Data to be written

Return Values Returns TRUE if successful, otherwise returns FALSE

10.6.3.3 DDK_IOMUX Examples

Example 10-4. CSPDDK IOMUX Signal Multiplexing

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

// Configure the signal multiplexing for GPIO1_5. The ALT0 mux mode is configured
// and the regular sion is assigned for the GPIO1_5 ot the GPIO module.
DDKIomuxSetPinMux(DDK_IOMUX_PIN_GPIO1_5, DDK_IOMUX_PIN_MUXMODE_ALT0,
DDK_IOMUX_PIN_SION_REGULAR);
```

Example 10-5. CSPDDK IOMUX Pad Configuration

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

// Configure the GPIO1_5 pad for the following configuration: fast slew rate,
// high drive strength, no opendrain, no pull, no hysteresis, and 3.3 V drive voltage.
DDKIomuxSetPadConfig(DDK_IOMUX_PIN_GPIO1_5, DDK_IOMUX_PAD_SLEW_FAST,
DDK_IOMUX_PAD_DRIVE_HIGH, DDK_IOMUX_PAD_OPENDRAIN_DISABLE,
DDK_IOMUX_PAD_PULL_NONE, DDK_IOMUX_PAD_HYSTERESIS_DISABLE,
DDK_IOMUX_PAD_VOLTAGE_3V3);
```

10.6.4 CSPDDK DLL SDMA (DDK_SDMA) Reference

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

10.6.4.1 DDK_SDMA Enumerations

Table 10-9. DDK_SDMA Enumerations

Programming Element	Description
DDK_DMA_ACCESS	Width of the data for a peripheral DMA transfer

Table 10-9. DDK_SDMA Enumerations (continued)

DDK_DMA_FLAGS	Mode flags within the DMA buffer descriptor
DDK_DMA_REQ	DMA request used to trigger SDMA channel execution

10.6.4.2 DDK_SDMA Functions

10.6.4.2.1 DDKSdmaOpenChan

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

```
UINT8 DDKSdmaOpenChan(
        DDK_DMA_REQ dmaReq,
        UINT8 priority)
```

Parameters

dmaReq [in] Specifies the DMA request that is bound to a virtual channel

priority [in] Priority assigned to the opened channel

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

10.6.4.2.2 DDKSdmaUpdateSharedChan

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

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

Parameters

chan [in] Virtual channel returned by DDKSdmaOpenChan

dmaReq [in] Specifies the DMA request that is bound to a virtual channel

Return Values Returns TRUE if successful, otherwise returns FALSE

10.6.4.2.3 DDKSdmaCloseChan

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

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

Parameters

chan [in] Virtual channel returned by DDKSdmaOpenChan function

Return Values Returns TRUE if successful, otherwise returns FALSE

10.6.4.2.4 DDKSdmaAllocChain

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

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

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan
numBufDesc	[in] Number of buffer descriptors to be allocated for the chan
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.4.2.5 DDKSdmaFreeChain

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

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

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.4.2.6 DDKSdmaSetBufDesc

This function configures a buffer descriptor for a DMA transfer.

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

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan.
index	[in] Index of buffer descriptor within the chain to be configured.
modeFlags	[in] Specifies the buffer descriptor mode word flags that control the continue, wrap, and interrupt settings
memAddr1PA	[in] For memory-to-memory transfers, this parameter specifies the physical memory source address for the transfer. For memory-to-peripheral transfers, this parameter specifies the physical memory source address for the transfer. For peripheral-to-memory transfers, this parameter specifies the physical memory destination address for the transfer
memAddr2PA	[in] Used only for memory-to-memory transfers to specify the physical memory destination address for the transfer. Ignored for memory-to-peripheral and peripheral-to-memory transfers
dataWidth	[in] Used only for memory-to-peripheral and peripheral-to-memory transfers to specify the width of the data for the peripheral transfer. Ignored for memory-to-memory transfers
numBytes	[in] Virtual channel returned by DDKSdmaOpenChan
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.4.2.7 DDKSdmaGetBufDescStatus

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

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

Parameters

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

10.6.4.2.8 DDKSdmaGetChainStatus

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

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

Parameters

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

10.6.4.2.9 DDKSdmaClearBufDescStatus

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

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

Parameters

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

10.6.4.2.10 DDKSdmaClearChainStatus

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

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

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.4.2.11 DDKSdmaInitChain

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

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

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan
waterMark	[in] Specifies the watermark level used by the peripheral to generate a DMA request. This parameter tells the DMA how many transfers to complete for each assertion of the DMA request. Ignored for memory-to-memory transfers
Return Values	Returns TRUE if successful, otherwise returns FALSE

10.6.4.2.12 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

10.6.4.2.13 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 11
Display Driver for IPUv1

The Windows Embedded CE 6.0 BSP display driver is based on the Microsoft DirectDraw Graphics Primitive Engine (DDGPE) classes and supports the Microsoft DirectDraw interface. This driver combines the functionality of a standard LCD display with DirectDraw support. The display driver interfaces with the Image Processing Unit v1 (IPUv1). 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 type:

- CHUNGHWA 7" WVGA Display With Touch Screen (CLAA070VC01)

11.1 Display Driver Summary

Table 11-1 identifies the source code location, library dependencies and other BSP information.

Table 11-1. Display Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\IPU
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SoC*>\IPU
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\IPU
Driver DLL	ddraw_ipu.dll
Import Library	ddgpe.lib, gpe.lib
Catalog Items	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Display > CHUNGHWA CLAA070VC01(WVGA)
SYSGEN Dependency	SYSGEN_DDRAW=1
BSP Environment Variables	BSP_NODISPLAY= BSP_I2CBUS1=1 BSP_PP=1 BSP_PF=1 (if need PF)

11.2 Supported Functionality

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

1. RGB565 user interface

2. DirectDraw Hardware Abstraction Layer (DDHAL)
3. One overlay surface
4. Video overlays containing image data in any of the following FOURCC pixel formats: RGB565, UYVY or YV12
5. Hardware-accelerated color space conversion in video overlays
6. Hardware-accelerated image resizing in video overlays
7. Overlay surface color key feature
8. Two power management modes: full on and full off (resume and suspend)
9. Screen rotation
10. Supports CHUNGHWA 7" WVGA Display With Touch Screen (CLAA070VC01)
11. Supports WVGA landscape display resolution(800x480)

Note: The following limitations apply to the display driver overlay support.

12. The display panel flickers when the ATA driver operates in UDMA5 by ADMA controller, due to memory bus bandwidth
13. For other limitations refer to the release notes

11.3 Hardware Operation

For operation and programming information, see the chapter on the Image Processing Unit (IPU) in the IC Reference Manual.

11.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

11.3.2 Rotation Control

The 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, 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. See the Windows Embedded CE 6.0 Help, stating that GDI screen rotation cannot be used with DirectDraw.

11.4 Software Operation

11.4.1 Communicating with the Display

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

11.4.1.1 Using the Graphics Device Interface

The Graphics Device Interface (GDI) provides basic controls for the display of text and graphics. For information, see the Help:

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

11.4.1.2 Using DirectDraw

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

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, YV12, or UYVY 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 multiple image processing hardware blocks, which are used within the display driver to accelerate the following operations:

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

11.4.1.3 Using Display Driver Escape Codes

In some cases, applications might need to communicate directly with a display driver. To make this possible, an escape code mechanism is provided as part of the display driver. For a detailed description of standard display driver escape codes, see the CE Help:

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

11.4.2 Configuring the Display

The display configuration is based on the **PanelType** registry key, which is described in Section 11.4.2.3, "Display Registry Settings". The **PanelType** registry key indicates the display panel that is being used. There is only one supported display panel: The CHUNGHWA CLAA070VC01 WVGA LCD panel.

11.4.2.1 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.

NOTE

Due to lack of support for the co-existence of GDI screen rotation and 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. See the Windows CE Help, stating that GDI screen rotation cannot be used with DirectDraw.

11.4.2.2 Display Driver Blit Acceleration

On-chip Graphics Processing Unit (GPU) 2D cores may be accessed through the display driver to accelerate a subset of the GDI graphical blit operations. The subsequent sections provide details on the features offered by the GPU core, and how to configure the BSP to enable acceleration through the GPU core.

11.4.2.2.1 GPU2D Graphics Acceleration

GPU2D core graphics acceleration may be enabled through the following steps:

1. Enable the GPU base by setting the BSP_GPU_BASE environment variable. This may be achieved by selecting at least one GPU catalog item from the Third Party Catalog.
2. Enable the GPU2D component by setting the platform environment variable BSP_DISPLAY_Z160=1. This may be achieved by navigating to the project properties, and adding the environment variable in the Configuration Properties->Environment window.

11.4.2.2.1.2 Supported Acceleration Features

1. Solid color fills.
2. BitBlt() - Simple operations not requiring rotation or resizing.
3. StretchBlt() - Support for COLORONCOLOR and BILINEAR stretch modes. For a DDraw blt, the default stretch mode is BILINEAR.
4. PolyLine() - Support for horizontal and vertical line draws and bias whose llGamma equals to 0.
5. PatBlt() - Pattern copy blits are accelerated.
6. Mask blt: MaskBlt() function calls use this feature. For ROP4 value MAKEROP4(SRCCOPY, 0X00AA0029)

7. Blitting a UYVY surface to an RGB surface: The UYVY data format should be yCbCr.

   The Y,U,V data range is:

   Y = 0.257R + 0.504G + 0.098B + 16(16~235)

   U = -0.148R - 0.291G + 0.439B + 128(16~240)

   V = 0.439R - 0.368G - 0.071B + 128(16~240)
8. Alphablend blt: Both perpixel alpha and constant alpha are supported. To enable this feature, the "alphablend API"(SYSGEN_GDI_ALPHABLEND) catalog item must be included in the OS image.
9. The following accelerated ROP operations: BLACKNESS, PATCOPY, SRCCOPY, WHITENESS.
10. All of the above features are also supported when the screen is rotated.
11. 16BPP and 32BPP are supported.

11.4.2.2.1.3 Hardware Restrictions

- The GPU2D cannot draw a line with a non-zero llGamma value.
- Due to a GPU2D precision limitation, the coordinates of certain pixels be offset by small amount after an accelerated blit completes. As a result, the MaskBlt and StretchBlt GDI CETK tests may not pass(case 208,218,...).
- The GPU2D bilinear algorithm differs from the algorithm used in the Micorsoft-provided emulated blit software routines. As a result, the GPU2D bilinear stretch blt will result in a mismatch with the CETK reference image(case #218).
- GPU2D fails the AlphaBlend CETK test(case #231). The color output after an alpha blend blit operation may have a single-bit mismatch when compared with the reference image.

11.4.2.3 Display Registry Settings

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

11.4.2.3.1 i.MX35 Registry Settings

If the CHUNGHWA WVGA 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:3             ; CHUNGHWA WVGA Panel
  "VideoMemSize"=dword:600000     ; 6.0MB (960KB/frame )
```

When the OS image is configured to use graphics acceleration through the GPU, the C2DFlag key is also included. The C2DFlag key controls the types of graphical blit operations that are accelerated by the GPU. The following bits control which blits are accelerated:

- Bit 0 - Enable/Disable solid color fill acceleration

- Bit 1 - Enable/Disable pattern fill acceleration
- Bit 2 - Enable/Disable simple bitblt (without rotation, stretchblt) acceleration
- Bit 3 - Enable/Disable line draw acceleration
- Bit 4 - Enable/Disable maskblt acceleration
- Bit 5 - Enable/Disable stretchblt acceleration
- Bit 8 - Enable/Disable acceleration for rotated screen cases

In the following example C2DFlag setting, acceleration is enabled for pattern fill, line draw, stretchblt, and rotated screen cases, while acceleration is disabled for solid color fill, simple bitblt, and maskblt:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
   "C2DFlag"=dword:12a     ; Flag for c2d
```

11.4.2.4 Post Filter

The Post Filter performs postfiltering for the MPEG-4 (deblocking and deringing) or H.264 (deblocking) video compression standards. To use the Post Filter, set “BSP_PF=1“.

11.4.3 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, such as IOCTL_POWER_CAPABILITIES, IOCTL_POWER_QUERY, IOCTL_POWER_GET and IOCTL_POWER_SET.

11.4.3.1 PowerUp

This function is not implemented for the display driver.

11.4.3.2 PowerDown

This function is not implemented for the display driver.

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

11.5 Unit Test

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

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

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

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

11.5.1 Unit Test Hardware

The CHUNGHWA CLAA070VC01 WVGA panel is needed to run the GDI and DirectDraw tests. The panel displays the graphics data.

11.5.2 Unit Test Software

11.5.2.1 GDI Tests

Table 11-4 lists the software required to run the GDI tests.

Table 11-4. GDI Software Requirements

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

11.5.2.2 DirectDraw Tests

Table 11-5 lists the software required to run the DirectDraw tests.

Table 11-5. DirectDraw Software Requirements

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

11.5.2.3 Windows Media Player Tests

Table 11-6 lists the software required to perform WMV playback with Windows Media Player.

Table 11-6. Windows Media Player Software Requirements

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

11.5.3 Building the Unit Tests

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

Windows Embedded CE Test Kit > Running the CETK

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

11.5.4 Running the Unit Tests

11.5.4.1 Running the GDI Tests

The command line for running the GDI tests is:

```
tux -o -d gdiapi.dll
```

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

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

11.5.4.2 Running the DirectDraw Tests

The command line for running the DirectDraw tests is:

```
tux –o –d ddrawtk
```

NOTE

The display driver fails the following DirectDraw CETK test cases: 1240, 1250, 1340, 1350. The failure occurs because the hardware requires the overlay surface dimensions to be a multiple of 16 pixels, and the failing tests create surfaces with dimensions that violate this restriction.

11.5.4.3 Running the Windows Media Player tests

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

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

For example, `ceplayer motocross_208x160_30fps.wmv`

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

To confirm the correct operation of this test, observe the application and verify that the video clip is playing at a smooth rate (it should not drop frames or otherwise appear jerky). It should have a clear image, normal coloring, and correct image sizing.

11.6 Display Driver API Reference

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

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

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

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

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

Chapter 12
Dynamic Voltage and Frequency Control (DVFC) Driver

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

12.1 DVFC Driver Summary

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

Table 12-1. DVFC Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\DVFC
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\DVFC
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\DVFC
Driver DLL	dvfc_max8660.dll or dvfc_mc13892.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > DVFC driver support using the MC13892
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_DVFC = 1

12.2 Supported Functionality

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

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

12.3 Hardware Operation

12.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

12.3.2 i.MX35 3-Stack Configuration

The DVFC driver is dependent upon the MAX8660/MC13892 PMIC interface for dynamic voltage control via the I^2C1 port. The I^2C driver and MAX8660/MC13892 PMIC SDK are needed by the DVFC driver. The I^2C interface is used to control voltage.

MX35 supports 2 CCM schemes called Consumer Path and AUTO Path, which are controlled by a fuse inside the chip. The target maximum CPU frequency for Consumer Path is 532 MHz, for AUTO Path is 399 MHz.

12.4 Software Operation

12.4.1 i.MX35 Registry Settings

The following registry keys are required to properly load the i.MX35 DVFC module.

```
;-------------------------------------------------------------------------------
; DVFC Driver
;
IF BSP_BOARD_RED !
IF BSP_DVFC_MC13892
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\DVFC]
   "Prefix" = "DVF"
   "Dll"="dvfc_mc13892.dll"
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"    ; PMCLASS_GENERIC_DEVICE
ENDIF ; BSP_DVFC_MC13892
ELSE
IF BSP_DVFC_MAX8660
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\DVFC]
   "Prefix" = "DVF"
   "Dll"="dvfc_max8660.dll"
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"    ; PMCLASS_GENERIC_DEVICE
ENDIF ; BSP_DVFC_MAX8660
ENDIF ; BSP_BOARD_RED !
;-------------------------------------------------------------------------------
```

12.4.2 Loading and Initialization

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

12.4.3 Operation

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

12.4.3.1 i.MX35 Voltage/Frequency Setpoints

The DVFC driver supports four voltage frequency setpoints. Table 12-2 provides the voltage/frequency characteristics for these setpoints.

Table 12-2. DVFC Setpoints

Setpoint Name	CPU/BUS/PER Clock (MHz)	Core Voltage
DDK_DVFC_SETPOINT_TURBO	532/133/66.5	1.450 V
DDK_DVFC_SETPOINT_HIGH	399/133/66.5	1.200 V
DDK_DVFC_SETPOINT_MEDIUM	133/66.5/66.5	1.200 V
DDK_DVFC_SETPOINT_LOW	133/66.5/33.25	1.200 V

The setpoint attributes are controlled by the definitions in the DVFC driver code (found in \PLATFORM\<*Target Platform*>\SRC\DRIVERS\DVFC\COMMON\setpoint.c). The DVFC driver uses these definitions to populate a global setpoint array (g_SetPointConfigCon/g_SetPointConfigAuto) that is referenced during setpoint transitions. For AUTO clock path, the maximum CPU frequency supported is 399 MHz, the nominal AUTO path setpoint TURBO is equal to HIGH.

12.4.3.2 i.MX35 Setpoint Mapping

The peripherals may not be able to operate properly in all of the supported setpoints due to minimum frequency/voltage requirements. Therefore, drivers that support these peripherals need a method of communicating setpoint requirements. Drivers communicate clocking and setpoint requirements through the use of APIs in the CSPDDK. The mapping of DDK_CLK clock management routines (DDKClockSetGatingMode) to DVFC setpoints is located in **UpdateSetpointRequestCount** (found in \PLATFORM\<*Target Platform*>\SRC\DRIVERS\CSPDDK\DDK_CLK\ddk_clk.c). To change the setpoint mapping for a specific peripheral, modify the code in **UpdateSetpointRequestCount**.

WARNING

Do not map a peripheral to a setpoint that violates the electrical specification or does not provide adequate clocking for the peripheral protocol specification.

The DVFC driver advertises support for IOCTL_POWER requests from Power Manager. A IOCTL_POWER_SET request is mapped to a setpoint by the DVFC driver. This mapping allows applications to use the Power Manager APIs to request changes in the DVFC setpoint. The mapping of

device power states (D0-D4) to DVFC setpoints is located in **DvfcMapDevPwrStateToSetpoint** (found in \PLATFORM\<*Target Platform*>\SRC\DRIVERS\DVFC\COMMON\mappm.c). To change the setpoint mapping for a specific device power state (D0-D4), modify the code in **DvfcMapDevPwrStateToSetpoint**.

12.4.3.3 i.MX35 Frequency Scaling Operation

The DVFC driver supports frequency scaling of the MCU, MAX (AHB), IPG (peripheral), HSP (IPU), and NFC clock domains. This section describes the main features of the frequency scaling.

12.4.3.3.1 i.MX35 DDR Bus Scaling

Scaling the AHB bus frequency with DDR memory requires a calibration sequence to be executed from an internal uncached memory space. The calibration sequence must disable interrupts to prevent bus masters from accessing the DDR during the sequence. A custom kernel IOCTL (**IOCTL_HAL_DVFC_BUS_SCALE**) called from the DVFC driver is used to perform DDR delay line calibration. The DDR calibration sequence is executed from uncached IRAM defined by **IMAGE_WINCE_DVFC_IRAM** definitions in \PLATFORM\<*Target Platform*>\SRC\INC\image_cfg.h.

12.4.3.3.2 i.MX35 IPG_PER_CLK Clock Scaling

The AHB to IPG clocks are kept at a fixed ratio of 1-to-2. The EPIT timer peripheral is configured to clock from PERCLK. The DVFC driver does not scale the IPG_PER_CLK clock unless all peripherals that depend on it are inactive.

Peripherals that use synchronous clocking can continue to operate at a reduced IPG_PER_CLK**/**PERCLK ratio without changes to the data transmission clock rate. However, peripherals that use asynchronous clocking (UART) or have specific clocking requirements, cannot tolerate PERCLK rate changes between transfers. In such cases, the CSPDDK source code can be modified to restrict setpoints for specific peripherals. Refer to Section 12.4.3.2, “i.MX35 Setpoint Mapping for more information regarding the mapping between DVFC setpoints and peripheral clocking. Upon invoking **DDKClockSetGatingMode,** the DVFC coordinates a setpoint transition, if necessary, to grant the driver a voltage/frequency setpoint that meets or exceeds the request.

Timer peripherals (EPIT, GPT) that are configured to clock from the IPG clock, experience clock skew/drift during IPG clock scaling. The OAL support for the OS tick timer using EPIT1 is configured by PERCLK that prevents IPG scaling.

Without removing SOC modules depends upon the MEDIUM setpoint; the LOW setpoint is never entered. The LOW setpoint is kept for backwards compatibility.

12.4.4 DDK Interface

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

- DDKClockSetpointRequest, Section 10.6.1.2.7, “DDKClockSetpointRequest.”

- DDKClockSetpointRelease, Section 10.6.1.2.8, "DDKClockSetpointRelease."

12.4.5 Power Management

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

12.4.5.1 PowerUp

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

12.4.5.2 PowerDown

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

12.4.5.3 IOCTL_POWER_CAPABILITIES

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

12.4.5.4 IOCTL_POWER_SET

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

12.4.5.5 IOCTL_POWER_GET

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

12.5 Unit Test

Chapter 13
Enhanced Secure Digital Host Controller (eSDHC) Driver

The eSDHC module supports Multimedia Cards (MMC), Secure Digital Cards (SD) and Secure Digital I/O and Combo Cards (SDIO). The eSDHC driver provides the interface between the Microsoft SD Bus driver and the eSDHC hardware.

13.1 eSDHC Driver Summary

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

Table 13-1. eSDHC Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\ESDHC
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ESDHC
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\ESDHC
Driver DLL	esdhc.dll
SDK Library	esdhcbase_common_fsl_v2PDK1_7.lib, esdhcbase_<*Target SOC*>.lib, sdcardlib.lib, sdhclib.lib, sdbus.lib
Catalog Item	Third Party > BSP > Freescale i.MX35 3DS PDK1_7: ARMV4I > Device Drivers > SD Host Controller > Enhanced SD Host Controller 1 (ESDHC1) Support
SYSGEN Dependency	SYSGEN_SD_MEMORY=1
BSP Environment Variables	BSP_NOESDHC= BSP_ESDHC1=1 BSP_ESDHC2=1 IMGSDBUS2

13.2 Supported Functionality

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

1. Enhanced Secure Digital Host Controllers
2. Designed and implemented as close as possible to Standard Host Controller Driver in CE 6.0 R2
3. Compliant with the SDBUS2 driver provided in CE 6.0 R2
4. Fast Path
5. DMA or PIO modes of data transfers based on value of eSDHC driver registry key, DisableDMA

6. SD, SD High Capacity (up to spec v2.1), MMC (up to spec v4.3), and SDIO cards (up to spec v2.0). High capacity MMC cards are not supported because SDBUS2 in CE 6.0 R2 does not support these cards
7. One host supports only one card connected to it
8. DLL supports multiple instances of the eSDHC controller
9. Configuration of the block sizes from 1–4096 bytes in single and multi-block modes
10. Insertion and removal of card, even when system is suspended; when the system resumes, the card (if present) is remounted
11. Power states on(D0) and off (D4), D1–D3 states are treated as D4
12. Clocks are gated in D4 state, and ungated in D0 state, except for SDIO cards for which clocks are never gated.
13. Write protect switch on SD cards
14. Combo cards (for example, SD memory + WiFi functionality on same card)
15. MMC cards in 1-bit mode and SD/SDIO cards in 4-bit modes due to limitation in SDBUS2 in CE 6.0 R2

13.3 Hardware Operation

Refer to the chapter on the eSDHC in the *Applications Processor Reference Manual* for detailed operation and programming information.

13.3.1 Conflicts with Other Peripherals and Catalog Options

13.3.1.1 Conflicts with SoC Peripherals

All ESDHC1 and ESDHC2 pads can be configured for their primary function in Alternate Mode 0. However, some ESDHC1 pads are shared with MSHC in ALT Mode1, IPU in ALT Mode 2 and USB OTG in ALT Mode 3. ESDHC2 conflicts with UART3/I2C3/CAN1/USB HOST, and some ESDHC2 pads can be configured as ESDHC1 DATA5–DATA8 for 8-bit mode. ESDHC3 pads can be configured in Alternate mode 3, while the primary function in Alternate Mode 0 are used fro IPD function.

13.3.1.2 Conflicts with Other Board Peripherals

No conflicts.

13.4 Software Operation

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

13.4.1 Required Catalog Items

13.4.1.1 SD and MMC Memory Card Support

Catalog > Device Drivers > SDIO > SDIO Memory > SD Memory

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

13.4.2 eSDHC Registry Settings

13.4.2.1 i.MX35 SDHC Registry Settings

```
; @CESYSGEN IF CE_MODULES_SDBUS

IF BSP_ESDHC1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ESDHC_MX35]
    "Order"=dword:19
    "Dll"="esdhc.dll"
    "Prefix"="SHC"
    "Index"=dword:1
    "DisableDMA"=dword:0 ; Use this reg setting to disable both internal and external DMA
    "MaximumClockFrequency"=dword:3197500          ; 52 MHz max clock speed
    "UseExternalDMA"=dword:0
    "SDIOPriority"=dword:64                        ; IST thread priority of 100

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

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

[HKEY_LOCAL_MACHINE\Comm\PegasSDN1\Parms]
    "SdioBitMode"=dword:00000001

[HKEY_LOCAL_MACHINE\Comm\PegasSDN1\Parms]
   "DisablePowerManagement"=dword:1
   "ResetOnResume"=dword:0
   "RebindOnResume"=dword:1
ENDIF BSP_ESDHC1

; @CESYSGEN ENDIF CE_MODULES_SDBUS
```

13.4.3 DMA Support

13.4.3.1 DMA Support

DMA mode is supported by the eSDHC driver. The driver does not allocate or manage DMA buffers internally except for a start buffer and an end buffer for non-aligned buffers that are provided 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.

13.4.3.1.1 i.MX35 DMA Support

Both DMA mode and non-DMA modes are supported by the driver. DMA mode is used by default, and user can change the *DisableDMA* value in registry file esdhc_mx35.reg to enable non-DMA mode. Internal DMA on ESDHC is used. Two internal DMA modes are supported by the ESDHC hardware: Simple DMA and Advanced DMA. The driver supports only ADMA mode.

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 page aligned address (4096 bytes).

Due to cache coherency issues from processor and DMA access of the memory, the above criteria is further constrained for the read or receive operation (it is not applicable for write or transmit) by:

- Number of bytes to transfer should be cache line size (32 bytes) aligned.

For buffer chain that does not meet the above criteria, the driver uses its own start and end buffers that are page and cache-aligned. Later, when the DMA completes, a memcpy is done from the temporary start and end buffers back to the original non-aligned buffers.

13.4.4 Power Management

The eSDHC driver does self-management of the module clocks for power savings during inactivity. Only two power states are supported by the driver: D0 when all clocks are on, and D4 when all clocks are gated. The IOCTL_POWER calls are never entered in this driver because it does not register with the CE Power Manager. Instead, the SHC_powerUp and SHC_PowerDown APIs are the entry points for suspend and resume functionality.

13.4.4.1 i.MX35-3DS Power Management

Clocks to the ESDHC module are turned off (gated) when there is no card present in the socket. No power gating is implemented on this platform. The power to the socket is always on.

13.5 Unit Test

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

- File System Driver Test

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

13.5.1 Unit Test Hardware

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

Table 13-2. Hardware Requirements

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

13.5.2 Unit Test Software

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

Table 13-3. Software Requirements

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

13.5.3 Building the Unit Tests

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

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

13.5.4 Running the Unit Tests

The following sections describe 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 Platform Builder Help, or view the online documentation at the following URL:
http://msdn2.microsoft.com/en-us/library/aa934353.aspx

13.5.4.1 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"
```

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

13.5.4.2 Storage Device Block Driver Read/Write Tests

Use the command line

```
tux –o –d rwtest –c "-z"
```

to run the tests. This only tests one card at a time.

13.5.4.3 Storage Device Block Driver API Tests

Use the command line

```
tux –o –d disktest –c "-z"
```

to run the tests. This only tests one card at a time.

13.5.4.4 Storage Device Block Driver Performance Tests

Use the command line

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

to run the tests. This tests only one card at a time.

13.5.4.5 Partition Driver Test

Use command line

```
tux –o –d msparttest –c "-z"
```

to run the tests.

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

13.5.5 System Testing

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

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

13.6 Secure Digital Card Driver API Reference

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

Chapter 14
Enhanced Serial Audio Interface (ESAI) Driver

The Enhanced Serial Audio Interface (ESAI) provides a serial port for serial communication with a variety of serial devices.

14.1 ESAI Driver Summary

The ESAI consists of independent transmitter and receiver sections, each section with its own clock generator. It is called synchronous because all serial transfers are synchronized to a clock. Up to six transmitters and four receivers are supported. Table 14-1 provides a summary of source code location, library dependencies and other BSP information.

Table 14-1. ESAI Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\ESAI
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ESAI
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\ESAI
Driver DLL	esai_wm8580.dll
SDK Library	esai_common_fsl_v2_PDK1_7.lib, esai_<*Target SOC*>.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > ESAI
SYSGEN Dependency	SYSGEN_AUDIO
BSP Environment Variables	BSP_NOAUDIO= BSP_ESAI=1 BSP_ASRC=1

14.2 Supported Functionality

The ESAI audio driver enables the 3-Stack System to provide the following software and hardware support:

1. Conforms to the audio driver architecture as defined for Windows Embedded CE 6.0 and all related operating systems
2. Uses double-buffered DMA operations to transfer audio data
3. Supports multi-channel PCM wave data playback function
4. Supports multi-channel PCM wave data record function

5. Supports 16-bit and 24-bit PCM data. (24-bit PCM data is packed in bits 23–0 of the 32 bits)
6. Supports 1–6 channel PCM data playback. (Refer to software operation for detail)
7. Supports 1–4 channel PCM data record. (Refer to software operation for detail)
8. Supports playback function with Freescale hardware platforms that include the WM8580 multi-channel audio Codec
9. Supports playback without ASRC support at sample rate: 16KHz,32 KHz, 44.1 KHz, 48 KHz, 64KHz,88.2 KHz, 96 KHz, 128KHz, 176.4KHz, 192KHz (WM8580 Codec) Please note: sample rate about 48KHz is not supported if the playback audio protocol is configured as network mode, refer to the registry setting about the protocol setting.
10. Supports playback with ASRC support from sample rate 32K–96K (16-bit PCM data is not supported by the ASRC)
11. Supports record function with Freescale hardware platform that includes the AK5702 Codec
12. Supports record sample rates: 48 KHz, 44.1 KHz, 32 KHz, 24 KHz, 16 KHz, 12 KHz, 8 KHz (AK5702 Codec)

14.3 Hardware Operation

Refer to the chapter on the ESAI in the *Multimedia Applications Processor Reference Manual* for detailed operation and programming information.

14.3.1 Conflicts with Other Peripherals and Catalog Items

N/A

14.3.1.1 Conflicts with 3-Stack Peripherals

N/A

14.3.2 Hardware Limitation

14.3.2.1 Channel Order

In the ESAI hardware implementation, all the transmitters share one data FIFO. When multiple transmitters are used, the audio data from the FIFO is transferred to the different transmitters is sequence, such as TX0, TX1, TX2, TX0, TX1, TX2, and so on. Since the different transmitters use the same slot mask, when multiple transmitters are used, the channel mask is not handled well. The channel mask can only be used when only one transmitter is being used.

Also, the mapping from channel number to the transmitter port changes according to the channel numbers when multiple transmitters are used.

For example, when three transmitters are used for 1–6 channels of audio playback:

Channel Number = 6: CH0,CH3->TX0, CH1,CH4->TX1, CH2,CH5->TX2

Channel Number = 4, CH0,CH2->TX0, CH1,CH3->TX1

Channel Number = 2, CH0,CH1->TX0

For receive, the problem is similar and the channel number should be even for both playback and record.

14.3.2.2 Full Duplex Support

The ESAI module cannot support full duplex. Therefore, playback or record cannot be performed at the same time.

14.4 Software Operation

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

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

14.4.1 Required Catalog Items

Third Party > BSP > Freescale<Target Platform>:ARMV4I > Device Drivers > CSPI Bus > CSPI Bus1

Third Party > BSP > Freescale<Target Platform>:ARMV4I > Device Drivers >I2C Bus > I2C Bus1

14.4.2 ESAI Registry Settings

14.4.2.1 i.MX35 Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ESAI]
"Prefix"="WAV"
    "Dll"="esai_wm8580.dll"
    "Index"=dword:3
    "Order"=dword:8
    "Priority256"=dword:95
    "AudioProtocol"=dword:0
    "IClass"=multi_sz:"{E92BC203-8354-4043-A06F-2A170BF6F227}",
                      "{37168569-61C4-45fd-BD54-9442C7DBA46F}"
```

The AudioProtocol key value can be set to 0 (one transmitter with network mode) or 2 (multi-transmitter with normal mode). When AudioProtocol is set to 0, one transmitter is used and the channel mask is well handled. Since all the audio data is transferred on one serial bus, the frame rate is limited by the bit clock. Sample rate beyond 48 K is not supported. In this mode, the mapping from slot number to the transmitter port is fixed.

When AudioProtocol is set to 2, multiple transmitters are used and 6-channel wave format is supported. To keep the mapping relationship between slot number and transmitters, the audio data needs to be packed to 6-channel format before it is transferred to the ESAI interface. In this case, the channel mask does not take effect.

This AudioProtocol affects only the playback function. For the record function, the bus protocol is decided by the driver and is not selectable.

14.4.3 Supported Wave Data Format

To access the ESAI audio interface, the WAVEFORMATEXTENSIBLE data structure must be used:

```
typedef struct {
  WAVEFORMATEX  Format;
  union {
    WORD  wValidBitsPerSample;
    WORD  wSamplesPerBlock;
    WORD  wReserved;
  } Samples;
  DWORD   dwChannelMask;
  GUID    SubFormat;
} WAVEFORMATEXTENSIBLE, *PWAVEFORMATEXTENSIBLE;
```

Format.wFormatTag must be set to WAVE_FORMAT_EXTENSIBLE. The dwChannelMask member does not take effect while AudioProtocol is set to 2 in the registry file. Format.nChannels can be set from 1 to 6, but when AudioProtocol is set to 2 in the registry file, only even number can be used (such as 2, 4, 6). For 24-bit audio data, set Samples.wValidBitsPerSample to 24 and Format.wBitsPerSample to 32. The SubFormat member is not used since only PCM data is supported.

14.4.4 DMA Support

14.4.4.1 DMA Support

Double-buffer is used for audio data transfer.

14.4.5 Power Management

This function is not implemented for the ESAI driver.

14.4.5.1 PowerUp

This function is not implemented for the ESAI driver.

14.4.5.2 PowerDown

This function is not implemented for the ESAI driver.

14.4.5.3 IOCTL_POWER_CAPABILITIES

This function is not implemented for the ESAI driver.

14.4.5.4 IOCTL_POWER_GET

This function is not implemented for the ESAI driver.

14.4.5.5 IOCTL_POWER_SET

This function is not implemented for the ESAI driver.

14.5 Unit Test

The ESAI driver interface supports only wave data that conforms with the WAVEFORMATEXTENSIBLE structure. Therefore the driver might not be supported by general audio applications.

14.5.1 Building the Unit Test

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

1. Within the Platform Builder, choose **Build OS** > **Open Release Directory**.
   A DOS prompt is displayed.
2. Change to the M_Player Tests directory: `\WINCE600\SUPPORT_PDK1_7\TEST\ESAI\M_PLAYER`
3. Enter **set WINCEREL=1** on the command prompt and press return.
   This copies the .exe file to the flat release directory
4. Enter **build -c** at the prompt and press return
5. Change to the M_Recorder Tests directory: `\WINCE600\SUPPORT_PDK1_7\TEST\ESAI\M_RECORDER`
6. Enter set **WINCEREL=1** on the command prompt and press return.
   This copies the .exe file to the flat release directory
7. Enter **build -c** at the prompt and press return

After the build completes, the m_player.exe and m_recorder.exe files are located in the $(_FLATRELEASEDIR) directory.

14.5.2 Hardware Setup

The audio extension card is necessary for the ESAI test. The multi-channel audio codec for playback and record function is located on the audio extension card. Connect the audio card with CN9 on the i.MX35 personality board (refer to the board user manual for details).

For playback function, connect the earphone/speaker with J2 on the audio card, and 6 channels are supported at most. For record function, connect audio input line with J3 on audio card, and 4 channels are supported at most.

14.5.3 Running the Unit Test

14.5.3.1 Playback Function Test

The m_player application is used for the playback function test. Earphone or speakers can be used to hear the sound.

Usage: `m_player wave_file_name channel_number channel_mask`

Example: `m_player temp\source.wav 6 0x3f`

In this example, the source.wav is played through the ESAI in six channels and the channel mask is 0x3f. The wave file used for testing is a general stereo wave file and the application extends it to multi-channel wave format. The wave file can be a 16-bit or 24-bit (packed into bits 23–0 of the 32 bits) data file.

To run the application within VS2005, go to the **Target** menu option and select the **Run Programs** menu option. This gives a list of applications that can be run on the OS. Select m_player.exe from this list and click on Run to run this application.

14.5.3.2 Record Function Test

The m_recorder application is used for the record function test. The sound from the audio input line is recorded in the destination wave file and can be played later for verification.

Usage: `m_recorder wave_file_name seconds_length sample_rate bit_depth channel_number channel_mask`

Example: `m_record temp\target.wav 10 48000 16 4 0xf`

In this example, the target.wav file is recorded through the ESAI. The file is in wave format: 10 seconds in length, 48 KHz sample rate, 16-bit depth, 4 channels and the channel mask is 0xf.

To run the application within VS2005, go to the **Target** menu option and select the **Run Programs** menu option. This gives a list of applications that can be run on the OS. Select m_record.exe from this list and click on Run to run this application.

If the bit depth is set to 32, the recorded data is 24-bit (packed into bits 23–0 of the 32 bits). The channel number indicates the number of channels in the audio data, and the mask indicates which channel contains data and which channel contains zero. Zeros should not be present in the data, but there is a limitation in the hardware conversion process that generates zeros. If a bit in the mask is zero, the corresponding bits are zeros in the interleaved audio data. The channel number also includes such “zero-data” channel.

NOTE

These applications are mainly used for simple function test and API demo usage. User might encounter wave file format related failures (like wave format chunk length and “fact” chunk is not well handled). Edit the source code to resolve the problem.

14.5.4 Known Issues

On some audio extension cards, the mulit-channel audio codec WM8580 is not stable at 44.1 KHz/88.2 KHz/176.4KHz. Playback with audio files at these sample rates may cause noise.

Chapter 15
Fast Ethernet Controller (FEC) Driver

The Fast Ethernet driver is used for connectivity with an IEEE 802.3 Ethernet using the on-chip Fast Ethernet Controller. The driver provides support to communicate with the Ethernet at 10/100 Mbps, using a MII compatible interface and an external transceiver (SMCS LAN8700 and Am79C874). The Fast Ethernet driver is NDIS 4.0 compliant miniport driver.

15.1 Fast Ethernet Driver Summary

Table 15-1 provides a summary of source code location, library dependencies and other BSP information.

Table 15-1. FEC Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\FEC
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\FEC
Driver DLL	fec.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>:ARMV4I > Device Drivers > FEC
SYSGEN Dependency	SYSGEN_NDIS=1 SYSGEN_TCPIP=1 SYSGEN_WINSOCK=1
BSP Environment Variables	BSP_NOFEC=

15.2 Supported Functionality

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

1. Compliant with the NDIS 4.0 miniport driver
2. 10/100 Mbps network
3. MII PHY or RMII PHY

15.3 Hardware Operations

The Fast Ethernet Controller connects with the external transceivers using standard MII (Media Independent Interface) connection. All the registers in the external transceivers can be accessed by the MII compatible management frames. The interrupt signal from the external transceiver is connected to the processor through the PBC (Peripheral Bus Controller). Refer to the Peripheral Bus Controller document for detailed operation and programming information. The attached transceiver for the Fast Ethernet Controller can detect the speed of the ethernet network automatically by the auto-negotiation process. The software accesses the status register of attached transceiver to determine the speed of the ethernet network (10 Mbps or 100 Mbps).

15.3.1 Conflicts with Other Peripherals and Catalog Items

15.3.1.1 Conflicts with SoC Peripherals

No conflicts.

15.3.1.2 Conflicts with 3-Stack Peripherals

No conflicts.

15.4 Software Operations

The Fast Ethernet driver follows the Microsoft-recommended architecture for NDIS miniport drivers. The details 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**.

15.4.1 FEC Driver Registry Settings

The following register keys are required to properly load the Fast Ethernet driver and to configure the TCP/IP for Ethernet interface. To enable dynamic IP address assignment using DHCP, the variable EnableDHCP should be set to 1.

```
[HKEY_LOCAL_MACHINE\Comm\FEC]
"DisplayName"="FEC Ethernet Driver"
"Group"="NDIS"
"ImagePath"="fec.dll"
[HKEY_LOCAL_MACHINE\Comm\FEC\Linkage]
"Route"=multi_sz:"FEC1"
[HKEY_LOCAL_MACHINE\Comm\FEC1]
"DisplayName"="FEC Ethernet Driver"
"Group"="NDIS"
"ImagePath"="fec.dll"
[HKEY_LOCAL_MACHINE\Comm\FEC1\Parms]
"BusNumber"=dword:0
"BusType"=dword:0
; DuplexMode: 0:AutoDetect; 1:HalfDuplex; 2:FullDuplex.
"DuplexMode"=dword:0
```

```
; The Ethernet Physical Address. For example,

; Ethernet Address 00:24:20:10:bf:03 is MACAddress1=0024,
; MACAddress2=2010,and MACAddress3=bf03.
"MACAddress1"=dword:1213
"MACAddress2"=dword:1728
"MACAddress3"=dword:3121

[HKEY_LOCAL_MACHINE\Comm\FEC1\Parms\TcpIp]
; This should be MULTI_SZ
"DefaultGateway"="" ; This should be SZ... If null it means use LAN, else WAN and
Interface.
"LLInterface"="" ; Use zero for broadcast address? (or 255.255.255.255)
"UseZeroBroadcast"=dword:0 ;Thus should be MULTI_SZ, the IP address list
"IpAddress"="0.0.0.0"; This should be MULTI_SZ, the subnet masks for the above IP
"Subnetmask"="0.0.0.0"
"EnableDHCP"=dword:1

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

[HKEY_LOCAL_MACHINE\Comm\Tcpip\Linkage]
; This should be MULTI_SZ
; This is the list of llip
"Bind"=multi_sz:"FEC1"
```

15.5 Unit Tests

The Fast Ethernet driver is tested using the following:

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

15.5.1 Unit Test Hardware

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

Table 15-2. Hardware Requirements

Requirement	Description
HW Platform System	—
PC/machine	Counterpart for network operation
An Ethernet or a cross Ethernet cable	To and from an Ethernet

15.5.2 Unit Test Software

Table 15-3 lists the required software to run the unit tests.

Table 15-3. Software Requirements

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

15.5.3 Building the Unit Tests

15.5.3.1 Network Utilities Related Tests

- To include the ping utilities, the SYSGEN_NETUTILS = 1 needs to be set. Under **Catalog** > **Core OS** > **CEBASE** > **Communication Services and Networking** > **Networking General** > **Network Utilities**, IpConfig, Ping, and Route should be included in the OS design.
- To include FTP, SYSGEN_FTPD = 1 needs to be set. **Catalog** > **Core OS** > **CEBASE** > **Communication Services and Networking** > **Servers** > **FTP Server** should be included in the OS design.
- The following registry entry needs to be added to reg to allow get and put of files using the anonymous FTP login:

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

15.5.3.2 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` file can be found alongside the other required CETK files in the following location:

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

15.5.3.3 Winsock Performance Test

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

`Perf_winsock2.dll` in:

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

`Perf_winsockd2.exe` in:

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

15.5.3.4 One-Card Network Card Miniport Driver Test

The One-card network card miniport driver test comes pre-built as part of the CETK. No steps are required to build these tests. The `ndt.dll` and `ndt_1c.dll` 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
```

15.5.3.5 NDIS Performance Test

The NDIS performance test comes pre-built as part of the CETK. No steps are required to build these tests. The `ndp.dll` and `perf_ndis.dll` 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
```

15.5.4 Running the Unit Tests

15.5.4.1 Network Utilities Related Tests

15.5.4.1.1 Ping Tests

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

15.5.4.1.2 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 Fast Ethernet network interface (Control Panel Network and Dial-up Connections) Proxy server, if used in the test network on the Pocket Internet explorer.

15.5.4.1.3 FTP Tests

For running FTP tests, the FTP service needs to be started on the tested device using the following command on the DOS prompt:

```
services start FTP0:
```

15.5.4.2 Winsock 2.0 Test (v4/v6)

The test can be executed by using

```
tux –o –d Ws2bvt.dl
```

in the command line on the tested device. For detailed information on the Winsock 2.0 Test (v4/v6) tests, see the Platform Builder Help:

Windows Embedded CE Test Kit > CETK Test and Test Tools > CETK Tests > Ethernet Tests > Tests Winsock 2.0 Test(v4/v6).

15.5.4.3 Winsock Performance Test

Start the server on the PC by typing

```
Perf_winsockd2 - install
```

at the command line. Then client side test executes on the second device by using

```
tux –o –d Perf_winsock2.dll –c "-s 10.193.101.41"
```

in the command line on the tested target device, where 10.193.101.41 denotes PC IP address and needs to be replaced appropriately. For detailed information on the Winsock Performance tests, see the Platform Builder Help:

Windows Embedded CE Test Kit > CETK Test and Test Tools > CETK Tests > Performance Test > Wisock Performance Test.

NOTE

Cases 1007 and 1008 fail. This is a known MSFT CETK issue.

15.5.4.4 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 FEC1"
```

on the command line on the tested target device. For detailed information on the Winsock Performance tests, see the Platform Builder Help:

Windows Embedded CE Test Kit > CETK Test and Test Tools > CETK Tests > Ethernet Tests > One-card Network Card Miniport Driver Test.

15.5.4.5 NDIS Performance Test

This test can be done by including `ndp.dll` and `perf_ndis.dll` in the image, and starting the test by entering

```
tux –o –d perf_ndis.dll –c "FEC1"
```

on the command line on the tested target device. For detailed information on the Winsock Performance tests, see the Platform Builder Help:

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

15.6 Fast Ethernet Driver API Reference

The Fast Ethernet driver conforms to NDIS 4.0 specification by Microsoft for the miniport network drivers. For reference information on basic NDIS driver functions, methods, and structures, see the CE Help:

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

Chapter 16
FM Radio Driver

The FM Radio driver is compatible with the Stream Interface driver framework. This chapter explains how to develop the FM radio application that interfaces to the hardware components of the radio chip. The FM radio driver controls the Si4702 chip.

16.1 Radio Driver Summary

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

Table 16-1. Radio Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\RADIO
Driver DLL	fm_radio.dll
SDK Library	N/A
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Radio
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_RADIO=1 BSP_I2CBUS1=1

16.2 Supported Functionality

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

1. Supports the Windows CE streams interface.
2. Supports the main functions of FM Radio: Power on/off, Set Frequency, Set Volume, Muted and Auto Scan.
3. Supports the Si4702 chip.

16.3 Hardware Operation

The driver uses I^2C to interact with radio chip. The radio driver uses I^2C to interact with Si4702 chip. For information, see *the Silicon Laboratories Si4702 Guide.*

16.3.1 Conflicts with Other Peripherals and Catalog Items

No Conflicts.

16.4 Software Operation

The only interface to control the Radio driver is provided by the IOCTLs.

16.4.1 Registry Settings

16.4.1.1 i.MX35 Registry Settings

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

```
; These registry entries load the FM Radio driver. The IClass value is 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}"
```

16.5 Power Management

The primary method for limiting power consumption in the Radio module is to power down the chip when the FM Radio driver is not in use. The application calls IOCTL_SET_POWER with the parameter POWER_OFF to power down the chip.

16.5.1 Power Up

This function is not implemented for the Radio driver.

16.5.2 Power Down

This function is not implemented for the Radio driver.

16.5.3 IOCTL_POWER_SET

16.5.3.1 i.MX35 IOCTL_POWER_SET Support

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, which puts the system in a suspend state. The system is resumed for a request of value D0.

16.6 Unit Test

The Radio CETK test cases verify the functionality of the radio driver with radio chip. The FM Radio Application can also be used to verify the radio driver. For information, see the *FM Radio Application section of the User's Guide.*

16.6.1 Unit Test Hardware

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

Table 16-2. Hardware Requirements

Requirement	Description
Radio chip	N/A

16.6.2 Unit Test Software

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

Table 16-3. Software Requirements

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

16.6.3 Building the Unit Tests

16.6.3.1 FM 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 OS** menu option and select the **Open Release Directory** menu option. This opens a DOS prompt.
2. Change to the Radio Tests directory: \WINCE600\SUPPORT_PDK1_7\TEST\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.

16.6.3.2 FM Radio Custom Application

In order to build the FM Radio custom application, 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 opens a DOS prompt.
2. Change to the Radio application directory: \WINCE600\SUPPORT_PDK1_7\APP\FM
3. Enter **set WINCEREL=1** on the command prompt and hit return. This copies the built executable to the flat release directory.
4. Enter the build command (*build -c*) at the prompt and press return.

After the build completes, the FM application file is located in the $(_FLATRELEASEDIR) directory.

16.6.4 Running the Unit Tests

The command line for running the radio tests is:

```
tux –o –d radio_test
```

To redirect the test results to a file, add the option **–f**. Radio tests do not contain any test-specific command line options.

16.7 Radio Driver API Reference

16.7.1 Radio Driver IOCTLS

This section describes 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 this IOCTL are described. Most of the IOCTLs are explained in other sections.

16.7.1.1 RADIO_IOCTL_GET_CAPS

This DeviceIoControl request determines the capability of the hardware.

Parameters

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

16.7.1.2 RADIO_IOCTL_GET_TUNER

This DeviceIoControl request used to retrieve the tuner information of the hardware.

Parameters

hOpenContext	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function.

pBufIn	NULL
pBufOut	pointer to RADIO_TUNER type data return to caller.

16.7.1.3 RADIO_IOCTL_SET_TUNER

This DeviceIoControl request sets tuner information of the hardware.

Parameters

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

16.7.1.4 RADIO_IOCTL_GET_AUDIO

This DeviceIoControl request used to retrieve the audio information of the hardware.

Parameters

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

16.7.1.5 RADIO_IOCTL_SET_AUDIO

This DeviceIoControl request sets audio information of the hardware, such as volume and whether muted.

Parameters

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

16.7.1.6 RADIO_IOCTL_GET_FREQ

This DeviceIoControl request used to retrieve the current frequency of the hardware.

Parameters

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

16.7.1.7 RADIO_IOCTL_SET_FREQ

This DeviceIoControl request tunes to the frequency.

Parameters

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

16.7.1.8 RADIO_IOCTL_GET_POWER

This DeviceIoControl request is used to retrieve the power state of the hardware.

Parameters

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

16.7.1.9 RADIO_IOCTL_SET_POWER

This DeviceIoControl request sets the power state of the hardware.

Parameters

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

16.7.1.10 RADIO_IOCTL_AUTO_TUNE

This DeviceIoControl request auto scans all available channels.

Parameters

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

16.7.1.11 RADIO_IOCTL_GET_LAST_ERROR

This DeviceIoControl returns the last return code.

Parameters

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

16.7.2 Radio Driver Structures

16.7.2.1 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;
```

16.7.2.2 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;
```

16.7.2.3 Radio Audio Structure

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

16.7.2.4 Radio Power State Structure

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

16.7.2.5 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 17
General Purpose Timer (GPT) Driver

The GPT is a multipurpose module used to measure intervals or generate periodic output. The GPT 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.

17.1 GPT Driver Summary

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

Table 17-1. GPT Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\GPT
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\GPT
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\GPT
Driver DLL	gpt.dll
SDK Library	gptsdk.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > GPT > GPT
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_GPT=1

17.2 Supported Functionality

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

1. Clock source selection including IPG_CLK (microsecond level precision) and GPT_32KCLK (microsecond level precision)
2. Both reset and free-run mode count operation
3. Two power management modes: power on and power off
4. Exposes the SDK API interface which is used by application

NOTE

GPT_IPGCLK is adapted for short time period (GPT_IPGCLK is 66.5 MHz, the maximum time period is 64.599 seconds), while the maximum time period of GPT_32KCLK is approximately 37 hours, 16 minutes, 57 seconds.

17.3 Hardware Operation

Refer to the chapter on GPT in the *Applications Processor Reference Manual* for detailed hardware operation and programming information.

17.3.1 Conflicts with Other Peripherals and Catalog Items

Because the external GPT clock source is not used, GPT module does not conflict with other peripherals.

17.4 Software Operation

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

17.4.1 GPT Registry Settings

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

17.4.2 Communicating with the GPT

The GPT driver controls the General Purpose Timer. This timer is used to provide high resolution (microsecond) timing functionality to other platform modules. The GPT is a stream interface driver and is thus accessed through the file system APIs. To communicate using the GPT, a handle to the device must first be obtained using the **GptOpenHandle** function. Subsequent commands to the device are issued using various APIs supported by this driver. For more information about the API refer to Section 17.7, "GPT SDK API Reference." To use this API, it is necessary to include the `gptsdk.lib` library.

17.4.2.1 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.

The following code shows how to open a handle to the GPT:

```
// Global data
// Handle to the GPT device
HANDLE g_hGpt = NULL;

// opening the GPT1 port.
g_hGpt = GptOpenHandle(L"GPT1:");
```

17.4.2.2 Create Event for GPT

```
HANDLE GptCreateTimerEvent(HANDLE hGpt, LPTSTR eventName)
// Function: GptCreateTimerEvent
//
// This method returns a handle triggered
// when the GPT timer period has elapsed.
//
// Parameters:
//      hGpt
//          [in] Handle to GPT driver.
//
//      eventName
//          [in] String identifying timer event.
//
// Returns:
//      Timer event handle created. Handle is NULL if failure.
```

The following is an example:

```
// Name to create the named event for Timer
#define GPT_EVENT_NAME L"GptTest1"

// create an event for the timer interrupt
hGptIntr = GptCreateTimerEvent(hGpt, GPT_EVENT_NAME);
```

17.4.2.3 Configuring the GPT

Calling the **GptStart**(`g_hGpt, pTimerConfig`) function starts the GPT module and enables the timer event trigger. g_hGpt is valid and opened handle for GPT, and pTimerConfig struct is as follows:

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

and timerSrc may select *GPT_IPGCLK* or *GPT_32KCLK*.

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

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

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

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

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

After the GPT starts to time and the timer event handle is created, call the following command to wait the coming of the predefined time:

```
// waiting for event triggering
if(WaitForSingleObject(g_hGptIntr, INFINITE) == WAIT_OBJECT_0)
{
}
```

17.4.2.4 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.

The following code shows how 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);
```

To pause the timer and then restart for a moment, use the **GptStop** function, as follows:

```
GptStop(g_hGpt);
Sleep(sometime);
GptResume(g_hGpt);

BOOL GptResume(HANDLE hGpt)
// Function: GptResume
//
// This method reactivates the GPT(Usually called after a Stop))
//
// Parameters:
//      hGpt
//          [in] Handle to GPT driver.
//
// Returns:
//      TRUE if success.
//      FALSE if failure.
```

17.4.3 DMA Support

The GPT driver does not use the DMA.

17.5 Power Management

The primary method for limiting power consumption in the GPT module is to gate off all clocks to the module when the 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()**.

17.5.1 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.

17.5.2 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 disabled.

17.5.3 IOCTL_POWER_SET

This function is not implemented for the GPT driver.

17.6 Unit Test

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

17.6.1 Unit Test Hardware

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

Table 17-2. Hardware Requirements

Requirement	Description
No additional hardware required	

17.6.2 Unit Test Software

Table 17-3 lists the required software to run the unit tests.

Table 17-3. Software Requirements

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

17.6.3 Building the Unit Tests

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

1. Within the Platform Builder, choose **Build OS** > **Open Release Directory**.
   A DOS prompt is displayed.
2. Change to the GPT Tests directory: `\WINCE600\SUPPORT_PDK1_7\TEST\GPT`
3. Enter **set WINCEREL=1** on the command prompt and press return.
   This copies the DLL to the flat release directory.

4. Input **build -c** to build GPT test.

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

17.6.4 Running the Unit Tests

To run this test the `tux.exe` and `kato.dll` files must be present in the release directory. These files are not present by default and need to be copied from this location:

```
\Program Files\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4i
```

to the release directory.

To run the test using the Target Control window use the following steps:

1. Within the Platform Builder, go to the Target menu option and select the Target Control menu option. This opens a Windows CE Command Prompt window
2. Run on the Command Prompt windows this command: `s tux -o -d gpttest.dll`

The test starts and the results can be viewed in the Output panel in the Visual Studio.

Table 17-4 describes the test cases contained in the GPT tests.

Table 17-4. 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_PeriodicMode	Periodic mode test
5: TST_FreerunMode	Free run mode test
6: TST_StopAndResume	Stop and resume test

17.7 GPT SDK API Reference

17.7.1 GPT SDK Functions

17.7.1.1 GptOpenHandle

This API creates a handle to the GPT stream driver.

```
HANDLE GptOpenHandle(
        LPCWSTR lpDevName);
```

Parameters

lpDevName [in] Device name to open

Return Values Open handle to the specified file indicates success INVALID_HANDLE_VALUE indicates failure

Remarks Use the GptCloseHandle function to close the handle returned by GptOpenHandle()

17.7.1.2 GptCreateTimerEvent

This API is used to create the GPT Timer event.

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

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
eventName	[in] Pointer to a null-terminated string that specifies the name of the object
Return Values	Non-null handle to the specified event indicates success. NULL indicates failure
Remarks	Use the **GptReleaseTimerEvent** function to close the event. The system closes the handle automatically when the process terminates. The event object is destroyed when its last handle has been closed.

17.7.1.3 GptStart

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

```
BOOL GptStart(
        HANDLE hGpt,
        pGPT_Config pTimerConfig);
```

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
pTimerConfig	[in] Object of the **pGPT_Config** structure
Return Values	TRUE on success and FALSE indicates a failure
Remarks	Set desired event trigger time and start GPT

17.7.1.4 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] Pointer to the variable which receives current counter value
Remarks	None

17.7.1.5 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

17.7.1.6 GptStop

This API disables the GPT interrupt and stops the GPT timer.

```
BOOL GptStop(
        HANDLE hGpt);
```

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
Return Values	TRUE on success and FALSE indicates a failure
Remarks	None

17.7.1.7 GptReleaseTimerEvent

This API closes the currently open GPT Timer Event.

```
BOOL GptReleaseTimerEvent(
        HANDLE hGpt,
        LPTSTR eventName);
```

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
eventName	[in] Pointer to a null-terminated string that specifies the name of the object
Return Values	Nonzero indicates success; Zero indicates failure To get extended error information, call GetLastError()
Remarks	None

17.7.1.8 GptCloseHandle

This API closes a handle to the GPT driver.

```
BOOL GptCloseHandle(
        HANDLE hGpt);
```

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
Return Values	Nonzero indicates success; Zero indicates failure To get extended error information, call GetLastError()
Remarks	None

17.7.2 GPT Driver Structures

17.7.2.1 GPT_Config

```
typedef struct
{       timerMode_c timerMode;
        UINT32 period;
        timerSrc_c timerSrc;
} 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)
timerSrc	Selects GPT clock source: GPT_IPGCLK or GPT_32KCLK

17.7.2.2 GPT_TIMER_SRC_PKT

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

Members

timerSrc	Select clock source between two supported timer clock sources: GPT_IPGCLK or GPT_32KCLK

Chapter 18
Global Positioning System (GPS) Driver

The Global Positioning System (GPS) enables a GPS receiver to determine its location, speed/direction, and time.

18.1 GPS Driver Summary

This 3-Stack platform supports the BroadCom BCM4750 Single Chip Assisted-GPS (*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, you must load certain software components on the platform in order to enable full operation. Table 18-1 provides a summary of source code location, library dependencies and other BSP information.

Table 18-1. GPS Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\GPS ..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\GPSCTRL
Driver DLL	gpscontroldriver.dll, GpsctService.dll, GlvcDriver.dll, log4net.dll, OpenNetCF_GL.dll, OpenNetCF.Windows.Form_GL.dll, OpenNetCF.Net_GL.dll,
SDK Library	N\A
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*>: ARMV4I > Device Drivers > GPS > GPS core DLL Third Party > BSPs > Freescale <*Target Platform*>: ARMV4I > Device Drivers > GPS > GPS Control Driver
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_GPS_CONTROL_DRIVER=1 BSP_GPS_COREDRIVER=1 BSP_SERIAL_UART3 =1 BSP_MCU_MC9S08DZ60=1

NOTE

Do not use FEC feature for the TO2 blue CPU board because UART3 and FEC share pins and UART3 is used by the GPS.

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. Resetting and power on/power off to the GPS module are controlled by the MCU GPIO pins. The GPS module functionality is segmented into subsystems. All of the subsystems do not need to be selected in order to enable GPS on the platform.

Figure 18-1 shows the architecture of GPS driver. Three layers in the GPS software system.

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

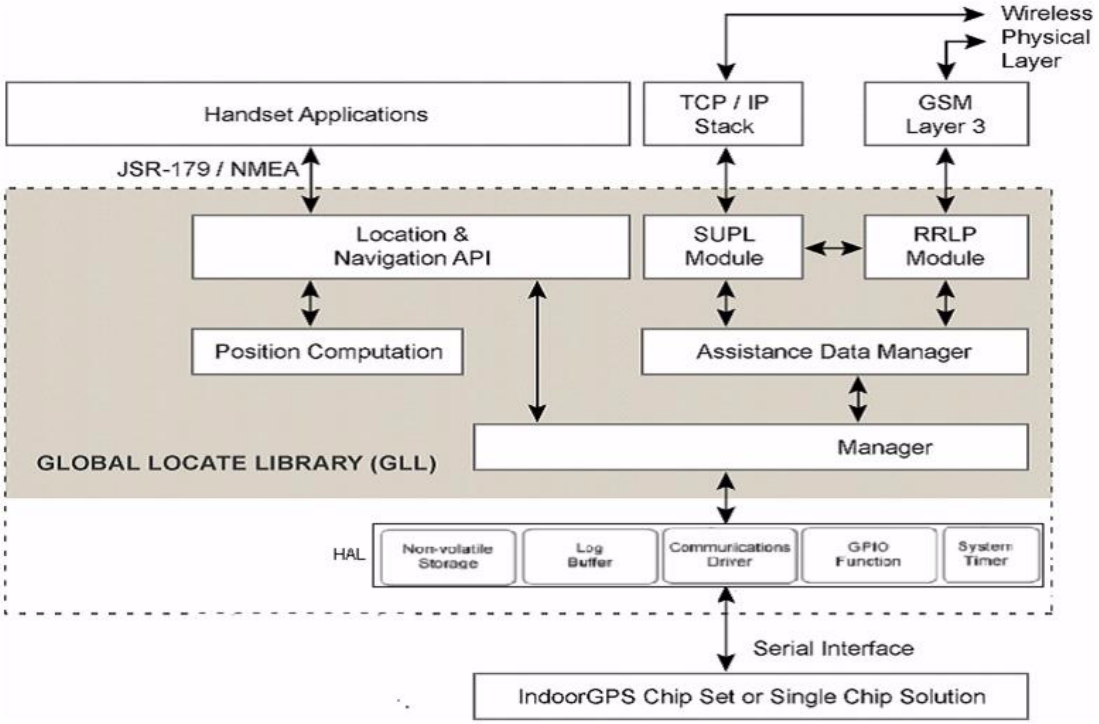

Figure 18-1. Software Architecture of GPS Driver

18.1.1 Application Layer

Handset applications, TCP/IP stack and the GSM layer3 belong to the 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 enable GPS functionality even when satellite signal is not good enough to get fix.

18.1.2 GPS Core Driver Layer

The deep color part (GLL) 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.

18.1.3 GPS HAL Driver Layer

GPS HAL drivers provide hardware related resource, such as serial port driver, non-volatile storage and GPIO functions. The driver is called as gpscontroldriver.dll, and source code can be found at:

PLATFORM\<Target Platform>\SRC\DRIVERS\GPSCTRL\.

18.2 Supported Functionality

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

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

18.3 Hardware Operation

The GPS driver exchanges data and command between GPS application layer and hardware module via UART port

18.3.1 Conflicts with Other Peripherals and Catalog Items

18.3.1.1 Conflicts with SoC Peripherals

None

18.3.1.2 Conflicts with 3-Stack Peripherals

18.3.1.3 i.MX35 Peripheral Conflicts

Because the GPS uses the UART3 port which shares some pins with the FEC module as a commication port in i.MX35 TO2 blue CPU board, do not add the GPS and FEC drivers into image at the same time.

18.3.2 i.MX35 Hardware Operation

18.3.2.1 UART Port

For i.MX35-3DS TO2 blue CPU board, UART3 port is used to communicate with the GPS module. If a different UART port is used for this purpose, then the following registry must be changed correspondingly:

..\PLATFORM\<Target Platform>\SRC\DRIVERS\GPS\GlobalLocate-Gpsct-flatrom.reg:

"GpsComPort"="COMx:", where “x” should be specified according to the UART actually used ("COM3:").

18.3.2.2 GPIO Control

Three MCU GPIO pins are used to control the GPS module as shown in Table 18-2.

Table 18-2. GPIO Control

GPIO Name	PIN	Value Description
GPS_RST	PTE6(MCU)	0: Reset of GPS module is asserted 1: Reset of GPS module is de-asserted
GPS_PWREN	PTE5(MCU)	1: GPS module is powered on 0: GPS module is powered off
32K_GPSA	PTG5(MCU)	Enable GPS 32K Clock

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*<Target Platform>*\SRC\DRIVERS\GPSCTRL\gpsctlpdd.cpp

18.4 Software Operation

18.4.1 Communicating with the GPS Module

Software applications communicate with the GPS module through a virtual COM port (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.

18.4.2 Power Management

The GPS_PowerUp and GPS_PowerDown functions 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.

18.4.3 GPS Driver Registry Settings

18.4.3.1 Configuration Registry Keys

The registry files can be found at:

PLATFORM\<Target Platform>\SRC\DRIVERS\GPS\GlobalLocate-Gpsct-flatrom.reg

PLATFORM\<Target Platform>\SRC\DRIVERS\GPS\gpscontroldriver.reg

18.5 Unit Test

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

Chapter 19
Graphics Processing Unit (GPU)

The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D graphics applications. The GPU2D (2D graphics processing unit) is based on the AMD Z160 core, which is an embedded 2D and vector graphics accelerator targeting the OpenVG 1.1 graphics API and feature set. The GPU driver is delivered only as binary code.

19.1 GPU Driver Summary

Table 19-1 provides a summary of source code location, library dependencies and other BSP information.

Table 19-1. GPU Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\GPU
Driver DLL	libkos.dll libgsl.dll amdgslldd.dll libgslmemcfg.dll libos.dlllibgsluser.dll libpanel.dll res_client.dllcsi.dll libOpenVG.dlllib2dz160k.dll lib2d-z160.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <*Target Platform*> > Device Drivers > GPU > Z160
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_GPU=1

19.2 Supported Functionality

The GPU driver enables the board to provide the following software and hardware support:

1. EGL™ (interface between Khronos rendering APIs such as OpenGL ES or OpenVG and the underlying native platform window system) 1.3 API defined by Khronos Group
2. OpenVG™ (royalty-free, cross-platform API that provides a low-level hardware acceleration interface for vector graphics libraries such as Flash and SVG) 1.1 API defined by Khronos Group
3. D0 (Full On) and D4 (Off) power states

19.3 Hardware Operation

Refer to the GPU chapter in the *i.MX35 Applications Processor Reference Manual* for detailed hardware operation and programming information.

19.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

19.4 Software Operation

19.4.1 Communicating with the GPU

The GPU driver is divided into two layers. The first layer is running in kernel mode, acting as the base driver for the whole stack and providing the essential hardware access, device management, memory management, command stream management, context management and power management. The second layer is running in user mode, implementing the stack logic and providing following APIs to the upper layer applications such as:

- EGL 1.3 API
- OpenVG 1.1 API

19.4.2 GPU Driver Files

Listed below is a brief introduction to the GPU driver files. This list is not complete. The platform.bib file contains the complete list.

- Files that reside in kernel space:
  - amdgslldd.dll—base GPU driver and the standard stream interface driver, provides essential access to GPU hardware
  - libkos.dll—contains OS helper functions
  - libgsl.dll—contains common Graphics System Layer (GSL) logic
  - lib2dz160k.dll—contains Z160 c2d helper functions
  - libgslmemcfg.dll—contains memory configuration helper functions
- Files that reside in user space
  - libos.dll—contains OS helper functions
  - libgsluser.dll—contains common Graphics System Layer (GSL) logic
  - lib2d-z160.dll—contains Z160 c2d helper functions
  - libpanel.dll—contains GPU configuration helper functions so that some configurations could be customized during runtime, instead of hard-built images
  - essc.dll—contains shader compiler logic
  - librenderboy.dll—contains the logic of rendering framework

— libEGL.dll—contains EGL implementation
— libOpenVG.dll—contains OpenVG 1.1 implementation
— res_client.dll—contains resource client helper functions for OpenVG
— csi.dll—contains command stream interface helper functions for OpenVG

19.4.3 Power Management

The GPU driver implements the PowerUp and PowerDown APIs with support for the D0 (Full On) and D4 (Off) power states. These states are handled in the following manner:

- D0—GPU clocks are not enabled until the GPU driver is required to enable the clocks, for example, when an OpenGL ES application is launched. The GPU driver disabled the clocks when applications exit. Additionally, the graphics core has integrated power management design that supports gated clock branches used to turn off idle blocks within the core. This block-level clock gating is managed automatically in the core and GPU driver enables this capability when configure the core at the initialization time.
- D4—GPU clocks are disabled and power supplies are also disabled when possible.

19.4.4 GPU Registry Settings

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

19.5 Float Pointing Acceleration using the ARM Vector Floating Point (VFP) Library

As this SOC includes a VFP module, graphics applications or drivers can use VFP to accelerate the mathematical algorithm. You can download the ARM VFP library release from the ARM website(http://www.arm.com/products/os/windowsce.html) and use the information in the release notes to enable the OEM floating point library support.

19.6 Unit Test

The following sections describe the unit tests for the GPU driver.

19.6.1 Unit Test Hardware

No special requirements.

19.6.2 Unit Test Software

The following sections describe the software for the GPU driver unit tests.

19.6.2.1 Tiger Test

This test application verifies the basic functionality of OpengVG 1.1. It is included into the release image and is located under `\Windows\tiger.exe`. Click to launch this test and a rotating tiger appears on the screen as shown in Figure 19-1. Press ESC to exit this application.

Figure 19-1. Tiger Test

19.6.2.2 OpenVG 1.1 Conformance Test

The OpenVG 1.1 conformance test is standard OpenVG conformance test designed by the Khronos Group. Visit the Khronos Group website at http://www.khronos.org/opengles/adopters/login/ for detailed information about how to download the source code, build the test binaries and run this tests.

19.6.2.3 Known Issues

- Refer to the release notes for up-to-date known issue list

19.7 GPU Driver API Reference

- For EGL 1.3 API refer to http://www.khronos.org/egl/ for detailed specifications
- For OpenVG 1.1 API refer to http://www.khronos.org/openvg/ for detailed specifications

Chapter 20
Inter-Integrated Circuit (I^2C) Driver

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

20.1 I^2C Driver Summary

Table 20-1 provides a summary of source code location, library dependencies and other BSP information.

Table 20-1. I^2C Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\I2C
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\I2C
Platform Driver Path	..\PLATFORM*Target Platform*>\SRC\DRIVERS\I2C
Import Library	N/A
Driver DLL	i2csdk.dll i2c.dll
Catalog Item	Third Party > BSP > Freescale <TGTPLAT> > Device Drivers > I2CBus
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_I2CBUS1=1

20.2 Supported Functionality

The I^2C driver supports the following features:

1. I^2C communication protocol
2. Multiple I^2C controllers
3. I^2C master mode of operation
4. I^2C slave mode of operation
5. Stream interface
6. Two power management modes: full on and full off

20.3 Hardware Operation

20.3.1 Conflicts with Other Peripherals and Catalog Items

20.3.1.1 Conflicts with SoC Peripherals

The i.MX35 platform contains three I^2C modules, but only one of these modules may be used on the i.MX35 3DS board, the I2C1 module. I2C2 and I2C3 pins are not available in the 3DS board, and the pads are used for CAN and ESAI.

20.3.1.2 Conflicts with Board Peripherals

No conflicts.

20.4 Software Operation

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

20.4.1 Registry Settings

20.4.1.1 i.MX35 Registry Settings

The following registry keys are required to properly load the I^2C 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
```

20.4.2 Communicating with the I^2C

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

20.4.3 Creating a Handle

Call the **CreateFile** function to open a connection to the I^2C device. An I^2C port must be specified in this call. The format is `I2CX:`, with `X` being the number indicating the I^2C port. This number should not exceed

the number of I^2C instances on the platform. If an I^2C port does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the I^2C:

1. Insert a colon after the I^2C port for the first parameter, *lpFileName*. For example, specify `I2C1:`.
2. Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to an I^2C 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 I^2C port.

```
// Open the I2C 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 I^2C port, configure the port. When an application opens an I^2C port, it uses the default configuration settings, which might not be suitable for the device at the other end of the connection.

20.4.4 Configuring the I^2C

Configuring the I^2C port for communications involves two main operations:

- Setting the master or slave mode
- Setting the I^2C clock rate

Before these actions can be taken, a handle to the I^2C port must already be opened. Each of these steps requires a call to the **DeviceIoControl** function. As parameters, the I^2C port handle, appropriate IOCTL code, and other input and output parameters are required. Use the helper APIs to correctly configure the port.

As an example, the code below configures an I^2C port:

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

if (hI2C == INVALID_HANDLE_VALUE)
{
        ERRORMSG(1, (L"Unable to open handle to I2C block\r\n"));
        retVal = -1;
        goto exit;
}

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

```
if (!I2CSetFrequency(hI2C, EEPROM_CLOCK_RATE))
{
        ERRORMSG(1, (L"Unable to set frequency\r\n"));
        retVal = -1;
        goto exit;
}
```

20.4.5 Data Transfer Operations

The I^2C driver provides one command, Transfer, that facilitates performing both reads and writes through the I^2C. The basic unit of data transfer in the I^2C 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 I^2C.

Before these actions can be taken, a handle to the I^2C port must already be opened, and it should already be configured in the correct mode with the correct frequency.

To perform an I^2C transfer:

1. Create an array of I2C_PACKET objects and initialize the fields of each packet as follows:
   a) Set the *byRW* field to I2C_RW_WRITE to specify that the I^2C operation is a write, or I2C_RW_READ to specify that the I^2C operation is a read.
   b) Set the *byAddr* field to the 7-bit I^2C slave address of the device to which the data is written.

NOTE

The byAddr field requires the 7-bit I2C slave address, aligned to the least significant 7 bits. This address is shifted left one bit and OR-ed with the read/write bit to compose the 8-bit value sent out during the I^2C 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 *byRW* is set to I2C_RW_READ, create a buffer of bytes to hold the data which is read from the slave device.
   d) Set the *wLen* field to the size, in bytes, of the read or write buffer. This indicates the number of bytes to write or read.
   e) Set the *lpiResult* field to point to an integer that holds the return value from the write operation.
2. Call the I2CTransfer SDK API to start the I^2C transfer.
3. After calling the I2CTransfer function, check the *lpiResult* field if the function returned FALSE, to narrow down the type of error that occurred.

The following code example demonstrates how to perform a transfer that contains one write and one read packet. The write is performed before the read operation.

```
I2C_TRANSFER_BLOCK I2CXferBlock;
I2C_PACKET I2CPacket[2];
BYTE byAddr = 0x2D;                         // Slave Address
```

```
BYTE byOutData = 0x39;                      // Data to write
BYTE byInData;                              // Read buffer

        // Packet 0 contains write operation
I2CPacket[0].pbyBuf = (PBYTE) &byOutData;
I2CPacket[0].wLen = sizeof(byOutData);

I2CPacket[0].byRW = I2C_RW_WRITE;
I2CPacket[0].byAddr = byAddr;
I2CPacket[0].lpiResult = lpiResult;

        // Packet 1 contains read operation
I2CPacket[1].pbyBuf = (PBYTE) &byInData;
I2CPacket[1].wLen = sizeof(byInData);

I2CPacket[1].byRW = I2C_RW_READ;
I2CPacket[1].byAddr = byAddr;
I2CPacket[1].lpiResult = lpiResult;

I2CXferBlock.pI2CPackets = I2CPacket;
I2CXferBlock.iNumPackets = 2;

        // Transfer data via I2C
if (!I2CTransfer(hI2C,&I2CXferBlock))
        {
                ERRORMSG(1, (_T("Data transfer failed!\r\n")));
                retVal = -1;
                goto exit; // examine value in lpiResult
        }
```

20.4.5.1 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 I^2C module. A START command of the I^2C 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 I^2C 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 I^2C STOP command.

20.4.6 Closing the Handle

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

20.4.7 Power Management

The power management method used in the I^2C 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

most BSP use cases, the I^2C module operates in master mode and never in slave mode. As a result, the I^2C module can be disabled, and its clocks turned off, whenever the module is not processing packets. In contrast, when the I^2C module operates in slave mode, the module has to be enabled, and have its clocks turned on at all times to properly receive the interrupt that signals the start of a data transfer from another I^2C master device.

As described in the **Data Transfer Operations** section, the I^2C data transfer operations are handled in I2C_TRANSFER_BLOCK objects, which contain one or more packets of I^2C data. The I^2C driver turns on the I^2C clocks and enables the I^2C module before processing an I2C_TRANSFER_BLOCK, and then disables and turns off clocks to the I^2C module after the block of packets has been processed. This limits the time during which the I^2C module is consuming power to the time during which the I^2C is actively performing data transfers.

20.4.7.1 PowerUp

This function is not implemented for the I^2C driver. Power to the I^2C module is managed as I^2C transfer operations are processed. There are no additional power management steps needed for the I^2C.

20.4.7.2 PowerDown

This function is not implemented for the I^2C driver.

20.4.7.3 IOCTL_POWER_SET

This function is implemented for the I^2C driver. When D4 power mode is set, the driver switches its operating mode to polling that does not produce interrupt events to the BSP system. When leaving the D4 power mode, the driver recovers its original operating mode.

20.5 Unit Test

20.5.1 Unit Test Hardware

20.5.1.1 I2C MCU Tests

MCU module which can be read/write by I2C.

20.5.2 Unit Test Software

20.5.2.1 I2C MCU Tests

Table 20-2 lists the software required to perform I2C MCU tests.

Table 20-2. I2C MCU Tests Software Requirements

Requirement	Description
i2cma.exe	MCU read/write tests by I2C sample application

20.5.3 Building the Unit Tests

20.5.3.1 I2C MCU Tests

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

1. Within the Platform Builder, choose **Build OS** > **Open Release Directory**.
   A DOS prompt is displayed.
2. Change to the I2C MCU tests directory: `\WINCE600\SUPPORT_PDK1_7\TEST\I2C\I2CMCU`
3. Input **build -c** to build the I2C MCU test.

After the build completes, the i2cma.exe file is located in the $(_FLATRELEASEDIR) directory.

20.5.4 Running the Unit Tests

20.5.4.1 I2C MCU Tests

To run the application within VS2005, go to the **Target** menu option and select the **Run Programs** menu option. This gives a list of applications that can be run on the OS. Select i2cma.exe from this list and click on Run to run this application.

If I2C read MCU successfully, will output massage “I2C read MCU successfully“.

If I2C read MCU failed, will output massage “I2C read MCU failed“.

20.6 Hardware Limitations

For the slave function, the hardware does not distinguish between a START and REPEATED START signal from the I^2C bus. Hence the driver checks the IAAS address cycle start flag to detect a new I^2C transmission.

20.7 I^2C Driver API Reference

20.7.1 I^2C Driver IOCTLS

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

20.7.1.1 I2C_IOCTL_GET_CLOCK_RATE

This DeviceIoControl request retrieves the clock rate divisor. The value is not the absolute peripheral clock frequency. The value retrieved should be compared against the I^2C specifications to obtain the true frequency.

Parameters

lpOutBuffer	Pointer to the divisor index. The true clock frequency is platform dependent. Refer to I^2C specification for more information
nOutBufferSize	Size in bytes of the divisor index

20.7.1.2 I2C_IOCTL_GET_SELF_ADDR

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

Parameters

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

20.7.1.3 I2C_IOCTL_IS_MASTER

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

Parameters

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

20.7.1.4 I2C_IOCTL_IS_SLAVE

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

Parameters

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

20.7.1.5 I2C_IOCTL_RESET

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

20.7.1.6 I2C_IOCTL_SET_CLOCK_RATE

This DeviceIoControl request initializes the I^2C device with the given clock rate. This IOCTL does not expect to receive the absolute peripheral clock frequency. Rather, it expects the clock rate divisor index stated in the I^2C specification. If absolute clock frequency must be used, use the macro I2C_MACRO_SET_FREQUENCY.

Parameters

lpInBuffer	Pointer to the divisor index. Refer to the I^2C specification to obtain the true clock frequency
nInBufferSize	Size in bytes of the divisor index

20.7.1.7 I2C_IOCTL_SET_FREQUENCY

This DeviceIoControl request estimates the nearest clock rate acceptable for I^2C device and initialize the I^2C 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

lpInBuffer	Pointer to the desired I^2C frequency
nInBufferSize	Size in bytes of the I^2C frequency requested

20.7.1.8 I2C_IOCTL_SET_MASTER_MODE

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

20.7.1.9 I2C_IOCTL_SET_SELF_ADDR

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

Parameters

lpInBuffer	Pointer to the expected I^2C device address, valid range is [0x00–0x7F]
nInBufferSize	Size in bytes of the I^2C device address
Remarks	The device expects to respond when any master on the I^2C bus wishes to proceed with any transfer. This IOCTL has no effect if the I^2C device is in Master mode.

20.7.1.10 I2C_IOCTL_SET_SLAVE_MODE

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

20.7.1.11 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

20.7.1.12 I2C_IOCTL_ENABLE_SLAVE

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

20.7.1.13 I2C_IOCTL_DISABLE_SLAVE

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

20.7.1.14 I2C_IOCTL_GET_SLAVESIZE

This DeviceIoControl request gets the interface buffer size of the I^2C device for slave mode.

20.7.1.15 I2C_IOCTL_SET_SLAVESIZE

This DeviceIoControl request sets the interface buffer size of the I^2C device for slave mode. The maximum size for the buffer is configured by I2CSLAVEBUFSIZE.

20.7.1.16 I2C_IOCTL_GET_SLAVE_TXT

This DeviceIoControl request gets the current data from interface buffer of the I^2C device for slave mode. Both slave device or external master can change this data.

20.7.1.17 I2C_IOCTL_SET_SLAVE_TXT

This DeviceIoControl request sets data to interface buffer of the I^2C device for slave mode. An external I^2C master can get this data immediately from driver after it connects the slave.

20.7.2 I^2C Driver SDK Encapsulation

20.7.2.1 I2COpenHandle

This function retrieves the I^2C device handle.

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

Parameters

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

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

20.7.2.2 I2CCloseHandle

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

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

Parameters

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

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

20.7.2.3 I2CSetSlaveMode

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

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

Parameters

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

20.7.2.4 I2CSetMasterMode

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

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

Parameters

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

20.7.2.5 I2CIsMaster

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

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

Parameters

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

20.7.2.6 I2CIsSlave

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

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

Parameters

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

20.7.2.7 I2CGetClockRate

This function retrieves the clock rate divisor. This value is not the absolute peripheral clock frequency. The value retrieved should be compared against the I^2C specifications to obtain the true frequency.

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

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pwClkRate	Pointer of WORD variable that retrieves divisor index. Refer to I^2C specification to obtain the true clock frequency
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

20.7.2.8 I2CSetClockRate

This function initializes the I^2C device with the given clock rate. This function does not expect to receive the absolute peripheral clock frequency. Rather, it expects the clock rate divisor index stated in the I^2C specification. If absolute clock frequency must be used, use the function I2CSetFrequency().

```
BOOL I2CSetClockRate(
        HANDLE hDev,
        WORD wClkRate);
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
wClkRate	Divisor index. Refer to I^2C specification to obtain the true clock frequency
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

20.7.2.9 I2CSetFrequency

This function estimates the nearest clock rate acceptable for I^2C device and initializes the I^2C device to use the estimated clock rate divisor. If the estimated clock rate divisor index is required, refer to the macro I2CGetClockRate to determine the estimated index.

```
BOOL I2CSetFrequency(
        HANDLE hDev,
        DWORD dwFreq);
```

Parameters

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

20.7.2.10 I2CSetSelfAddr

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

```
BOOL I2CSetSelfAddr(
```

```
        HANDLE hDev,
        BYTE bySelfAddr);
```

Parameters

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

20.7.2.11 I2CGetSelfAddr

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

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

Parameters

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

20.7.2.12 I2CTransfer

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

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

Parameters

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

20.7.2.13 I2CReset

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

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

Parameters

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

20.7.2.14 I2CEnableSlave

This function enables a I^2C slave access from the bus. After the I^2C slave interface is enabled, the I^2C slave driver waits for an external master access.

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

Parameters

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

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

20.7.2.15 I2CDisableSlave

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

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

Parameters

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

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

20.7.2.16 I2CGetSlaveSize

This function returns the I^2C slave interface buffer length. The I^2C slave driver directly returns data to the master from the interface buffer. The interface buffer can be set at any time, even when the I^2C slave module has been turned off.

```
BOOL I2CGetSlaveSize(
        HANDLE hDev,
        PDWORD pdwSize);
```

Parameters

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

pdwSize — Pointer to DWORD variable that retrieves interface buffer length

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

20.7.2.17 I2CSetSlaveSize

This function sets the I^2C slave interface buffer length. The maximum acceptable length is I2CSLAVEBUFSIZE. If input length is longer than I2CSLAVEBUFSIZE, the operation fails, and the original buffer length is not changed. The I^2C slave driver directly returns data to the master from the interface buffer. The interface buffer can be set at any time, even when the I^2C slave module has been turned off.

```
BOOL I2CSetSlaveSize(
        HANDLE hDev,
        DWORD dwSize);
```

Parameters

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

dwSize	DWORD variable that sets interface buffer length
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

20.7.2.18 I2CGetSlaveText

This function returns the I^2C slave interface buffer text. The I^2C slave driver directly returns data to the master from the interface buffer. The interface buffer can be accessed at any time, even when the I^2C slave module has been turned off.

```
BOOL I2CGetSlaveText(
        HANDLE hDev,
        PBYTE pbyTextBuf,
        DWORD dwBufSize,
        PDWORD pdwTextLen );
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pbyTextBuf	User buffer to store text returned from interface buffer
pdwBufSize	User buffer size
pdwTextLen	Actual data bytes returned
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

20.7.2.19 I2CSetSlaveText

This function returns the I^2C slave interface buffer text. The I^2C slave driver directly returns data to the master from the interface buffer. The interface buffer can be accessed at any time, even when the I^2C slave module has been turned off.

```
BOOL I2CSetSlaveText(
        HANDLE hDev,
        PBYTE pbyTextBuf,
        DWORD dwTextLen );
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pbyTextBuf	User buffer to store text to interface buffer
dwTextLen	Text length in user buffer
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

20.7.3 I^2C Driver Structures

20.7.3.1 I2C_PACKET

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

```
typedef struct {
        BYTE byAddr;
        BYTE byRW;
        PBYTE pbyBuf;
        WORD wLen;
```

```
        LPINT lpiResult;
} I2C_PACKET, *PI2C_PACKET;
```

Members

byAddr 7-bit slave address that specifies the target I^2C device to or from which data is 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. Set to I2C_POLLING_MODE to force polling mode for transfer.

pbyBuf Pointer to a buffer of bytes. For a read operation, this is the buffer into which data is read. For a write operation, this buffer contains the data to write to the target device.

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

20.7.3.2 I2C_TRANSFER_BLOCK

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

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

Members

pI2CPackets Pointer to an array of I2C_PACKET objects

iNumPackets Number of I2C_PACKET objects pointed to by pI2CPackets

Chapter 21
MediaLB Device Module (MLB)

The MediaLB module implements the Physical Layer and Link Layer of the MediaLB specification, interfacing the hardware platform to the MediaLB controller.

21.1 MLB Summary

The MLB implements the 3-pin MediaLB mode and runs at speeds up to 1024FS. It does not implement the MediaLB controller functionality, and only works as the MLB device. Windows CE provides driver support for MLB device using stream interface architecture. The interface driver exports functions to the file system. Users can operate the MLB functions using IOCTL or read/write functions. Table 21-1 provides a summary of source code location, library dependencies and other BSP information.

Table 21-1. MLB Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\MLB
Driver DLL	mlb.dll
SDK Library	N/A
Catalog Item(s)	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > MLB > MLB
SYSGEN Dependency	N/A
BSP Environment Variable(s)	BSP_MLB = 1

21.2 Supported Functionality

The MLB driver enables the i.MX35-3DS board to provide the following software and hardware support:

1. Supports the Windows CE stream interface driver.
2. Supports the IC test bench environment from SMSC.
3. Supports the data processing that is received and transmitted

21.3 Hardware Operation

21.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts

21.4 Software Operation

The stream interface driver is described in the Wince600 Help. MLB is developed in the stream interface architecture.

21.4.1 Compile-Time Configuration Options

N/A.

21.4.2 Registry Settings

21.4.2.1 i.MX35 Registry Settings

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

21.4.3 DMA Support

21.4.3.1 i.MX35 DMA Support

The MLB module supports internal DMA transfer when receiving or transmitting data. This is realized in the MLB internal operation. However, for software design, the registers that point to the address of the internal RAM have to be set with the physical address. If not, the receiving or transmitting cannot succeed.

21.4.4 IOCTL

21.4.4.1 i.MX35 IOCTL

For the stream interface driver, IOCTL is used to implement the specific operations. MLB driver provides several IOCTLs.

21.4.4.1.1 MLB_CTL_256FS_SWLOOP

This IOCTL is used to test the MLB at a speed of 256FS in control mode, using the software loop-back test environment provided by SMSC.

21.4.4.1.2 MLB_CTL_512FS_SWLOOP

This IOCTL is used to test the MLB at a speed of 512FS in control mode, using the software loop-back test environment provided by SMSC.

21.4.4.1.3 MLB_ASYN_256FS_SWLOOP

This IOCTL is used to test the MLB at a speed of 256FS in asynchronous mode, using the software loop-back test environment provided by SMSC.

21.4.4.1.4 MLB_ASYN_512FS_SWLOOP

This IOCTL is used to test the MLB at a speed of 512FS in asynchronous mode, using the software loop-back test environment provided by SMSC.

21.4.4.1.5 MLB_CTL_READ

This IOCTL is used to read the data that is received in the control mode.

21.4.4.1.6 MLB_CTL_WRITE

This IOCTL is used to write the data that is to be transmitted in the control mode.

21.4.4.1.7 MLB_ASYN_READ

This IOCTL is used to read the data that is received in the asynchronous mode.

21.4.4.1.8 MLB_ASYN_WRITE

This IOCTL is used to write the data that is to be transmitted in the asynchronous mode.

21.5 Power Management

21.5.1 i.MX35 Power Management

N/A.

21.6 Unit Test

The MLB driver is tested using the Test Bench from SMSC using the following applications:

- Software Loop-Back Test at speeds of 256/512FS in control mode
- Software Loop-Back Test at speeds of 256/512FS in asynchronous mode

21.6.1 Unit Test Hardware

Table 21-2 lists the required hardware to run the unit tests.

Table 21-2. Hardware Requirements

Requirement	Description
Test Bench from SMSC	Demo board and I/O board to work as Controller and Device in the MLB environment

21.6.2 Unit Test Software

An application tests the MLB driver and is used in the Test Bench environment. The application opens the driver and operates different functions using IOCTL. The application is located at:

\WINCE600\SUPPORT_PDK1_7\TEST\MLBTest

Users can change the codes to send different IOCTLs.

21.6.3 Building the Unit Tests

In order to build the MLB test, complete the following steps:

Build an OS image for the desired configuration

- Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This opens a DOS prompt.
- Change to the MLBTest directory: \WINCE600\SUPPORT\TEST\MLBTest
- Enter **set WINCEREL=1** on the command prompt and hit return. This copies the built exe to the flat release directory
- Input **build -c** command to build MLBTest

After the building completes, the mlb.exe file is located in the $(_FLATRELEASEDIR) directory.

21.6.4 Running the Unit Tests

Copy mlb.exe to the device using SD/MMC cards or other modes. Click the mlb.exe and run the test.

Chapter 22
Micro Controller Unit (MCU) Driver

The MCU drivers is a stream interface driver that directly controls the hardware components provided by Freescale microcontroller unit MC9S08DZ60.

22.1 MCU Driver Summary

The MCU description can be found in the MC9S08DZ60 datasheet and HW MCU design documentation. This document is intended for device driver and application developers who need to understand and gain access to the functionality provided by the MCU. Table 22-1 provides a summary of source code location, library dependencies and other BSP information.

Table 22-1. MCU Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*TargetPlatform*>\SRC\DRIVERS\MCU\MC9S08DZ60\PDK ..\PLATFORM\<*TargetPlatform*>\SRC\DRIVERS\MCU\MC9S08DZ60\SDK
Driver DLL	mcupdk_mc9s08dz60.dll mcusdk_mc9s08dz60.dll
SDK Library	mcusdk_mc9s08dz60.lib
Catalog Item	Third Party **>** BSP **>** Freescale <*Target Platform*>:ARMV4I **>** Device Drivers **>** MCU
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_MCU= 1

22.2 Supported Functionality

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

1. Supports the RTC function
2. Supports GPIO extension
3. Supports keypad scanning
4. Supports touch screen control
5. Supports power management

22.3 Hardware Operation

The i.MX35 device uses the I2C1 port to interact with the MCU as shown in Figure 22-1 and the I2C1 signals are selected in the IOMUX. GPIO1_0 is used as MCU GPIO interrupt

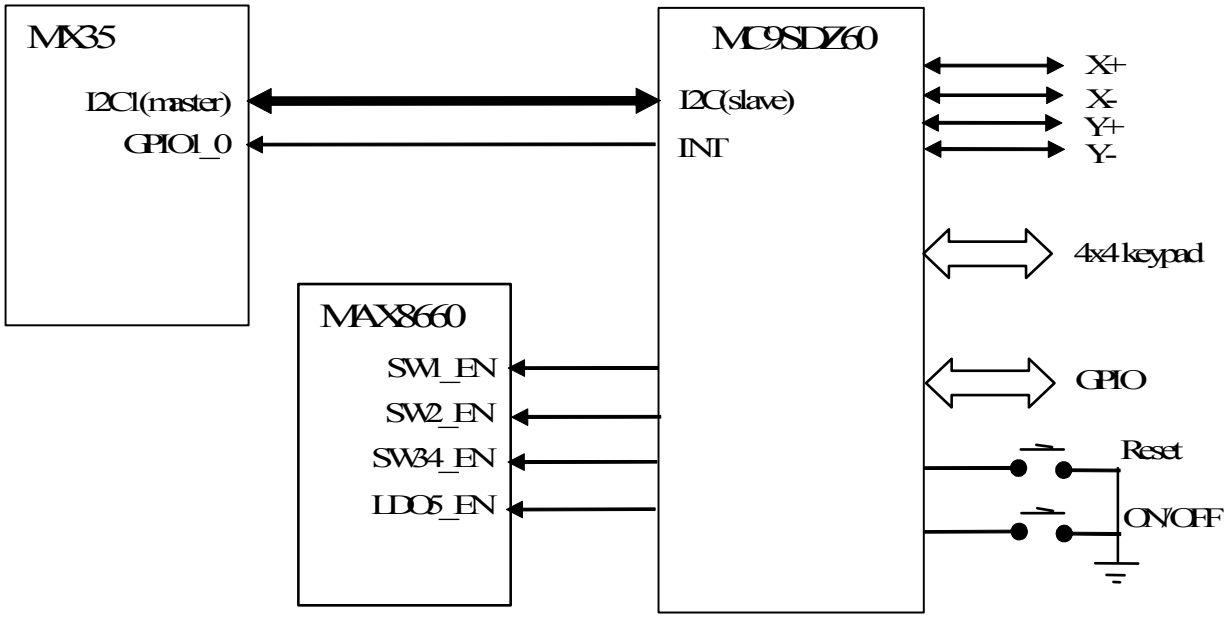

Figure 22-1. i.MX35 and MCU Hardware Connections

22.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

22.4 Software Operation

The MCU driver modules can be used by applications or device drivers. For example, the GPIO function is used for the FEC driver to control the FEC external transceiver power enable pin. The MCU 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 MCU Stream Interface driver controls the MCU hardware directly via the I2C bus. The Stream Interface driver provides an IOCTL interface for MCU SDK DLLs. The MCU SDK DLL provide APIs for WindowsCE drivers and applications. We access the MCU only need use the MCU SDK DLL.

22.4.1 MCU Driver Function

Table 22-2. MCU Driver Functions

Function	Description
McuRegisterRead	Reads a register in MCU
McuRegisterWrite	Writes a register in MCU
McuInterruptRegister	Initializes a interrupt in MCU
McuInterruptDeregister	Uninitializes a interrupt in MCU.
McuInterruptHandlingComplete	This function handle Complete a interrupt in MCU
McuInterruptDisable	Disables an interrupt
McuInterruptEnable	Enables an interrupt
McuGpioEnable	Enables an GPIO in MCU
McuGpioReset	Enables an GPIO Reset
McuTouchInit	Initializes the MCU touch controller
McuTouchDeinit	Uninitializes the MCU touch controller
McuADCTouchRead	Reads touch screen sample
McuTouchIsPressed	Get touch panel is pressed Status
McuKeybdInit	Initializes the MCU key interface
McuKeybdDeinit	Uninitializes the MCU key interface
McuKeybdRead	Scan the press key data

22.4.2 MCU RTC Function

OAL RTC function use KernelIoControl to access the MCU driver to get RTC.We implement four basic KernelIoControl for exchange RTC.

```
{ IOCTL_HAL_RTC_INIT, 0,  OALIoCtlInitRTC },
{ IOCTL_HAL_RTC_QUERY, 0,  OALIoCtlHalRtcQuery },
{ IOCTL_HAL_RTC_SYNC, 0,  OALIoCtlHalRtcSync },
{ IOCTL_HAL_RTC_ALARM, 0,  OALIoCtlHalRtcAlarm },
```

22.4.3 Registry Settings

MCU driver only need a standard stream interface driver registry settings.

```
; These registry entries load the MCU driver. The IClass value is GUID for generic power-managed
devices.
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\MCU]
   "Prefix"="MCU"
   "Dll"=”mcupdk_mc9s08dz60.dll"
   "Index"=dword:1
   "Order"=dword:3
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

22.5 Power Management

22.5.1 PowerUp

This function is not implemented for the MCU driver.

22.5.2 PowerDown

This function is not implemented for the MCU driver.

22.5.3 IOCTL_POWER_CAPABILITIES

We advertise the power management capabilities with power manager through this IOCTL. The MCU module supports only two power states: D0 and D4.

22.5.4 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 MCU driver. Any request that is not D0 is changed to a D4 request results in the system entering into suspend state. For a value of D0, the system is resumed. For all platforms, the following registry entry must be defined:

```
"IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

22.5.5 IOCTL_POWER_GET

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

22.6 Unit Test

The MCU driver is used by an application or by other drivers. Therefore, other driver are used to test the MCU driver function.

22.6.1 RTC Function

Use OAL RTC CEKT function to test the RTC function.

22.6.1.1 Unit Test Hardware

Only need the 3-Stack board.

22.6.1.2 Unit Test Software

Select the MCU RTC catalog item for the RTC function test. Figure 22-3 lists the required software to run the unit tests.

Table 22-3. Software Requirements

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

22.6.1.3 Running the Unit Tests

The command line for running the tests is:

```
tux -o -d oalTestTimers.dll -x 1260-1290
```

22.6.1.4 MCU Touch Driver

Use the touch driver CEKT to test the MCU touch driver function.

Chapter 23
NAND Flash Driver

The NAND Flash Driver provides the functionality of NAND storage accessing. The flash driver follows Windows CE 6.0 R2 Flash driver MDD and PDD model.

23.1 Flash Driver Summary

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

The flash driver supports both SLC and MLC NAND Flash devices. As for page size, 512 byte (small page size) is not supported. Table 23-1 provides a summary of source code location, library dependencies and other BSP information.

Table 23-1. Flash Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\NAND
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\NAND
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\BLOCK\NANDFMD ..\PLATFORM\<*Target Platform*>\SRC\COMMON\NANDFMD
Driver DLL	flashpdd_nand.dll
SDK Library	N/A
Catalog Item	Device Drivers > Storage Devices > MSFlash Drivers > Flash MDD Third Party > BSP > Freescale i.MX35 3DS PDK1_7: ARMV4I > Storage Drivers > MSFlash Drivers > K9LBG08U0D NAND Flash
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NONAND= BSP_NAND_K9LAG08U0M=1 BSP_NAND_K9LBG08U0M=1 BSP_NAND_K9LBG08U0D=1

23.2 Supported Functionality

The flash driver enables the 3-Stack System to provide the following software and hardware support:

1. Supports the Windows CE MDD/PDD interface
2. Supports both MLC and SLC NAND
3. Supports both 2 Kbyte and 4 Kbyte page size NAND
4. Supports MLC NAND Flash K9LBG08U0D as default

23.3 Hardware Operation

Refer to the chapter on the NAND Flash Controller (NFC)in the *i.MX35 Multimedia Applications Processor Reference Manual* for detailed operation and programming information.

23.3.1 Conflicts with Other Peripherals and Catalog Items

23.3.1.1 Conflicts with SoC Peripherals

No conflicts.

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

23.4.1 MDD/PDD Layer Overview

The Microsoft Windows Embedded CE 6.0 Flash Driver component contains two components: the module device driver (MDD), and the platform-dependent driver (PDD).

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

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

The block diagram below (Figure 38-1) describes the high level architecture and basic interactions of the i.MX NAND driver implementation. The i.MX flash driver PDD consists of three major components:

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

The i.MX flash driver currently supports a limited number of commercially available flash modules. However, the i.MX flash driver software architecture allows support of new flash modules. The i.MX flash driver must be modified to support new flash modules that the BSP does not currently support.

The i.MX flash driver is table driven. That is, by appropriately modifing definitions described in this chapter, the flash driver can be reconfigured to support a different flash module. No other source code changes are required.

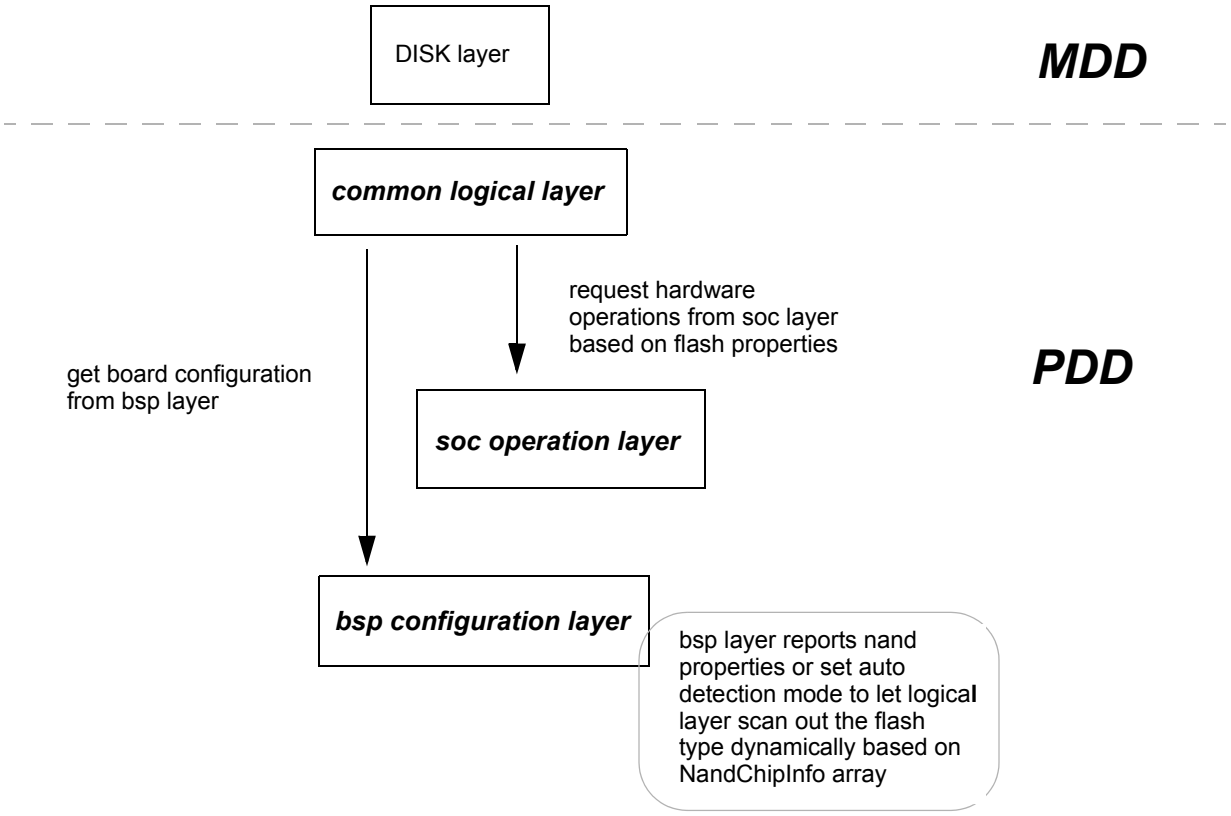

Figure 23-1. PDD Layer Block Diagram

23.4.2 Definitions

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

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

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

The i.MX flash driver abstracts the characteristics of the NAND Flash memory device to a single header file. This header file is named in terms of the NAND device name, could be found in:

\WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7**\NAND\INC**

For each NAND flash device, there are two header files, one is .h format used for C language code and another is .inc format used for assembly language code. The two files are highly similar.

Table 23-2. Nand Property Definitions

Member	Description
CMD_READID	Command used to read nand flash ID. Default value is 0x90.
CMD_READ	Command used as initial command for reading operation. Default value is 0x00.
CMD_READ2	Command used as start command for reading operation. Default value is 0x30.
CMD_RESET	Command used to reset nand flash. Default value is 0xFF.
CMD_WRITE	Command used as initial command for writing operation. Default value is 0x80.
CMD_WRITE2	Command used as start command for writing operation. Default value is 0x10.
CMD_ERASE	Command used as initial command for erasing operation. Default value is 0x60.
CMD_ERASE2	Command used as start command for erasing operation. Default value is 0xD0.
CMD_STATUS	Command used to read nand flash status. Default value is 0x70.
NAND_STATUS_BUSY_BIT	Bit number in status byte to indicate BUSY/IDLE status of nand flash status. Default value is 6.
NAND_STATUS_ERROR_BIT	Bit number in status byte to indicate PASS/FAIL status of nand flash operation. Default value is 0.
NAND_NUM_OF_CS	Number of dies per nand flash chip.
NAND_BLOCK_CNT	Number of blocks per nand flash die.
NAND_PAGE_CNT	Number of pages per block.

Table 23-2. Nand Property Definitions (continued)

Member	Description
NAND_PAGE_SIZE	Number of bytes in main data area per page.
NAND_SPARE_SIZE	Number of bytes in spare area per page.
NAND_BUS_WIDTH	Bit number of nand flash, it should be 8bits or 16bits.
NAND_MAKER_CODE	Nand ID defined in terms of manufacturer.
NAND_DEVICE_CODE	Nand ID defined in terms of nand type.
BBI_NUM	Number of pages, defined by manufacturer, that is used to indicate initial bad block during manufacturing.
BBMarkPage[BBI_NUM]	An array that indicates which pages are used to indicate initial bad block during manufacturing.
BBI_MAIN_ADDR	Byte address that is used to swap data with bad block mark to guarantee BBI position is untouched. For 2K+64B page size nand, define it as 464; For 4K+128B page size nand, define it as 400; For 4K+218B page size nand, define it as 330.
NAND_PAGE_CNT_LSH	The power exponent (log2) of NAND_PAGE_CNT. For exsample, it should be 7 if page number is 128 (2^7). The definition is defined in .inc file.
NAND_PAGE_SIZE_LSH	The power exponent (log2) of NAND_PAGE_SIZE. For exsample, it should be 12 if page size is 4096 (2^12). The definition is defined in .inc file.
NAND_BLOCK_SIZE_LSH	The power exponent (log2) of nand flash block size. The definition is defined in .inc file.
NUM_OF_NAND_DEVICES	Number of nand flash dies per nand flash chip. The definition is defined in .inc file.
NUM_OF_NAND_DEVICES_LSH	The power exponent (log2) of NUM_OF_NAND_DEVICES. The definition is defined in .inc file.

23.4.3 Adding New Flash Configurations

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

The flash driver currently supports the flash modules listed in section 38.2. The flash driver does not dynamically detect which flash module is installed. Instead, the flash driver must be modified and re-compiled to support a different flash module

To support a flash module that is not currently supported by the BSP, use the following steps:

1. create two new header files (.h and .inc format files) by using one of the existing NAND device headers as templates
2. update the newly created header file to include device-specific definitions described in 38.4.2
3. update the reference in the `nandbsp.h` device-specific header located in the following directory:
   `\WINCE600\PLATFORM\<Target Platform>\SRC\COMMON\NANDFMD`
4. recompile the flash driver for the new device

NOTE

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

23.4.4 Registry Settings

The registry keys implemented for the flash driver provide basic support for loading and configuring the NAND as a file system mount. Many 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 Storage Management** > **Storage Management** > **Storage Manager Registry Settings**

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

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

The two items can be deleted to disable the functionality.

23.4.5 DMA Support

The flash driver does not support DMA.

23.4.6 Power Management

Flash driver handles power requests in MDD layer by default.

23.5 Unit Test

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

23.5.1 CETK Testing

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 23-3. CEKT Tests

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

Table 23-3. CEKT Tests

Storage Device Block Diver API Test	tux -o -d disktest -c"/profile MSFlash /zorch /part /sectors 256"
Storage Device Block Diver Benchmark Test	tux -o -d rw_all -c"/profile MSFlash /zorch /part"
Storage Device Block Diver Read/Write Test	tux -o -d rwtest -c"/profile MSFlash /zorch /part"

23.5.2 System Testing

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

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

Chapter 24
Power Management IC (PMIC)

24.1 PMIC Summary

This chapter provides information to develop:

- Device drivers that interface directly to the Freescale power management IC (PMIC) hardware components. The PMIC that is specifically referenced in this document is the MC13892.
- Applications that use the special hardware capabilities that are provided by the PMIC (for example, touch I/O, BackLight function.).

This chapter describes the API provided by Freescale which allows complete access to the 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. Table 24-1 provides a summary of source code location, library dependencies and other BSP information.

Table 24-1. PMIC Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\PMIC\MC13892
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\PMIC\MC13892
Driver DLL	pmicPdk_mc13892.dll
SDK Library	pmicSdk_mc13892.lib
Catalog Item(s)	N/A
SYSGEN Dependency	N/A
BSP Environment Variable(s)	BSP_PMIC_MC13892 = 1

24.2 Supported Functionality

The PMIC device driver framework for Windows CE is a stream interface driver and a SDK DLL. A description of the stream interface driver may be found in the Windows CE Platform Builder documentation at **Developing a Device Driver** > **Windows CE Drivers** > **Stream Interface Drivers**.

The PMIC Stream Interface driver controls the PMIC hardware directly using the SPI or I^2C bus. The Stream Interface driver provides an IOCTL interface for SDK DLLs. The SDK DLLs provide APIs for Windows CE drivers and applications.

The API covers the PMIC functionality of the following areas:

1. Register Access
2. Tri-Color LED
3. Battery
4. Regulators
5. Keys (Power, PTT)
6. ADC /Touch
7. Backlight (Keyboard, LCD)
8. Battery Charger
9. GPO

24.3 Hardware Operation

Refer to the MC13892 document for details on the MC13892 PMIC.

24.3.1 Conflicts with Other On-Chip Peripherals

24.3.1.1 i.MX35 Peripheral Conflicts

No Conflicts.

24.3.2 Conflicts with Other 3-Stack Peripherals

No conflicts.

24.4 Software Operation

24.4.1 Configuring the PMIC

The PMIC modules can be used by applications or device drivers. For example, the battery API of the PMIC is used by the battery driver. Configuring the PMIC port for communications involves some basic operations. A handle to the desired PMIC port must be opened prior to accessing the module registers. This handle is required to call the **DeviceIoControl** function. The function parameters include the PMIC port handle, appropriate IOCTL code, and other input and output parameters.

24.4.2 Creating a Handle to the PMIC

Before calling any PMIC API make sure that the PMIC device is attached by calling the **CreateFile** function which opens a file and it returns a handle that can be used to access the MC13892 hardware. If the MC13892 hardware does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the PMIC:

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.

```
// Open the PMIC port.
hPMI =   CreateFile(TEXT("PMI1:"),
         GENERIC_READ | GENERIC_WRITE,                // access (read-write) mode
         FILE_SHARE_READ | FILE_SHARE_WRITE,                    // sharing mode
         NULL,                                     // security attributes (ignored)
         OPEN_EXISTING// sharing mode                      // creation disposition
         FILE_FLAG_RANDOM_ACCESS,                          //flags and attributes
         NULL);                                      // template file (ignored)
if ((hPMI == NULL) || (hPMI == INVALID_HANDLE_VALUE))
{
         ERRORMSG(1, (_T("Failed in create File()\r\n")));
}
```

24.4.3 Write Operations

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

24.4.4 Read Operations

Like the write operation, the PMIC driver does not provides for reading through the PMIC_Read function. This is a stub function and always returns success.

24.4.5 Closing the Handle to the PMIC

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

24.4.6 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 need to be accessed and then disabled once it is done.

24.4.6.1 PowerUp

This function is not implemented for the PMIC driver.

24.4.6.2 PowerDown

This function is not implemented for the PMIC driver.

24.4.6.3 IOCTL_POWER_CAPABILITIES

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

24.4.6.4 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** in the PMIC driver. Any request that is not D0 is changed to a D4 request and results in the system entering into suspend state. For a request of value of D0, the system is resumed.

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

24.4.7 PMIC Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PMI]
   "Prefix"="PMI"
   "Dll"="pmicpdk_mc13892.dll"
   "Index"=dword:1
   "Order"=dword:2
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"     ; PMCLASS_GENERIC_DEVICE
```

24.4.8 DMA Support

No support.

24.5 Unit Test

24.5.1 Unit Test Hardware

The 3-Stack and the MC13892 PMIC boards are required.

24.5.2 Unit Test Software

No software is necessary for this test.

24.5.3 Running the PMIC Tests

The PMIC driver can be tested using the following actions:

- The command line for running the PMIC tests is `s i2cpmic`
- Use the touch driver CEKT to test MC13892 touch function
- Use the battery driver CEKT to test MC13892 battery function
- Use the backlight driver CEKT to test MC13892 backlight function

24.6 PMIC Driver API Reference

24.6.1 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 with in the API developed for specific modules of the PMIC device. Most of the IOCTLs are explained in the specific sections where they are more relevant.

24.6.1.1 PMIC_IOCTL_LLA_READ_REG

This DeviceIoControl request reads the register content.

Parameters

hPMI	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
lpInBuffer	Index of the register
lpOutBuffer	[out] Long pointer to a buffer that receives the output data for the operation. Set to NULL if the dwIoControlCode parameter specifies an operation that does not produce output data

24.6.1.2 PMIC_IOCTL_LLA_WRITE_REG

This DeviceIoControl request writes the data to the said register of the PMIC device.

Parameters

hPMI	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
lpInBuffer	Index of the register
lpOutBuffer	Pointer to data which needs to be written to the register

24.6.1.3 PMIC_IOCTL_LLA_INT_REGISTER

This DeviceIoControl is used to register interrupt.

Parameters

hPMI	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
lpInBuffer	Index of the register
lpOutBuffer	Pointer to event name and interrupt ID

Code example:

```
param.int_id = int_id;
param.event_name = event_name;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_REGISTER, &param,sizeof(param), NULL, 0,
NULL, NULL);
```

24.6.1.4 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)
```

24.6.1.5 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);
```

24.6.1.6 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);
```

24.6.1.7 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);
```

24.6.2 Interrupt Handling

This section describes the interrupt handling of the PMIC driver.

24.6.2.1 Interrupt Handling Overview

The PMIC has interrupt generation capability to inform the CPU when events occur. This is signaled to the processors driving the SPI or I^2C buses. 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 is invoked by the kernel once the kernel receives an interrupt from the PMIC. This IST first queries 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 then sets a predefined event. For any drivers and applications that need notification of an interrupt, they must register the interrupt and wait for the event. They also need to reset the event after handling the event.

24.6.2.2 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 triggers the event when the corresponding interrupt occurs.

24.6.2.3 PMIC Interrupt Events

Table 24-2 shows the events and corresponding MC13892 interrupts.

Table 24-2. PMIC Interrupt Events

PMIC Interrupt	Description
ADCDONEI	ADC has finished requested conversions
ADCBISDONEI	ADCBIS has finished requested conversions
TSI	Touch screen wakeup
VBUSVALIDI	VBUSVALID detect
IDFACTORYI	ID factory mode detect
USBOVI	USB over-voltage detection
CHGDETI	Charger attach
CHGFAULTI	Charger fault detection
CHGREVI	Charger path reverse current
CHGSHORTI	Charger path short circuit
CCCVI	Charger path CC / CV transition detect
CHGCURRI	Charge current below threshold warning
BPONI	BP turn on threshold
LOBATLI	Low battery low threshold warning
LOBATHI	Low battery high threshold warning
IDFLOATI	USB ID float detect
IDGNDI	USB ID ground detect
1HZI	1 Hz time tick
TODAI	Time of day alarm
PWRON3I	PWRON3 event
PWRON1I	PWRON1 event
PWRON2I	PWRON2 event
WDIRESETI	WDI system reset event
SYSRSTI	PWRON system reset event
RTCRSTI	RTC reset event
PCI	Power cut event

Table 24-2. PMIC Interrupt Events (continued)

WARMI	Warm start event
MEMHLDI	Memory hold event
LPBI	Low power USB boot detection
THWARNLI	Thermal warning low threshold
THWARNHI	Thermal warning high threshold
CLKI	Clock source change
SCPI	Short circuit protection trip detection
BATTDETBI	Battery removal detect

24.6.2.4 Interrupt Data Structures

```
typedef enum _PMIC_MC13892_INT_ID {
    PMIC_MC13892_INT_ADCDONEI = 0,
    PMIC_MC13892_INT_ADCBISDONEI = 1,
    PMIC_MC13892_INT_TSI = 2,
    PMIC_MC13892_INT_VBUSVALIDI = 3,
    PMIC_MC13892_INT_IDFACTORYI = 4,
    PMIC_MC13892_INT_USBOVI = 5,
    PMIC_MC13892_INT_CHGDETI = 6,
    PMIC_MC13892_INT_CHGFAULTI = 7,
    PMIC_MC13892_INT_CHGREVI = 8,
    PMIC_MC13892_INT_CHGSHORTI = 9,
    PMIC_MC13892_INT_CCCVI = 10,
    PMIC_MC13892_INT_CHGCURRI = 11,
    PMIC_MC13892_INT_BPONI = 12,
    PMIC_MC13892_INT_LOBATLI = 13,
    PMIC_MC13892_INT_LOBATHI = 14,
    PMIC_MC13892_INT_IDFLOATI = 19,
    PMIC_MC13892_INT_IDGNDI = 20,
    PMIC_MC13892_INT_1HZI = 32,
    PMIC_MC13892_INT_TODAI = 33,
    PMIC_MC13892_INT_PWRON3I = 34,
    PMIC_MC13892_INT_PWRON1I = 35,
    PMIC_MC13892_INT_PWRON2I = 36,
    PMIC_MC13892_INT_WDIRESETI = 37,
    PMIC_MC13892_INT_SYSRSTI = 38,
    PMIC_MC13892_INT_RTCRSTI = 39,
    PMIC_MC13892_INT_PCI = 40,
    PMIC_MC13892_INT_WARMI = 41,
    PMIC_MC13892_INT_MEMHLDI = 42,
    PMIC_MC13892_INT_LPBI = 43,
    PMIC_MC13892_INT_THWARNLI = 44,
    PMIC_MC13892_INT_THWARNHI = 45,
    PMIC_MC13892_INT_CLKI = 46,
    PMIC_MC13892_INT_SCPI = 48,
    PMIC_MC13892_INT_BATTDETBI = 54,
    PMIC_INT_MAX_ID
} PMIC_INT_ID;
```

24.6.2.5 Interrupt Functions

Table 24-3 shows the interrupt functions.

Table 24-3. Interrupt Functions

Function	Description
PmicInterruptRegister	Register the interrupt if the interrupt is to be enabled
PmicInterruptDeregister	Deregisters an interrupt
PmicInterruptHandlingComplete	Completion of a interrupt handling, enable an interrupt
PmicInterruptDisable	Disables an interrupt
PmicInterruptEnable	Reenable an interrupt

24.6.3 Register Access API

The PMIC Low Level Access API allows drivers and applications to read and write PMIC registers. There are some restrictions to prohibit drivers and application from accessing some registers. Interrupt registers is one example. The interrupt library functions are in this Low Level Access DLL.

24.6.3.1 Read Register

This function reads a PMIC register.

Prototype

PMIC_STATUS PmicRegisterRead(unsigned char index, UINT32* reg);

Parameters

index [in] register index

reg [out] The contents of the register

Return Value Status code

24.6.3.2 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

24.6.4 Power Control Reference

24.6.4.1 Power Control Function

This section provides information about MC13892 power control module. The API MC13892 Power control module can be accessed using the functions shown in Table 24-4.

Table 24-4. Power Control Functions

Function	Usage
PmicPwrctrlSetPowerCutTimer	Set the power cut timer duration
PmicPwrctrlGetPowerCutTimer	Get the power cut timer duration
PmicPwrctrlEnablePowerCut	Enable the power cut
PmicPwrctrlDisablePowerCut	Disable the power cut
PmicPwrctrlSetPowerCutCounter	Set the power cut counter
PmicPwrctrlGetPowerCutCounter	Get the power cut counter
PmicPwrctrlSetPowerCutMaxCounter	Set the maximum number of power cut counter
PmicPwrctrlGetPowerCutMaxCounter	Get the setting of maximum power cut counter
PmicPwrctrlEnableCounter	Enable the power counter
PmicPwrctrlDisableCounter	Disable the power counter
PmicPwrctrlEnableClk32kMCU	Enable the CLK32KMCU
PmicPwrctrlDisableClk32kMCU	Disable the CLK32KMCU
PmicPwrctrlEnableDRM	Set Keeps VSRTC and CLK32KMCU on for all states
PmicPwrctrlDisableDRM	Disable Keeps VSRTC and CLK32KMCU on for all states
PmicPwrctrlEnableUSEROFFCLK	Enable Keeps VSRTC and CLK32KMCU during user off
PmicPwrctrlDisableUSEROFFCLK	Disable VSRTC and CLK32KMCU during user off
PmicPwrctrlEnablePCUTEXPB	E nable PCUTEXPB=1 at a startup event
PmicPwrctrlDisablePCUTEXPB	Disable PCUTEXPB=1 at a startup event
PmicPwrctrlEnableUserOffModeWhenDelay	Place the phone in User Off Mode after a delay
PmicPwrctrlDisableUserOffModeWhenDelay	Set not to place the phone in User Off Mode after a delay
PmicPwrctrlEnableWarmStart	Warm start enable
PmicPwrctrlDisableWarmStart	Warm start disable
PmicPwrctrlEnablePWRONRESET	System reset on PWRON pin
PmicPwrctrlDisablePWRONRESET	Disable system reset on PWRON pin
PmicPwrctrlSetDebtime	Set debounce time on PWRON pin
PmicPwrctrlEnableSTANDBYINV	Set STANDBY is interpreted as active low
PmicPwrctrlDisableSTANDBYINV	Set disable STANDBY is interpreted as active not low
PmicPwrctrlEnableSTANDBYSECINV	Set disable STANDBYSEC is interpreted as active low

Table 24-4. Power Control Functions (continued)

Function	Usage
PmicPwrctrlDisableSTANDBYSECINV	Disable STANDBYSEC is interpreted as active not low
PmicPwrctrlEnableWDIRESET	Enable system reset through WDI
PmicPwrctrlDisableWDIRESET	Disable system reset through WDI
PmicPwrctrlSetSPIDRV	Set SPI drive strength
PmicPwrctrlGetSPIDRV	Get SPI drive strength
PmicPwrctrlSetCLK32KDRV	Set CLK32K and CLK32KMCU drive strength
PmicPwrctrlGetCLK32KDRV	Get the CLK32K and CLK32KMCU drive strength
PmicPwrctrlSetSTBYDLY	Set Standby delay
PmicPwrctrlGetSTBYDLY	Get the Standby delay
PmicPwrctrlGetMODES	Get the MODE sense
PmicPwrctrlGetI2CS	Get the I2CS mode
PmicPwrctrlGetPUMSS	Get the PUMSS mode

24.6.4.2 Power Control Data Structures

```
typedef enum _MC13892_PWRCTRL_PWRON{
    PWRON1=0,
    PWRON2,
    PWRON3,
} MC13892_PWRCTRL_PWRON;

typedef enum _MC13892_PWRCTRL_MODES{
    MODES_GROUNDED=0,
    MODES_RESEVED,
    MODES_VCOREDIG,
    MODES_VCORE,
} MC13892_PWRCTRL_MODES;

typedef enum _MC13892_PWRCTRL_I2CS{
    SPI=0,
    I2C,
} MC13892_PWRCTRL_I2CS;

typedef enum _MC13892_PWRCTRL_PUMSS{
    PUMSS_GROUNDED=0,
    PUMSS_OPEN,
    PUMSS_VCOREDIG,
    PUMSS_VCORE,
} MC13892_PWRCTRL_PUMSS;
```

24.6.5 Buck Switchers and Linear Regulators

This section provides information about control MC13892 buck switchers and linear regulators.

24.6.5.1 Functions

```
PMIC_STATUS PmicSwitchModeRegulatorOn (PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorOff (PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorGetVoltageLevel (PMIC_REGULATOR_SREG regulator,
            PMIC_REGULATOR_SREG_VOLTAGE_TYPE voltageType,PMIC_REGULATOR_SREG_VOLTAGE*voltage);
PMIC_STATUS PmicSwitchModeRegulatorSetMode (PMIC_REGULATOR_SREG
            regulator,PMIC_REGULATOR_SREG_STBY standby,PMIC_REGULATOR_SREG_MODE mode);
PMIC_STATUS PmicSwitchModeRegulatorGetMode (PMIC_REGULATOR_SREG regulator,
            PMIC_REGULATOR_SREG_STBY standby,PMIC_REGULATOR_SREG_MODE* mode);
PMIC_STATUS PmicSwitchModeRegulatorEnableSTBYDVFS (PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorDisableSTBYDVFS (PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorSetDVSSpeed (PMIC_REGULATOR_SREG regulator, UINT8dvsspeed);
PMIC_STATUS PmicSwitchModeRegulatorSetSidLevel (PMIC_REGULATOR_SREG regulator, UINT8
            hilevel,UINT8 lowlevel);
PMIC_STATUS PmicSwitchModeRegulatorGetSidLevel(PMIC_REGULATOR_SREG regulator, UINT8*
            hilevel,UINT8* lowlevel);
PMIC_STATUS PmicSwitchModeRegulatorSetPLLMF (UINT8 mf);
PMIC_STATUS PmicSwitchModeRegulatorEnableHIRANGE(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorDisableHIRANGE(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorEnableMemoryHoldMode(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorDisableMemoryHoldMode(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorEnableUserOffMode(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorDisableUserOffMode(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorEnableSIDMode();
PMIC_STATUS PmicSwitchModeRegulatorDisableSIDMode();
PMIC_STATUS PmicSwitchModeRegulatorEnablePLL();
PMIC_STATUS PmicSwitchModeRegulatorDisablePLL();
PMIC_STATUS PmicSwitchModeRegulatorEnableSWBST();
PMIC_STATUS PmicSwitchModeRegulatorDisableSWBST();
PMIC_STATUS PmicVoltageRegulatorOn (PMIC_REGULATOR_VREG regulator);
PMIC_STATUS PmicVoltageRegulatorOff (PMIC_REGULATOR_VREG regulator);
PMIC_STATUS PmicVoltageRegulatorSetVoltageLevel (PMIC_REGULATOR_VREG regulator,
            PMIC_REGULATOR_VREG_VOLTAGE voltage);
PMIC_STATUS PmicVoltageRegulatorGetVoltageLevel (PMIC_REGULATOR_VREG regulator,
            PMIC_REGULATOR_VREG_VOLTAGE* voltage);
PMIC_STATUS PmicVoltageRegulatorSetPowerMode (PMIC_REGULATOR_VREG regulator,
            PMIC_REGULATOR_VREG_POWER_MODE powerMode);
PMIC_STATUS PmicVoltageRegulatorGetPowerMode (PMIC_REGULATOR_VREG regulator,
            PMIC_REGULATOR_VREG_POWER_MODE* powerMode);
PMIC_STATUS PmicVoltageGPOOn (MC13892_GPO_SREG gpo);
PMIC_STATUS PmicVoltageGPOOff (MC13892_GPO_SREG gpo);
```

24.6.6 Backlight and Led

This section provides information about control MC13892 backlight system and signaling LEDs.

24.6.6.1 Backlight and LED Functions

```
PMIC_STATUS PmicBacklightEnableHIMode(BACKLIGHT_CHANNEL channel);
PMIC_STATUS PmicBacklightDisableHIMode(BACKLIGHT_CHANNEL channel);
```

```
PMIC_STATUS PmicBacklightEnableRamp(BACKLIGHT_CHANNEL channel);
PMIC_STATUS PmicBacklightDisableRamp(BACKLIGHT_CHANNEL channel);
PMIC_STATUS PmicBacklightSetCurrentLevel(BACKLIGHT_CHANNEL channel, UINT8 level);
PMIC_STATUS PmicBacklightGetCurrentLevel(BACKLIGHT_CHANNEL channel, UINT8* level);
PMIC_STATUS PmicBacklightSetDutyCycle(BACKLIGHT_CHANNEL channel, UINT8 cycle);
PMIC_STATUS PmicBacklightGetDutyCycle(BACKLIGHT_CHANNEL channel, UINT8* cycle);
PMIC_STATUS PmicLEDIndicatorEnableRamp(LED_CHANNEL channel);
PMIC_STATUS PmicLEDIndicatorDisableRamp(LED_CHANNEL channel);
PMIC_STATUS PmicLEDIndicatorSetCurrentLevel(LED_CHANNEL channel, unsigned char level);
PMIC_STATUS PmicLEDIndicatorGetCurrentLevel(LED_CHANNEL channel, unsigned char* level);
PMIC_STATUS PmicLEDIndicatorSetDutyCycle(LED_CHANNEL channel, unsigned char dc);
PMIC_STATUS PmicLEDIndicatorGetDutyCycle(LED_CHANNEL channel, unsigned char* dc);
PMIC_STATUS PmicLEDIndicatorSetBlinkPeriod(LED_CHANNEL channel, unsigned char bp);
PMIC_STATUS PmicLEDIndicatorGetBlinkPeriod(LED_CHANNEL channel, unsigned char* bp);
PMIC_STATUS PmicLEDIndicatorEnableSWBST();
PMIC_STATUS PmicLEDIndicatorDisableSWBST();
```

24.6.6.2 Backlight and Led Data Structures

```
typedef enum _BACKLIGHT_CHANNEL {
    BACKLIGHT_MAIN_DISPLAY,
    BACKLIGHT_AUX_DISPLAY,
    BACKLIGHT_KEYPAD
} BACKLIGHT_CHANNEL;
typedef enum _LED_CHANNEL {
    TCLED_RED,
    TCLED_GREEN,
    TCLED_BLUE
} LED_CHANNEL;
```

24.6.7 ADC and Touch Controller

24.6.7.1 ADC and Touch Controller Function

```
PMIC_STATUS PmicADCInit(void);
PMIC_STATUS PmicADCGetSingleChannelOneSample(UINT16 channel, UINT16 * pResult);
PMIC_STATUS PmicADCGetSingleChannelEightSamples(UINT16 channel, UINT16 * pResult);
PMIC_STATUS PmicADCGetMultipleChannelsSamples(UINT16 channels, UINT16 * pResult);
PMIC_STATUS PmicADCGetHandsetCurrent(PMIC_ADC_CONVERTOR_MODE mode, UINT16 *pResult);
PMIC_STATUS PmicADCTouchRead(UINT16* x, UINT16* y);
PMIC_STATUS PmicADCTouchStandby(BOOL intEna);
void        PmicADCDeinit(void);
```

24.6.7.2 ADC and Touch Controller Data Structures

```
typedef enum _MC13892_TOUCH_MODE {
    TM_INACTIVE = 0,
    TM_INTERRUPT,
    TM_TOUCHSCREEM
} MC13892_TOUCH_MODE;
typedef MC13892_TOUCH_MODE PMIC_TOUCH_MODE;

typedef enum _PMIC_ADC_CONVERTOR_MODE
{
    ADC_8CHAN_1X = 0, // RAND = 0, 8 channels, 1 sample
```

```
    ADC_1CHAN_8X      // RAND = 1, 1 channel, reads 8 sequential values
} PMIC_ADC_CONVERTOR_MODE;
```

24.6.8 Battery Charger

This section provides information about control MC13892 battery charger system.

24.6.8.1 Battery Charger Functions

```
PMIC_STATUS PmicBatterEnableCharger(BATT_CHARGER chgr, UINT8 c_voltage, UINT8 c_current);
PMIC_STATUS PmicBatterDisableCharger(BATT_CHARGER chgr);
PMIC_STATUS PmicBatterSetCharger(BATT_CHARGER chgr, UINT8 c_voltage, UINT8 c_current);
PMIC_STATUS PmicBatterGetChargerSetting(BATT_CHARGER chgr, UINT8* c_voltage, UINT8*c_current);
PMIC_STATUS PmicBatterGetChargeCurrent(UINT16* c_current);
PMIC_STATUS PmicBatterLedControl(BOOL on);
PMIC_STATUS PmicBatterSetReverseSupply(BOOL enable);
PMIC_STATUS PmicBatterSetUnregulated(BOOL enable);
```

24.6.8.2 Battery Charger 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,
    DUAL_INPUT_DUAL_PATH,
    INVALID_CHARGER_MODE
}CHARGER_MODE;
```

Chapter 25
Serial Driver

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

25.1 Serial Driver Summary

The serial port driver is implemented as a stream interface driver and supports all the standard I/O control codes and entry points. The serial port driver handles all the internal UARTs except UART1 which is used for debugging. In the BSP implementation, the hardware-specific code that corresponds to the serial port driver lower layer is implemented as the platform-dependent driver (PDD). This PDD is linked with Microsoft-provided public serial MDD library (com_mdd2.lib) to form the whole serial port driver. Table 25-1 provides a summary of source code location, library dependencies and other BSP information.

Table 25-1. Serial Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\SERIAL
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\SERIAL
Driver DLL	csp_serial.dll
SDK Library	N\A
Catalog Item	Third Party —> BSP —> Freescale <*Target Platform*>: ARMV4I — > Device Drivers —> Serial —> UART2serial port support Third Party —> BSP —> Freescale <*Target Platform*>: ARMV4I —> Device Drivers —> Serial —> UART3 serial port support
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_SERIAL_UART2 =1 BSP_SERIAL_UART3 =1

NOTE

Do not select UART3 and FEC function in PB catalog window on the TO2.0 blue CPU board, because UART3 and FEC share some pins.

25.2 Supported Functionality

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

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

25.3 Hardware Operation

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

25.3.1 Conflicts with Other Peripherals and Catalog Items

25.3.1.1 Conflicts with SoC Peripherals

All the pins of UART3 can be configured for alternate functionality (FEC, ATA, SD2) using the IOMUX. The configuration is specified by BSP serial driver. Changing this configuration results in a conflict and prevents proper operation of the UART3. In the TO2 blue CPU board, GPIO1_5 is used to switch UART3 and FEC pins function.

25.3.1.2 Conflicts with Board Peripherals

No conflicts.

25.4 Software Operation

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

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

25.4.1 Registry Settings

25.4.1.1 i.MX35 Registry Settings

```
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:4
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="i.MX35 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
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:4
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM3\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="i.MX35 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
```

25.4.2 Power Management

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

25.5 Unit Test

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

25.5.1 Unit Test Hardware

- i.MX35 3DS board

25.5.2 Unit Test Software

Table 25-2 lists the required software to run the unit tests.

Table 25-2. Software Requirements

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

25.5.3 Building the Unit Tests

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

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

25.5.4 Running the Unit Tests

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

For detailed information on the Serial Port tests, see

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

The serial port tests are designed to test that the serial port driver works properly and the API behaves correctly, and it should be pass all the test cases. Table 25-3 describes the Serial Port driver test cases.

Table 25-3. Serial Port Driver Test Cases

Test Case	Description
1001	Configures the port and writes data to the port at all possible baud rates, data bits, parities, and stop bits. This test fails if it cannot send data on the port with a particular configuration.
1002	Tests the **SetCommEvent** and **GetCommEvent** functions. This test fails if the driver does not properly support the **SetCommEvent** or **GetCommEvent** functions.
1003	Tests the **EscapeCommFunction** function. This test fails if the driver does not support one of the Microsoft Win32 **EscapeCommFunction** functions.
1004	Tests the **WaitCommEvent** function on the EV_TXEMPTY event. The test creates a thread to send data and waits for the EV_TXEMPTY event to occur when the thread finishes sending data. This test fails if the **WaitCommEvent** function behaves improperly or if the EV_TXEMPTY event does not signal appropriately.
1005	Tests the **SetCommBreak** and **ClearCommBreak** functions. This test fails if the driver does not properly support the **SetCommBreak** or **ClearCommBreak** functions.
1006	Makes the **WaitCommEvent** function return a value when the handle for the current COM port is cleared. This test fails if the **WaitCommEvent** function behaves improperly.
1007	Makes the **WaitCommEvent** function return a value when the handle for the current COM port is closed. This test fails if the **WaitCommEvent** function behaves improperly.

Table 25-3. Serial Port Driver Test Cases

Test Case	Description
1008	Tests the **SetCommTimeouts** function and verifies that the **ReadFile** function properly times out when no data is received. This test fails if the COM timeouts do not function correctly.
1009	Verifies that previous Device Control Block (DCB) settings are preserved when the **SetCommState** function call fails with DCB settings that are not valid. This test fails if the serial port driver does not keep previous DCB settings when DCB settings that are not valid are passed to the driver.

25.6 Serial Driver API Reference

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

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

25.6.1 Serial PDD Functions

Table 25-4 shows a mapping of Serial PDD functions to the functions used in the serial driver.

Table 25-4. Serial PDD Functions

PDD Function Pointer	Serial Driver Function
HWInit	SerSerialInit
HWPostInit	SerPostInit
HWDeinit	SerDeinit
HWOpen	SerOpen
HWClose	SerClose
HWGetIntrType	SL_GetIntrType
HWRxIntrHandler	SL_RxIntrHandler
HWTxIntrHandler	SL_TxIntrHandler
HWModemIntrHandler	SL_ModemIntrHandler
HWLineIntrHandler	SL_LineIntrHandler
HWGetRxBufferSize	SL_GetRxBufferSize
HWPowerOff	SerPowerOff
HWPowerOn	SerPowerOn
HWClearDTR	SL_ClearDTR
HWSetDTR	SL_SetDTR
HWClearRTS	SL_ClearRTS
HWSetRTS	SL_SetRTS

Table 25-4. Serial PDD Functions

PDD Function Pointer	Serial Driver Function
HWEnableIR	SerEnableIR
HWDisableIR	SerDisableIR
HWClearBreak	SL_ClearBreak
HWSetBreak	SL_SetBreak
HWXmitComChar	SL_XmitComChar
HWGetStatus	SL_GetStatus
HWReset	SL_Reset
HWGetModemStatus	SL_GetModemStatus
HWGetCommProperties	SerGetCommProperties
HWPurgeComm	SL_PurgeComm
HWSetDCB	SL_SetDCB
HWSetCommTimeouts	SL_SetCommTimeouts

25.6.2 Serial Driver Structures

25.6.2.1 UART_INFO

This structure contains information about the UART Module.

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

Members

pUartReg	Pointer to UART Hardware registers
sUSR1	This value contains the UART status register
sUSR2	This value contains the UART status register
bDSR	This boolean value keeps the DSR state
UartType	This value contains the type of UART like DCE or DTE
UlDiscard	This is used to discard the echo characters in IrDa Mode
UseIrDA	This boolean value determines the driver is in IR mode or not
HwAddr	This value contains the hardware address of the UART Module
EventCallback	This is a callback to the Model Device Driver
pMDDContext	This contains the context of the UART, which is the first parameter to the callback function
dcb	This value contains the copy of Device Control Block
CommTimeouts	This contains the copy of CommTimeouts structure used to get and set the time-out parameters for a communication device
pBaudTable	Pointer to baud rate table
DroppedBytes	This value contains the number of bytes dropped
FlushDone	Handle to the flush done event
CTSFlowOff	This boolean value is used to store the CTS flow control state
DSRFlowOff	This boolean value is used to Store the DSR flow control state
AddTXIntr	This boolean value is used to fake a Tx interrupt
Status	This value contains the comm status
CommErrors	This value contains Win32 comm error status
ModemStatus	This value shows the Win32 Modem status
TransmitCritSec	This value is used as Critical Section for UART registers
RegCritSec	This value is used as Critical Section for UART
ChipID	This value contains Chip identifier (CHIP_ID_16550 or CHIP_ID_16450)

25.6.2.2 SER_INFO

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

```
typedef struct __SER_INFO {
    UART_INFO    uart_info;
    BOOL    fIRMode;
    DWORD    dwDevIndex;
    DWORD    dwIOBase;
    DWORD    dwIOLen;
    PCSP_UART_REG pBaseAddress;
    UINT8    cOpenCount;
```

```
    COMMPROP    CommProp;
    PHWOBJ    pHWObj;
    BOOL        useDMA;
    DDK_DMA_REQ     SerialDmaReqTx;
    DDK_DMA_REQ     SerialDmaReqRx;
    PHYSICAL_ADDRESS SerialPhysTxDMABufferAddr;
    PHYSICAL_ADDRESS SerialPhysRxDMABufferAddr;
    PBYTE       pSerialVirtTxDMABufferAddr;
    PBYTE       pSerialVirtRxDMABufferAddr;
    UINT8       SerialDmaChanRx;
    UINT8       SerialDmaChanTx;
    UINT8       currRxDmaBufId;
    UINT8       currTxDmaBufId;
    UINT        dmaRxStartIdx;
    UINT        availRxByteCount;
    UINT32      awaitingTxDMACompBmp;
    UINT32      dmaTxBufFirstUseBmp;
    UINT16      rxDMABufSize;
    UINT16      txDMABufSize;
} SER_INFO, *PSER_INFO;
```

Members

uart_info	This structure contains information about UART
fIRMode	This boolean value determines the module is FIR or serial
dwDevIndex	This static value contains the device index value which is read from registry
dwIOBase	This static value contains the I/O Base address of UART module which is read from registry
dwIOLen	This static value contains the I/O length of UART Module which is read from registry
pBaseAddress	Pointer to the start address of the UART registers mapped
cOpenCount	Contains count of the concurrent open
CommProp	Pointer to CommProp structure
pHWObj	Pointer to PDDs HWObj structure
useDMA	This boolean flag indicates if SDMA is to be used for transfers through this UART
SerialDmaReqTx	SDMA request line for Tx
SerialDmaReqRx	SDMA request line for Rx
SerialPhysTxDMABufferAddr	Physical address of Tx SDMA address
SerialPhysRxDMABufferAddr	Physical address of Rx SDMA address
pSerialVirtTxDMABufferAddr	Virtual address of Tx SDMA address
pSerialVirtRxDMABufferAddr	Virtual address of Rx SDMA address.
SerialDmaChanRx	SDMA virtual channel indices for Rx
SerialDmaChanTx	SDMA virtual channel indices for Tx

currRxDmaBufId	Index of the buffer descriptor next expected to complete its SDMA in the Rx SDMA buffer descriptor chains
currTxDmaBufId	Index of the buffer descriptor next expected to complete its SDMA in the Tx SDMA buffer descriptor chains
dmaRxStartIdx	Keeps the start index of byte to be delivered to MDD for Read
availRxByteCount	This variable keeps the remaining bytes in the Rx SDMA buffer
awaitingTxDMACompBmp	Indicates if an SDMA request is in progress on Tx SDMA buffer descriptor
dmaTxBufFirstUseBmp	Indicator for first time use of a Tx SDMA buffer descriptor
rxDMABufSize	Receive DMA buffer size
txDMABufSize	Transfer DMA buffer size

Chapter 26
Sony/Philips Digital Interface (SPDIF) Driver

The Sony/Philips Digital Interface (SPDIF) audio module is a stereo transceiver that allows the processor to receive and transmit digital audio.

26.1 SPDIF Driver Summary

The SPDIF driver module (`spdifdev.dll`) provides receiver (RX) and transmitter (TX) functions as a waveform audio driver. For more information about the waveform audio driver, see the Platform Builder Help topic:

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

Table 26-1 provides the source code location, library dependencies, and other BSP information.

Table 26-1. SPDIF Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\SPDIFDEV
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\SPDIF
Driver DLL	spdifdev.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale<*Target Platform*>:ARMV4I > Device Drivers > SPDIF > SPDIF
SYSGEN Dependency	SYSGEN_AUDIO
BSP Environment Variables	BSP_NOAUDIO= BSP_SPDIF=1

26.2 Supported Functionality

The SPDIF driver enables the board to provide the following software and hardware support:

1. Conforms to the Microsoft audio driver architecture as defined for Windows Embedded CE 6.0 and all related operating systems
2. Supports Freescale hardware platforms that include the SPDIF module

3. Double-buffered DMA operations to transfer audio data between memory and the SPDIF TX/RX FIFO
4. Two power management modes, full on and full off
5. PCM data and compressed data transmission according with IEC958 spec
6. Both TX and RX function, support 44.1 Kbyte, 48 Kbyte sample rate

26.2.1 Conflicts with Other Peripherals and Catalog Items

26.2.1.1 Conflicts with SoC Peripherals

The SPDIF controller shares pins with the CSPI2 module. The SPDIF-specific signals are AUD5_TXD, AUD5_RXD, AUD5_SCK.

26.2.1.2 Conflicts with board Peripherals

No conflicts

26.2.2 Known Issues

The SPDIF driver may cause the audio playback driver CETK to fail for MSFT CETK fault. To run the audio playback driver CETK, remove the SPDIF driver from the catalog temporarily or run the AudioRouting application to select Audio Output/Input as the default device.

26.3 Software Operation

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

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

26.3.1 SPDIF Transmitter (TX)

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

26.3.2 SPDIF Receiver (RX)

The operation of the SPDIF driver for receiving is similar to the hardware configuration. Once the hardware components are configured, the audio driver handles the input DMA buffer full interrupts. This is done via the interrupt handler, which copies the contents of each input DMA buffer to an

application-supplied buffer, and then returns the empty DMA buffer to the SDMA controller. If the application-supplied buffer does not have enough space for all of the new data, any extra data is discarded. The application is signaled using a callback function when the application-supplied buffer is full. The SPDIF driver also picks-up C Channel and U Channel information, so the application can query these when need.

26.3.3 Compile-Time Configuration Options

Table 26-2 shows the compile-time configuration options.

Table 26-2. SPDIF Driver Configuration Options (hwctxt.cpp)

Configuration Setting Name	Description
AUDIO_DMA_PAGE_SIZE	The size in bytes of each DMA buffer. Default is 6144 bytes.
SPDIF_SFCSR_TX_WATERMARK	The transmitter watermarks that are to be used with SPDIF TX FIFO. The default is 16.
SPDIF_SFCSR_RX_WATERMARK	The receiver watermarks that are to be used with SPDIF RX FIFO. The default is 16.
SPDIF_TX_ENABLED	Enable/Disable SPDIF TX module bye define/undef this macro.
SPDIF_RX_ENABLED	Enable/Disable SPDIF RX module bye define/undef this macro.

26.3.4 Registry Settings

At least one registry key must be properly defined so that the Device Manager loads the SPDIF driver when the system is booted. The following registry keys are required in order for the Device Manager to properly load the SPDIF device driver during the normal device boot process. These registry settings should typically not be modified. If they are missing or incorrectly defined, then the SPDIF driver may not be loaded and all SPDIF functions are disabled.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SPDIF]
"Prefix"="WAV"
"Dll"="spdifdev.dll"
"Index"=dword:1
"Order"=dword:7
;"Priority256"=dword:99
"IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"       ; PMCLASS_GENERIC_DEVICE
```

26.3.5 DMA Support

As indicated previously, the SPDIF driver uses the SDMA controller to transfer the digital audio data between the audio application and the RX/TX FIFOs. This minimizes the processing required by the core and can also reduce the power consumption during SPDIF transmitting and receiving operations. This section describes the SPDIF driver DMA implementation issues and trade-offs, and the available compile-time DMA-related configuration options.

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

- The DMA data buffers where the application data is kept

- The DMA buffer descriptors, which are used by the DMA hardware to manage the state of each DMA buffer

The DMA data buffers can be allocated from either internal memory (which is provided by on-chip internal RAM) or external memory (which is provided by off-chip external DRAM). The issues and considerations for the type of memory to use for the DMA data buffers is as follows:

- Internal memory region:
  - 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.
  - Total size of the internal memory region is limited.
  - The limited amount of internal memory may have to be shared by multiple device drivers.
  - The entire internal memory region must be manually managed with predefined addressed ranges being reserved for each specific use.
- External memory region:
  - The total size of the external memory is typically much greater than the size of the internal memory. This provides much greater flexibility in selecting the size of the DMA data buffers.
  - There is typically no need to worry about the possible impact and memory requirements of any other device driver.
  - Memory allocation is handled using the standard Windows Embedded CE 6.0 system calls.
  - The external memory cannot be placed into a low power mode while the DMA is active.

The build configuration options such that the SPDIF driver allocates its DMA data buffers from either internal or external memory are as follows:

- Internal memory region—Set the BSP_SPDIF_DMA_BUF_ADDR macro in `bsp_cfg.h` to an address within the internal memory region. Also set BSP_SPDIF_DMA_BUF_SIZE to the total size (in bytes) for all DMA data buffers that are allocated.
- External memory region—Comment out the BSP_SPDIF_DMA_BUF_ADDR macro in `bsp_cfg.h`

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 API, specifically DDKSdmaAllocChain(). See the Chapter 10, “Chip Support Package Driver Development Kit (CSPDDK),” for additional information about the DDKSdmaAllocChain() API.

26.4 Power Management

The primary method for limiting power consumption in the SPDIF driver is to gate off all clocks to the SPDIF when those clocks are not needed and set SPDIF to lower power mode. This is accomplished through the **DDKClockSetGatingMode** function call and the SPDIF related register setting. The clock gating and the disabling of the SPDIF is handled automatically within the SPDIF module and requires no additional configuration or code changes. The SPDIF driver operates correctly after resuming from the power down mode.

26.4.1 PowerUp

This function resumes an SPDIF I/O operation that was previously terminated by calling the PowerDown() API. It begins by restoring power and then it restarts the DMA transfers to complete the powerup process for the SPDIF driver. This function is intended to be called only by the Power Manager and must not block or depend on any hardware interrupts. Therefore, all required timed delays must be handled by using a polling loop instead of any of the normal wait for an event to be signalled functions. This functionality is currently handled by IOCTL_POWER_SET and the function is just a stub.

26.4.1.1 i.MX35 PowerUp Support

Power enable clock and leave SPDIF from lower-power mode.

26.4.2 PowerDown

This function suspends all currently active SPDIF I/O operations just before the entire system enters the low power state. This function is intended to be called only by the Power Manager and must not block or depend on any hardware interrupts. This functionality is currently handled by IOCTL_POWER_SET and the function is just a stub.

26.4.2.1 i.MX35 Power Down Support

Power gating clock and set SPDIF to lower-power mode.

26.4.2.2 IOCTL_POWER_SET

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

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

26.5 Unit Test

26.5.1 Unit Test Hardware

Table 26-3 lists the required hardware to run the unit tests.

Table 26-3. Hardware Requirements

Requirement	Description
M-Audio Card on PC	M-Audio Card to send/receive SPDIF digital data
Audio Daughter Card	SPDIF input/output interface on the card

26.5.2 Unit Test Software

Table 26-4 lists the required software to run the unit tests.

Table 26-4. Software Requirements

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

26.5.3 Building the Unit Tests

To build the SPDIF tests, build an OS image for the desired configuration using the following steps:

1. Within Platform Builder, choose **Build OS** > **Open Release Directory**.
   A DOS prompt is displayed.
2. Change to the SPDIF Tests directory: `\WINCE600\SUPPORT_PDK1_7\TEST\SPDIF`
3. Enter **set WINCEREL=1** on the command prompt and hit return.
   This copies the built DLL to the flat release directory.
4. Input **build -c** at the prompt and press return.

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

26.5.4 Running the Unit Tests

The command line for running the SPDIF tests is:

```
tux –o –d spdiftest.dll -n
```

To redirect the test results to a file, add the option **–f**. The SPDIF tests do not contain any test-specific command line options.

26.6 System Testing

In addition to running the SPDIF driver tests in the CETK, simple applications can be developed to perform various system-level tests that involve the use of the SPDIF driver. For example, a small modification can be made to WAVPLAY and WAVEREC to test the SPDIF TX and RX functions (Windows CE sample application source code located in `WINCE600\PUBLIC\COMMON\SDK\SAMPLES\AUDIO`).

```
pwfx->wFormatTag = WAVE_FORMAT_WMASPDIF; // SPDIF FORMAT
```

For perform this testing, a SPDIF transmitter and receiver device which can be used to send or receive audio data to/from i.MX353DS board is required, such as an M-Audio USB card (which can be connected to the PC by the USB port).

The TX path should be connected as follows:

M-Audio optical port [in] <—> line dual-optical interface <—> i.MX35 SPDIF TX optical port

The connection diagram of RX as following:

M-Audio optical port [out] <—> line dual-optical interface <—> i.MX35 SPDIF RX optical port

Then Spectralab, WaveLab (which can be find in M-Audio software disc) can be used capture/play audio data from/to 3DSSPDIF device.

26.7 SPDIF Driver API Reference

SPDIF driver is a standard waveform audio driver. For detailed reference information for the SPDIF driver, see the Platform Builder Help:

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

Chapter 27
Touch Panel Driver

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

27.1 Touch Panel Driver Summary

Table 27-1 provides a summary of source code location, library dependencies and other BSP information.

Table 27-1. Touch Panel Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\TOUCH
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\TOUCH
Driver DLL	touch.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX35 3DS PDK1_7:ARMV4I > Device Drivers > TOUCH > TI TSC2007 > for TI TSC 2007 Third Party > BSP > Freescale i.MX35 3DS PDK1_7:ARMV4I > Device Drivers > TOUCH > Freescale MC9S08DZ60 Microcontroller > for MCU Third Party > BSP > Freescale i.MX35 3DS PDK1_7:ARMV4I > Device Drivers > TOUCH > MC13892 TOUCH > for MC13892 PMIC Touch
SYSGEN Dependency	SYSGEN_TOUCH = 1
BSP Environment Variables	BSP_NOTOUCH= BSP_TOUCH_TSC2007=1 ->for TI TSC 2007BSP_TOUCH_MCU9S08DZ60=1 ->for MCU BSP_TOUCH_MC13892=1 ->for MC13892 PMIC Touch

27.2 Supported Functionality

The touch panel should conform to the standards as explained in documentation under:

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

27.3 Hardware Operations

The hardware consists of a LCD Panel with a touch screen and a TI TSC2007 touch controller. The I^2C module sends control information to the TSC2007 and reads back the touch samples. More details about the I^2C can be found in Chapter 20, "Inter-Integrated Circuit (I2C) Driver."

The hardware also consists of a LCD Panel with touch screen and the MCU. The MCU driver module sends control commands and reads back the touch samples. More details about the MCU driver can be found in the MCU driver chapter.

The hardware also consists of a LCD panel with a touch screen and the MC13892. The MC13892 touch screen driver sends control commands and reads back the touch samples. More details about MC13892 driver can be found in Chapter 24, "Power Management IC (PMIC)."

27.3.1 Conflicts with SOC Peripherals

The Touch Driver requires a timer to provide the necessary timings between different touch samples. Therefore, the EPIT2 is dedicated for Touch Panel and cannot be used by any other module.

27.4 Software Operations

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

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

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

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

For every calibration point, perform the following steps:

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

27.4.1 Touch Driver Registry Settings

```
IF BSP_NOTOUCH !
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]
    "DriverName"="touch.dll"
```

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

[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]
    "MaxCalError"=dword:7
"CalibrationData"="539,520 280,259 280,778 793,781 794,259"

; For Touch Panel calibration. Note that the Windows Mobile PocketPC touch panel
; calibration is handled automatically by the welcome.exe application so a
; separate "Launch" registry key is not required. Also, Windows Mobile
; SmartPhone does not support a touch panel at all which means that this is
; not required for SmartPhone either.

[HKEY_LOCAL_MACHINE\init]
    "Launch80"="touchc.exe"
    "Depend80"=hex:14,00, 1e,00
ENDIF    ; BSP_NOTOUCH !
```

27.5 Unit Tests

27.5.1 Unit Test Hardware

Table 27-2 lists the required hardware to run the unit tests.

Table 27-2. Hardware Requirements

Requirement	Description
LCD panel	Display panel required for display of graphics data

27.5.2 Unit Test Software

Table 27-3 lists the required software to run the unit tests.

Table 27-3. Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
Ktux.dll	Ktux.dll which is required to run in kernel mode
Touchtest.dll	The Test.dll File
Touch.dll	Touch Panel Driver

NOTE

The touch driver does work after the CETK Touch Panel Test. This is a known MSFT CETK issue. The MSFT online help notes that "When you complete the test, the operating system does not regain control of the touch panel. You must reset the touch panel to restore normal operation." Refer to **CETK Tests and Test Tools** > **CETK Tests** > **Touch Panel Tests**

Cases 8011, 9001–9003 fail. The touch panel shows several lines when a circle or arc is drawn. This is also a known MSFT CETK issue. All these points are captured, but are not painted in time.

Case 8011 cannot draw in the right part of screen after a 90° rotation. ethca.exe works after rotation and the CETK works when the case runs again.

27.5.3 Running the Touch Panel Tests

The touch panel test cases can be run by typing:

```
tux -o -n -d touchtest.dll -x <Test case id>
```

The test case IDs are described in the documentation at:

Windows Embedded CE Test Kit > **CETK Tests and Test Tools** >**CETK Tests** > **Touch Panel Tests** > **Touch Panel Test**

27.6 Touch Panel API Reference

The complete API reference is given in the documentation at:

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

Chapter 28
Universal Serial Bus (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.MX. 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 a set of three drivers: USB OTG host controller driver, USB client driver and/or USB transceiver controller (Full Function) driver, which performs the host/function client switching.

The USB host driver can be configured for class support for mass storage, HID, printer, and RNDIS peripherals. The device/client portion can be configured to provide 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 Section 28.1, "USB OTG Driver Summary." and Section 28.2, "USB Host Driver Summary").

28.1 USB OTG Driver Summary

28.1.1 USB OTG Client Driver Summary

Table 28-1 provides a summary of source code location, library dependencies and other BSP information for the USB OTG client driver.

Table 28-1. OTG Client Driver Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
Common SOC	COMMON_FSL_V2
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\USBD ..\PLATFORM\COMMON\SRC\SOC\<*Common Soc*>\ms\USBFN
CSP Static Library	usb_usbfn_<*Target SOC*>_PDK1_7.lib usb_usbfn_os_<*Target SOC*>_PDK1_7.lib usb_ufnmddbase_<*Common Soc*>_PDK1_7.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBD
Import Library	N/A
Driver DLL	usbfn.dll

Table 28-1. OTG Client Driver Summary (continued)

Driver Attribute	Definition
Catalog Item	High Speed OTG: Third Party > BSP > Freescale <*Target Platform*>: 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_NOUSB= BSP_USB_HSOTG_CLIENT=1

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 28.5.5, "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.

28.1.2 OTG Host Driver Summary

Table 28-2 provides a summary of source code location, library dependencies and other BSP information for the USB OTG host driver.

Table 28-2. OTG Host Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX35-3DS-PDK1_7
Target SOC (TGTSOC)	MX35_FSL_V2_PDK1_7
Common SOC	COMMON_FSL_V2
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<*Common SOC*>\ms\USBH\EHCI ..\PLATFORM\COMMON\SRC\SOC\<*Common SOC*>\ms\USBH\EHCIPDD ..\PLATFORM\COMMON\SRC\SOC\<*Common SOC*>\ms\USBH\USB2COM
CSP Static Library	usbh_ehcdmdd_<*Common SOC*>_PDK1_7.lib usbh_ehcdpdd_<*Common SOC*>_PDK1_7.lib usbh_usb2com_<*Common SOC*>_PDK1_7.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBH\HSOTG
Import Library	N/A
Driver DLL	hcd_hsotg.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device To support only host mode, choose .. >High Speed OTG Port Pure Host Function.
SYSGEN Dependency	SYSGEN_USB=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSOTG_HOST=1

Host driver requires a set of class drivers to be loaded. See Section 28.5.6, "Class Drivers," for class driver information.

28.1.3 OTG Transceiver Driver Summary (For High-Speed Only)

Table 28-3 provides a summary of source code location, library dependencies and other BSP information for the USB OTG transceiver driver.

Table 28-3. OTG Transceiver Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX35-3DS-PDK1_7
Target SOC (TGTSOC)	MX35_FSL_V2_PDK1_7
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\USBXVR
CSP Static Library	usb_xvc_<*Target SOC*>_PDK1_7.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBXVR
Import Library	N/A
Driver DLL	imx_xvc.dll
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device > High Speed OTG Port Full OTG Function Support
SYSGEN Dependency	SYSGEN_USBFN=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSOTG_CLIENT=1 BSP_USB_HSOTG_HOST=1 BSP_USB_HSOTG_XVC=1

28.2 USB Host Driver Summary

Table 52-4 provides a summary of source code location, library dependencies and other BSP information for the HS host driver.

28.2.1 FS Host2 Driver Summary

Table 28-4. HS Host2 Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX35-3DS-PDK1_7
Target SOC (TGTSOC)	MX35_FSL_V2_PDK1_7
CSP Driver Path	..\SOC\<*Common SOC*>\ms\USBH\EHCI ..\SOC\<*Common SOC*>\ms\USBH\EHCIPDD ..\SOC\<*Common SOC*>\ms\USBH\USB2COM
CSP Static Library	usbh_ehcdmdd_<*Common SOC*>.lib usbh_ehcdpdd_<*Common SOC*>.lib usbh_usb2com_<*Common SOC*>.lib
Platform Driver Path	\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBH\HSH2
Import Library	N/A
Driver DLL	hcd_hsh2.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed Host To support only host mode, choose .. >High Speed Host Function.
SYSGEN Dependency	SYSGEN_USB=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSH2=1

Host driver requires a set of class drivers to be loaded. See Section 28.5.6, “Class Drivers for class driver information.

Notes: The H2 host on MX35 is a FULL SPEED Host controller.

28.3 Supported Functionality

The OTG driver provides the following software and hardware support:

1. High Speed OTG/Host driver supports USB specification 2.0.
2. 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 awaits attachment to a host by raising its D+ signal. On attachment of a mini-A plug the driver switches to host mode.
3. When a mini-B plug is connected to the OTG port, and the cable opposite end is connected by a mini-A (or A-type) plug to a PC, then the OTG initiates operation as peripheral and responds to USB protocol from the host.

4. When a mini-A plug is connected to the OTG port and the cable opposite end is connected by a mini-B plug to another OTG device, then the OTG initializes/re-initializes itself into host mode and begin to act as a host. The OTG port remains in host mode whenever a mini-A plug is mated to the OTG socket connector.
5. OTG port as client/peripheral supports mass storage, RNDIS and serial clients
6. OTG port as host or HS Host supports mass storage, HID and RNDIS classes
7. When nothing is attached to the OTG port, the driver configures the controller and transceiver into a low power state
8. When the system is suspended with nothing attached to the OTG/Host port, the system does not create a wake condition upon attachment of the port to a host or attachment of a device with mini-A plug
9. When the system is suspended while the OTG/Host port is connected to a host or to a device with a mini-A plug, the system remains suspended when the OTG port connection is unplugged
10. When the system resumes after suspend, any attached devices are enumerated and their class drivers loaded appropriately
11. Data transfer rates on the client port exceeds 40 Mbits/sec in Mass Storage client

28.4 Hardware Operation

The USBOH module on i.MX35 contains all of the functionality required to support two independent USB ports compatible with the USB 2.0 specification. In addition to the normal USB functionality, the module also provides support for direct connections to on-board USB peripherals with Serial, UTMI or ULPI protocol, and supports multiple interface types for ULPI and Serial Transceivers. In addition to the USB cores, the module provides for Full-Speed Transceiver less Link (TLL) operation on the host port and the OTG core also supplies the UTMI interface for the internal UTMI PHY.

28.4.1 Conflicts with Other Peripherals and Catalog Items

28.4.1.1 Conflicts with SoC Peripherals

The High Speed OTG port conflicts with CAN

28.4.1.2 Conflicts with Board Peripherals

No conflicts.

28.5 Software Operation

28.5.1 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 28-1.

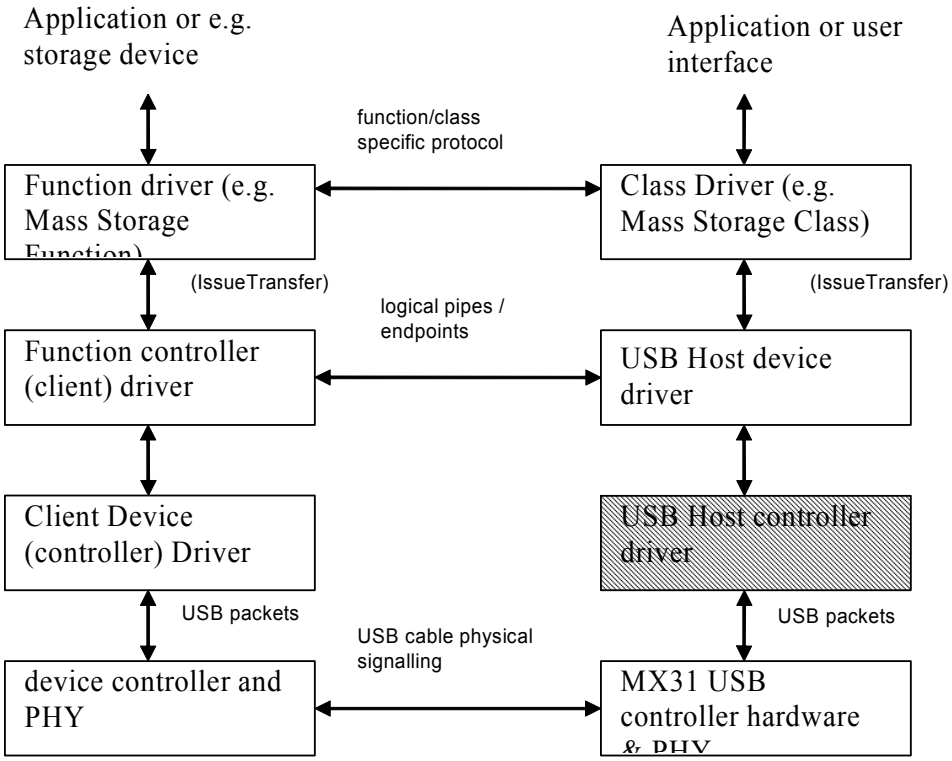

Figure 28-1. Windows USB Driver Architecture

For further details of the Windows CE USB driver architecture and usage, see the 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, the host controller driver enumerates and activates the appropriate class driver (see Section 28.5.1, “USB OTG Host Controller Driver,”).

The BSP supports the following USB class drivers:

- Mass Storage—SD cards, CF cards, HDD drive, thumb drive (disk-on-key); some card reader firmware is not supported by the Microsoft standard Mass Storage class driver
- HID—Keyboard and mouse
- RNDIS—Network Device Interface communication class

Hubs are supported in all configurations with Full and Low Speed peripherals.

28.5.1.1 User Interface

User access to the USB host driver is by class drivers. For further details on these Host Client Drivers refer to the 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.

28.5.1.2 Host Controller Configuration

The driver is configured into the BSP build by dragging and 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 28.5.1.5, "Registry Settings," for details on excluding OTG host support from the build.

The internal i.MX USB OTG signals can be multiplexed to a choice of pins on the IC as described in the IOMUX chapter of the *i.MX35 Applications Processor Reference Manual*.

28.5.1.3 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 Kbyte. This value can be altered by registry setting PhysicalPageSize.

28.5.1.4 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 via the platform-specific function BSPUsbhCheckConfigPower() as described in Section 28.5.1.8.1, "BSPCheckConfigPower," and located in:

```
\PLATFORM\<Target Platform>\SRC\DRIVERS\USBH\Common\hwinit.c
```

This function must be modified to correspond with the platform hardware capabilities.

The i.MX system can supply a total of 100 mA 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 500 mA and therefore are not supported for use. Self-powered hubs are required to expand the number of USB sockets and also to support devices that require greater than 100 mA.

28.5.1.5 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 by `platform.reg`. They do not normally require customization. Table 28-5 shows the default values contained in `hsotg.reg`.

Table 28-5. hsotg.reg Default Values

Value	Type	Content	Description
Dll	sz	hcd_hsotg.dll	Driver dynamic link library
OTGSupport	dword	01	This value must be set to 1 to enable host driver on the OTG. If no host support is required (client only) then this value can be set to 0, though the HCD_HSOTG key is not normally configured in the image at all when pure Host function is selected.
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.
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 128 Kbytes. From this buffer, 75% are allocated for transfer descriptors and the remaining buffer is available for allocation to simultaneous transfers. In most cases, only one transfer is active at any time (for example, in the Mass Storage Class). A good value is at least 3x as large as the largest data buffer transferred using IssueTransfer(). This key is optional, if it does not exists in the registry, it takes the default value, otherwise a specific value can be assigned.

28.5.1.6 Host USB Test Modes

The USB 2.0 specification defines PHY-level test modes for the USB host ports (see definitions in USB 2.0 specification section 7.1.20). The i.MX 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\<Target Platform>\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.

28.5.1.7 Unit Test

The USB driver has many devices to be tested. Tests are performed manually and include connecting the devices, and confirming the attach, detach (on unplug) re-attach (on subsequent plug in of device), and transferring and verifying data (and/or functions).

To verify the RNDIS class device, a CEPC containing Netchip 2280 USB function is attached and used to access a remote file server on the CEPC. To verify the low-level transport for Bulk, Interrupt and Isochronous transfers, the CETK Host test kit is performed. This requires a CEPC configured with Netchip 2280 USB function and loopback driver.

28.5.1.7.1 USB Host Controller Driver Test

Documentation for the Windows CE 6.0 CETK USB Host tests is normally found under the Platform Builder Windows CE product documentation:

Debugging and Testing > **Windows CE Test Kit** > **CE Test Kit**

28.5.1.7.2 Build the Test Image

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 <Target Platform> :ARMV4I-Device Drivers-USB Devices-USB High Speed Host1-High Speed Host 1
   - Core OS > Windows CE devices > Core OS Services > USB HOST Support; and all the sub-components of this catalog item (Sub-Components like USB Storage Class Driver.)
   - Core OS > Windows CE devices > File Systems And Data store > Storage Manager; (Sub-Components: FAT File System, Partition Driver, Storage Manager Control Panel Applet)
   - Device Drivers > USB Function > USB Function Clients-Serial.
3. Sysgen and build the image

28.5.1.7.3 Abstract

This test suite can be used to test USB host controller drivers that provide the same interface as Windows CE USB host controller driver does (for more information, see Section 28.5.1.1, “User Interface,”). It also can be used to verify whether a certain USB host controller (either stand alone card or onboard logic) can operate with Windows CE. The test setup and scenario is shown in Figure 28-2.

This test suite acts as a client driver above the USB bus driver (`usbd.dll`). It is loaded when a test device is connected to the host through a USB cable. The test device is a CEPC with a NetChip2280 USB function controller card in it. After this CEPC is booted up and `net2280lpbk.dll` is loaded, the CEPC acts as a

generic USB data loopback device. The USB test suite (the test client driver on the host side) can then stream data or issue device requests to or from this data loopback device. This is how the USB host controller and its corresponding host controller drivers are exercised.

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.

The Netchip2280 controller has six endpoints besides endpoint 0. The data loopback driver (`net2280lpback.dll`) configures these endpoints to be three pairs: one bulk IN/OUT pair, one Interrupt IN/OUT pair, and one Isochronous IN/OUT pair. The data loopback tests are done by sending data from host side to device side through the OUT pipe, receiving it back through the IN pipe, and then verify the data.

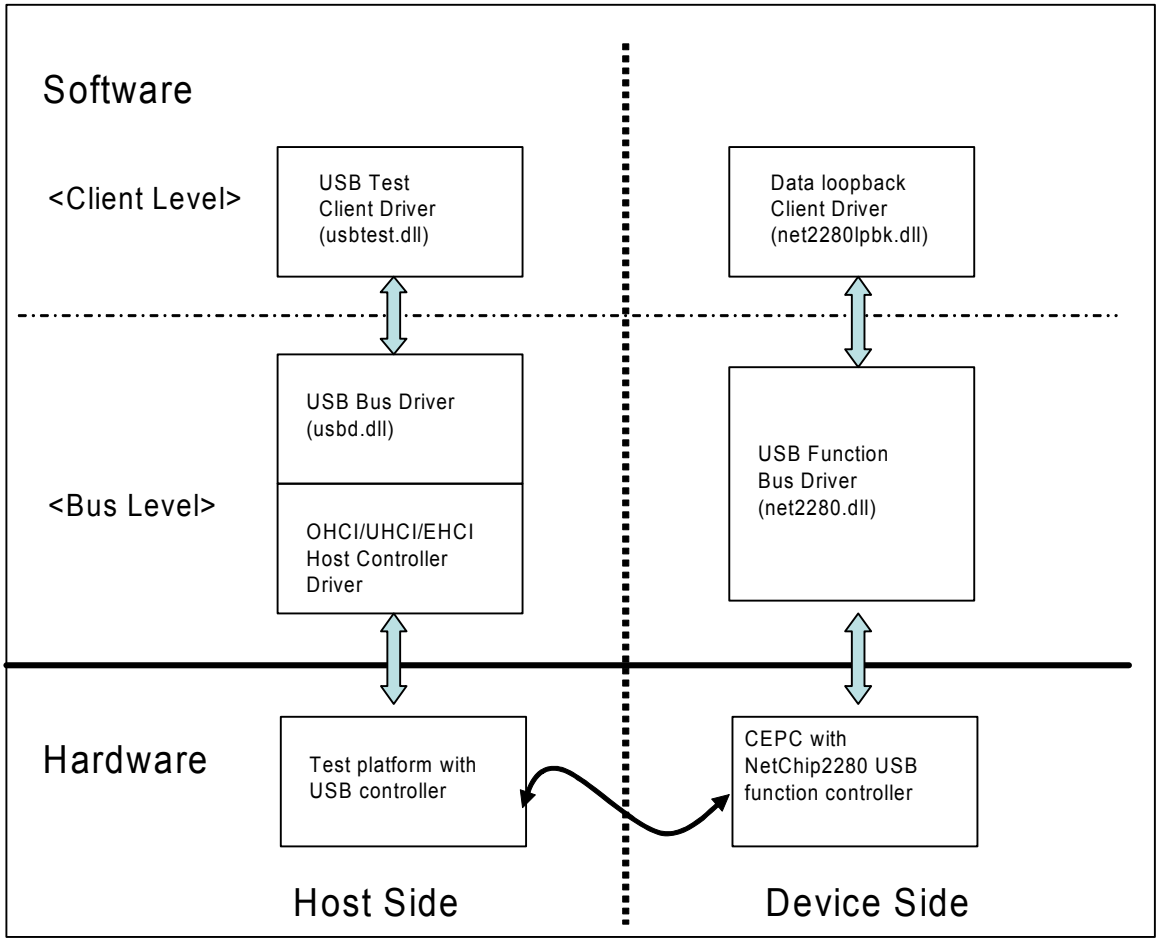

Figure 28-2. Test Setup

28.5.1.7.4 Unit Test Hardware

- Test platform
- Host Controller Card (if not onboard logic)
- CEPC
- Netchip2280 Card

- USB cable

28.5.1.7.5 Unit Test Software

Host side requirements:

- Tux.exe
- Ddlx.dll
- Usbtest.dll
- Tooltalk.dll
- Kato.dll
- USB component (usbd.dll, EHCI/OHCI/UHCI host controller driver(s)) must be included in the run time image

Device side requirements:

- Lufldrv.exe
- Net2280lpbk.dll
- NetChip2280 USB function support (net2280.dll) must be included in the CEPC run time image

28.5.1.7.6 Running the Test

The test procedure is as follows:

1. Download the runtime image to the CEPC (Windows Embedded CE PC-based hardware platform) with the Netchip2280 card on it
2. After the system is booted up, run `s lufldrv`, the tester should verify that `net2280lpbk.dll` is loaded
3. Download the runtime image to the test platform with a USB host controller on it
4. After the system is booted up, make sure there is no connection between the host side and the device through the USB cable. Then launch command `s tux -o -d ddlx -c "usbtest" "-xYYYY"`, where `YYYY` is the test case(s) to be run
5. The test indicates that there should be no connection between host and device side. Then after seven seconds, the test asks to connect two sides with a USB cable
6. The test main body starts to run
7. After test(s) is(are) done, and if other tests in the test suite are to be run, do not disconnect the two sides of the USB cable. Type the next test command, and the tests starts directly. If the USB connection was disconnected before the next test, the tests asks to make the connection again before the test begins

28.5.1.7.7 Test Cases

Table 28-6 shows the test cases contained in the test suite.

Table 28-6. USB Host Controller Driver Test Cases

Test Case ID	Test Description
1001-1315, 1501-1515	Data loopback tests: Concerning the transfer type, there are five categories: 1) Bulk pipe loopback tests (tests with ID end with 1, like xxx1) 2) Interrupt pipe loopback tests (tests with ID end with 2, xxx2) 3) Isochronous pipe loopback tests (tests with ID end with 3, xxx3) 4) All pipe transfer simultaneously (tests with ID end with 4, xxx4) 5) All three types transfers carry on simultaneously (tests with ID end with 5, xxx5) [1] There are five categories for how data is transferred: 1) Normal loopback tests (tests with ID start with 10, like 10) 2) loopback tests using physical memory (tests with ID start with 11, 11xx) 3) loopback tests using a part of allocated physical memory (tests with ID start with 12, 12xx) 4) Normal short transfer loopback tests (tests with ID start with 13, 13xx) 5) Stress short transfer loopback tests (tests with ID start with 15, 15xx) Also both synchronous and asynchronous transfer methods are exercised (test cases like xx1x using asynchronous transfer method, test cases like xx0x using synchronous method There are a total of 5×5×2 = 50 test cases
1401-1413	Additional data loopback tests. that mainly focus on testing APIs like GetTransferStatus(), AbortTransfer() and CloseTransfer()
2001-2013	Test related to Device requests
9001-9004	Special tests that test APIs such as SuspendDevice(), ResumeDevice() and DisableDevice()
9005	Test that stresses EP0 transfer (Vendor Transfer)

[1] This category of tests is designed for testing some other USB function devices which have more endpoints than host controller driver can handle. When using Netchip2280, it should be the same as category 4). Tester can just ignore this category.

By default, the data loopback device configures the endpoints with some often-used packet sizes that are DWORD aligned, and neither too big nor too small. By having all tests in Table 28-6 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 by themselves and run these test cases again for more comprehensive testing.

This test suite provides a way to change packet sizes of on NetChip2280 device on the fly. They are:

- Test case 3001—Using very small packet sizes in NetChip2280 device full speed configuration
- Test case 3002—Using very small packet sizes in NetChip2280 device high speed configuration
- Test case 3003—Using irregular packet sizes (like non DWORD-aligned size) in NetChip2280 device full speed configuration
- Test case 3004—Using irregular packet sizes (like non DWORD-aligned size) in NetChip2280 device high speed configuration

- Test case 3005 (High Speed only)—Using very large packet sizes (like 2×1024 for Isochronous endpoints) in NetChip2280 device full speed configuration. In the real world, Netchip2280 cannot handle transfers using such large packet size because its onboard FIFO buffer is small

Run one of the test case above, then after 15–20 seconds, `usbtest.dll` is unloaded and loaded again automatically through the Platform Builder. The packets sizes on netchip2280 side have already been changed. Then those normal tests can be run. Use test case 3011 (for full speed config) and 3012 (for high speed) to restore the default packet sizes.

Another category test that is important for USB2.0 host controllers and drivers is called the 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 lufldrv` to load loopback driver, run `s lufldrv -f`. This forces the Netchip2280 to be configured as a full speed device.

Also testers are encouraged to do some manual tests. Here are some examples:

- Plug in real USB devices, suspend system, and then resume; USB devices should still be there
- Plug in real USB devices, suspend system, unplug it, plug in another device, then resume; system should enumerate that new device properly
- Run one of the data transfer tests, in the middle of transfer stage, suspend the system (host side), then resume; tests may fail, but system should not crash
- Run one of the data transfer tests, in the middle of transfer stage, disconnect the USB connection; tests should fail, but system should not crash

28.5.1.8 Platform-Specific API

This section describes the platform-specific API functions.

28.5.1.8.1 BSPCheckConfigPower

This function is used to evaluate whether a device can be supported on the specified USB port.

Parameters

UCHAR bPort	[in] Unused. Each USB controller has only one port
DWORD dwCfgPower	[in] Power requirement (number of milliamps) requested by the device being evaluated for attachment support on this port
DWORD dwTotalPower	[in] current total power (number of milliamps) used by other previously attached devices on this port
Return Value	Return TRUE if device requesting dwCfgPower can be safely attached Return FALSE if device can not be attached

28.5.1.8.2 BSPUsbSetWakeUp

This function enables or disables the wakeup on the USB port. This function does not actually enable wake-up when a device is currently attached to the port.

Parameters

BOOL bEnable [in] TRUE to enable wakeup, FALSE to disable wakeup

28.5.1.8.3 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

28.5.1.8.4 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 a i.MX system uses a ULPI-bus based ISP1504 transceiver for the HS OTG port, and this function configures the ULPI-bus for sleep state. If platform hardware uses other transceivers, this function must be modified appropriately.

Parameters

BOOL fSuspend [in] TRUE: system/controller is going to suspend mode. FALSE: resuming

28.5.2 USB Client Driver

This driver enables the USB device functionality for the i.MX device. It is activated when a USB Mini B connector is connected to the Mini USB OTG socket. When the i.MX System is connected to a USB host system (for example, high speed or full speed port of PC), it is enumerated according to the current configuration settings, and the appropriate class driver is loaded on the PC. By default the system is configured for USB serial class. The system 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

28.5.2.1 User Interface

The USB client driver provides a standard Windows CE USB driver implementation. For an overview see:

Developing a Device Driver > Windows Embedded 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 the 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.

28.5.2.2 Client Driver Configuration

The OTG client driver is configured into the BSP build by dragging and dropping the appropriate catalog item (see Section 28.1.1, "USB OTG Client Driver Summary,"). 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).

28.5.2.3 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 28-7 shows the USB OTG client registry settings.

Table 28-7. 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

28.5.2.4 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 is not able to leave test mode. The device port does not by default support the USB test modes. Sample code for test mode support (SET_FEATURE on the device) is included in:

```
..\PLATFORM\COMMON\SRC\SOC\<Target SOX>\MS\USBFN\CONTROLLER\MDD
```

In addition, USBFN_TEST_MODE_SUPPORT must be defined during compilation of the CSP USBD device driver library.

28.5.2.5 Unit Test

There is no CETK test case for USB client (function) drivers. The USB function is tested by configuring the i.MX system as either USB serial function, 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.

28.5.2.5.1 Unit Test Hardware

Table 28-8 lists the required hardware to run the unit tests.

Table 28-8. Hardware Requirements

Requirement	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	Required as a storage device when the board is configured as mass storage class

28.5.2.5.2 Unit Test Software

Table 28-9 shows the software requirements for the USB Function controller driver test.

Table 28-9. Software Requirements

Requirement	Description
ActiveSync 4.1 and above	Host side software that is required to be available for testing the Serial class functionality

28.5.2.5.3 Running the USB Function Controller Driver Tests

Table 28-10 lists USB function controller driver tests.

Table 28-10. USB Function Controller Driver Tests

Test Cases	Entry Criteria/Procedure/Expected Results
Board configured as USB Serial class and connected to a host system after the board boots up completely	Entry Criteria: Make sure there is no mini USB OTG plug B is connected and the board is turned on and wait until the board boots-up completely Procedure: 1. Connect the mini USB OTG plug B to the mini USB OTG socket 2. Observe that the ActiveSync on the host side gets connected and is synchronized 3. Copy files from Host system to the Mobile Device. Files are copied 4. Copy files from the Mobile Device to the Host system. Files gets copied 5. Unplug the mini USB OTG plug B from the i.MX mini USB OTG socket to unload the Serial class driver Expected Result: ActiveSync should get synchronized and copying of files should happen between the Host and the System
Board configured as USB Mass storage client, with ATA drive as DSK1 mounted, and connected to a host system after the board boots up completely	Entry Criteria: Make sure there is no mini USB OTG plug B is connected and the board is turned on and wait until the board boots-up completely Procedure: 1. Connect the mini USB OTG plug B to the mini USB OTG socket 2. Observe that a new disk in My Computer having as Removable Disk appearing in it 3. Copy files from Host system to the new disk drive. Files are copied 4. Copy files from the new disk drive to the Host system. Files gets copied 5. Unplug the mini USB OTG plug B from the mini USB OTG socket to unload the mass storage class driver Expected Result: Files copied into mass storage client device match those copied out (when compared on Windows XP PC using file compare utility). Note that files are not visible from within the System until the system has been reset. The file system should not be used inside the System when it is being accessed via USB as a mass storage client.
Board configured as USB RNDIS client and connected to a host system after the board boots up completely. Browsing the Internet	Entry Criteria: Make sure there is no mini USB OTG plug B is connected and the board is turned on and wait until the board boots-up completely. See to it that the NIC's local area connection is not having any IP address Procedure: 1. Connect the mini USB OTG plug B to the mini USB OTG socket 2. Observe that a new Local area connection in the Network and Dial up connections appears on the Windows XP machine. Bridge the NIC's local area connection and the RNDIS's local area connection 3. Configure the bridge by giving IP address, Subnetmask, Default gateway, DNS 4. On the System, a new Local area connection can be found in the Network and dial up connections. Configure the local area connection by giving IP address, Subnetmask, Default gateway, DNS 5. In the Internet explorer on the System, configure the Lan settings as per the local area settings Expected Result: Browsing the Internet should be possible

28.5.2.6 Platform-Specific API

This section describes the platform-specific API functions.

28.5.2.6.1 InitializeMux

This function is called to initialize the IOMUX connection within i.MX, from the 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

28.5.2.6.2 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

28.5.3 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 the system. Upon detection the driver activates the USB host controller driver or USB function controller driver.

28.5.3.1 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.).

28.5.3.2 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.

28.5.3.3 Registry Settings

The USB OTG transceiver settings are values located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\XVC]
```

The values under this registry key are automatically included in the image via platform.reg. They do not normally require customization. Table 28-11 shows the USB OTG transceiver registry settings.

Table 28-11. 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 are not normally 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

28.5.3.4 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 the system. The test verifies that the USB host or function controller driver is activated on cable plug-in.

28.5.3.4.1 Unit Test Hardware

Table 28-12 lists the required hardware to run the unit tests.

Table 28-12. Hardware Requirements

Requirement	Description
System to act as a device	System 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 end. 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

28.5.3.4.2 Running the Transceiver Test

Table 28-13 lists transceiver tests.

Table 28-13. Transceiver Tests

Test Cases	Entry Criteria/Procedure/Expected Results
Idle case when no cable plugged in	Entry Criteria: Make sure there is no mini USB OTG plug connected and the board is turned on and wait until the board boots-up completely Procedure: When the board is powered and completely booted-up, the board should be idle (and as mass storage client, not verifiable) Expected Result: Device boots up and is stable
Mass storage client visible from PC	Entry Criteria: Make sure there is no mini USB OTG plug connected and the board is turned on and wait until the board boots-up completely Procedure: When the board is powered and completely booted-up, verify that board responds as a mass storage client when plugged into PC. Expected Result: New storage must be visible on PC.
Mini USB OTG plug-A connected to the mini USB OTG socket of System and mouse plugged into OTG port via this cable	Entry Criteria: Unplug board from PC (in previous step) Procedure: 1. Connect the USB HID device (Mouse) which has a Mini USB OTG plug-A to it. The control goes to the USB Host controller driver 2. The corresponding device gets enumerated and starts functioning. For example, if a USB mouse is connected, on movement of the mouse, the pointer in the LCD screen is seen moving Expected Result: Device should start functioning

28.5.3.5 Platform-Specific API

The transceiver driver library code contains i.MX chip-specific implementation, and is located in:

```
..\PLATFORM\COMMON\SRC\SOC\<Target SOC>\USBXVR
```

The transceiver driver operation can be customized through the platform-specific code located in:

```
..\PLATFORM\<Target Platform>\SRC\Drivers\USBXVR
```

The standard implementation located in `hwinit.c` is provided for the System with ISP1504 transceiver attached to the High Speed OTG port. Customizations 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 by 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 System Platform.

28.5.3.5.1 Structure BSP_USB_CALLBACK_FNS

Structure BSP_USB_CALLBACK_FNS is defined in `MX35_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() (see Section 28.5.3.6.2, "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);
        void (*pfnUSBPowerUp)(CSP_USB_REGS *regs, BOOL *pUSBCoreClk);
        // 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;
```

28.5.3.5.2 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 (for example, 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.MX, 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.
Return Value	TRUE—USB Core Clock is running FALSE—USB Core Clock is stopped

28.5.3.6 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 (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 see pfnUSBPowerDown, Section 28.5.3.5.2, "pfnUSBPowerDown,"

28.5.3.6.1 pfnUSBSetPhyPowerMode

This function is called when the system is in USB transceiver mode with no USB activity. With standard implementation on the system, 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.MX), 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.MX, 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.
Return Value	TRUE—callback function must resume transceiver activity FALSE—callback function must suspend transceiver activity

28.5.3.6.2 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 System. When no callback function is required, those elements of the BSP_USB_CALLBACK_FNS structure should be initialized to NULL.

Parameters

BSP_USB_CALLBACK_FNS *pFn

[in/out] Pointer to BSP_USB_CALLBACK_FNS structure for the developer to register the callback function inside the BSP_USB_CALLBACK_FNS. The callback functions inside this structure is used by the CSP transceiver code.

28.5.4 Power Management

There are four aspects of power management for the USB device drivers:

- Special i.MX Vcore requirements
- Clock gating to the USB peripheral block within the i.MX
- 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).

28.5.4.1 Special 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 the i.MX System), normal DVFS scaling of the i.MX Vcore must be suspended whenever there is potential of ULPI bus communication. This is the case whenever a device is connected (in host mode) or the device is connected to a host (in client mode). The USB OTG transceiver driver, and USB host and client drivers constrain the DVFS behavior by calling DDKClockEnablePanicMode() whenever a device or host connection is detected, and calling DDKClockDisablePanicMode() when a timeout period expires with no device or host connected to the port. No configuration is required, just note the effect on the DVFS (DVFC driver) behavior.

28.5.4.2 Clock Gating

The USB driver(s) for the various USB ports automatically manages clock gating to the i.MX 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)
```

28.5.4.3 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.

28.5.4.4 Transceiver Power Mode

Software timeout is used to establish whether the transceivers and their related bus 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.

28.5.4.5 PowerUp

Each of the OTG client, host and transceiver drivers have PowerUp routine associated. (For the host driver, this is referenced by the MDD to a function PowerMgmtCallback()).

For the host, the routine does the following:

- Ungate the USB peripheral block clock
- Force the port to resume and clear PHCD bit in the portsc register
- Reset and configure USB host controller
- Disable the wake-up conditions
- Set the PHY to normal work mode using SetPHYPowerMgmt(FALSE) platform routine
- Enable the interrupts and start the USB controller

For the client, the routine does the following:

- Ungate the USB peripheral block clock
- 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).

28.5.4.6 PowerDown

As for the PowerUp routine, OTG client, host and transceiver drivers have PowerDown routine associated. (For the host driver, this is referenced via the MDD to a function PowerMgmtCallback()).

For the host, the routine does the following:

- Verify the wake-up conditions using the BSPUsbCheckWakeUp() platform routine
- Stop the host controller
- Suspend the relevant port
- Set the PHY to low power mode using SetPHYPowerMgmt(TRUE) platform routine
- 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)
- Gate the USB peripheral block clock

For the transceiver driver, the PowerDown routine calls the relevant platform-specific callback routine, pfnUSBPowerDown().

Under normal circumstances there is nothing to be done in this routine, since the transceiver remains in its suspended state while nothing is connected to the port. Should any attachment have been made, the transceiver wakes through its wake-up mechanism and launch the appropriate (client or host) driver.

28.5.4.7 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

28.5.5 Function Drivers

The function drivers can be configured into the image using 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), 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`

28.5.5.1 Mass Storage Function

Table 28-14. Mass Storage Function

Driver Attribute	Definition
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<*Common SOC*>\ms\USBFN\CLASS
CSP Static Library	N/A
Platform Driver Path	N/A
Import Library	USBMSFN_LIB_<*Common SOC*>.lib UFNCLIENTLIB.LIB
Driver DLL	usbmsfn.dll
Catalog Item	Device Drivers > USB Function > USB Function Clients > Mass Storage
SYSGEN Dependency	SYSGEN_USBFN_STORAGE

The Mass Storage function exposes a local data store as a USB peripheral storage device. The device used can be specified in registry. In platfrom.reg, the following template is provided:

```
PUBLIC\Common\OAK\Files\common.reg
"DeviceName"=-;
; "DeviceName"="ATA HARD DISK"
; "DeviceName"="SDMEMORY CARD"
; "DeviceName"="MMC CARD"
; "DeviceName"="USB HARD DISK"
; "DeviceName"="NAND FLASH"
```

Any item from this list can be specified to act as the mass storage medium. Uncomment the corresponding line and rebuild the BSP to make that item active. If none of the items are specified explicitly, a pre-coded priority is used to determine what active drive acts as mass storage medium. The priority is described as the following:

ATA HARD DISK > SDMEMORY CARD (MMC CARD) > USB HARD DISK > NAND FLASH

platform.reg can also over-ride other USBMSFN related default settings. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Mass_Storage_Class]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
   "idVendor"=dword:045E
   "Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
   "idProduct"=dword:FFFF
   "Product"="Generic Mass Storage (PROTOTYPE--Remember to change idVendor)"
   "bcdDevice"=dword:0
```

28.5.5.2 Serial Function

The primary use for the serial function is ActiveSync.

Table 28-15. 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 ufnclientlib.lib
Driver DLL	SerialUsbFn.dll
Catalog Item	Device Drivers > USB Function > USB Function Clients > Serial Client
SYSGEN Dependency	SYSGEN_USBFN_SERIAL

NOTE

ActiveSync has been tested using connection to a 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
```

28.5.5.3 RNDIS Function

The RNDIS function allows communication over USB to be supplied to ethernet NDIS interface of protocol stack.

Table 28-16. RNDIS Function

Driver Attribute	Definition
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	N/A
PUBLIC Driver Path	PUBLIC\COMMON\OAK\Drivers\USBFN\Class\RNDIS
Import Library	ndis.lib
Driver DLL	RNDISFN.DLL
Catalog Item	Device Drivers > USB Function > USB Function Clients > RNDIS Client
SYSGEN Dependency	SYSGEN_USBFN_ETHERNET

RNDIS function has been tested using Freescale RNDIS class driver as located at:

```
Support\RNDIS\ce6_rndis.inf
%WINDIR%\System32\drivers\usb8023x.sys
```

When SYSGEN_USBFN_ETHERNET is defined, the default registry entry is automatically included from:

```
PUBLIC\Common\OAK\FILES\common.reg
```

For commercial products, this registry entry must be copied into platform.reg and modified to over-ride the defaults. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\RNDIS]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
"idVendor"=dword:045E
"Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
"idProduct"=dword:0301
"Product"="Generic RNDIS (PROTOTYPE--Remember to change idVendor)"
"bcdDevice"=dword:0
```

28.5.6 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 28-17 shows the standard Microsoft-supplied drivers that are available by drag and drop from the catalog.

Table 28-17. Class Drivers

Class Driver	Configuration Flag	Catalog Item
HID	SYSGEN_USB_HID	Core OS > Windows CE devices > Core OS Services > USB Host Support > USB Human Input Device (HID) Class Driver
Printer	SYSGEN_USB_PRINTER	.. > USB Printer Class Driver[1]
Keyboard	SYSGEN_USB_HID_KEYBOARD	.. > USB HID Keyboard Only[1]
	SYSGEN_USB_HID_MOUSE	.. > USB HID Mouse Only[1]
RNDIS	SYSGEN_ETH_USB_HOST	Core OS > Windows CE devices > Core OS Services > USB Host Support > USB Remote NDIS Class Driver
Storage	SYSGEN_USB_STORAGE	Core OS > Windows CE devices > Core OS Services > USB Host Support > USB Storage Class Driver

[1] See additional configuration in Section 28.6.2, "Dependencies of Drivers."

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`

28.5.6.1 HID Mouse

For mouse support, the cursor is required to test and use the mouse as shown in Table 28-18.

Table 28-18. HID Mouse Class Driver

Catalog Item	Configuration Flag	Catalog Item
HID	SYSGEN_CURSOR	Core OS > Shell and User Interface > User Interface > Mouse

28.5.6.2 HID Keyboard

The system keyboard key mapping conflicts with that used for the HID keyboard. When USB keyboard support is included, remove the System keyboard (Table 28-19) and include the appropriate stub keyboard and keyboard .dll (Table 28-20)

Table 28-19. HID Keyboard Driver to Remove

Remove Item	Remove Catalog Item
Keyboard	Third Party > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Input Devices > Keyboard/Mouse

Include stub keyboard:

Table 28-20. ID 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 the platform.bib (immediately before the FILES section):

```
IF BSP_KEYBD_NOP
    kbdmouse.dll    $(_FLATRELEASEDIR)\KbdnopUs.dll                    NK SH
ENDIF; BSP_KEYBD_NOP
```

28.6 Basic Elements for Driver Development

This section provides details of the basic elements for driver development in the Platform System.

28.6.1 BSP Environment Variables

Table 28-21 shows the system environment variables.

Table 28-21. System Environment Variables Summary

Name	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 pin muxing (platform-specific implementation) mean certain configurations are mutually exclusive, as listed in Table 28-22.

Table 28-22. Mutual Exclusive Driver Summary

Functionality[1]	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

Table 28-22. Mutual Exclusive Driver Summary (continued)

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	—	—	—	—

[1] yes = Required, no = Not permitted, — = Do not care

28.6.2 Dependencies of Drivers

Table 28-23 summarizes the Microsoft-defined environment variables used in the BSP.

Table 28-23. USB Driver

Name	Definition
SYSGEN_USBFN_SERIAL	Set to support serial class for USB Function controller
SYSGEN_USBFN_STORAGE	Set to support mass storage class for USB Function controller
SYSGEN_USBFN_ETHERNET	Set to support RNDIS class for USB Function controller
SYSGEN_CURSOR	Set to support mouse cursor
SYSGEN_FATFS	Set to support FAT16 file system
SYSGEN_PCL	Set to support PCL printing
SYSGEN_PRINTING	Set to support printer
SYSGEN_STOREMGR	Set to support storage manager
SYSGEN_UDFS	Set to support Universal Disc File System
SYSGEN_USB	Set to support USB driver
SYSGEN_USB_HID	Set to support Human Interface driver (HID) class
SYSGEN_USB_HID_CLIENTS	Set to support HID clients
SYSGEN_USB_HID_KEYBOARD	Set to support HID keyboards (keyboard stub and associated .dll are required)
SYSGEN_USB_HID_MOUSE	Set to support HID mouse
SYSGEN_USB_PRINTER	Set to support Printer (printer driver support, such as PCL (SYSGEN_PCL), may be required)
SYSGEN_USB_STORAGE	Set to support storage medium

Chapter 29
USB Boot and KITL

USB Boot and KITL are supported by implementing a RNDIS client device over USB on the target board. This feature configures the USB OTG port as a USB device and implements the RNDIS USB function driver. The USB RNDIS device acts as a normal ethernet device and connects to the PC over a USB cable. Eboot and KITL then operate with the RNDIS ethernet device.

29.1 USB Boot and KITL Summary

Table 29-1 identifies the source code location, library dependencies, and other BSP information.

Table 29-1. USB Boot and KITL Summary

Driver Attribute	Definition
Target Platform	iMX35-3DS-PDK1_7
Target SOC	MX35_FSL_V2_PDK1_7
SOC Common Path	WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\MS\RNE_MDD WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\MS\USBKITL
SOC Specific Path	WINCE600\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\USBD\KITL
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\COMMON\USBFN ..\PLATFORM\<*Target Platform*>\SRC\KITL
Driver DLL	fsl_usbfn_rndiskitl_PDK1_7.lib
SDK Library	N/A
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variable	N/A

29.2 Supported Functionality

The USB Boot and KITL provides the following software and hardware support:

1. Image downloading over USB RNDIS
2. KITL over USB
3. Provides menu options to determine whether or not to enable USB Boot and/or USB KITL

29.3 Hardware Operation

For detailed operation and programming information of the USB OTG, see the chapter on the High-Speed USBOTG_UTMI in the corresponding platform Users Guide.

29.3.1 Conflicts with Other Peripherals and Catalog Items

The USB Boot and KITL does not have conflicts with any other module. However, since USB KITL and USB OTG drivers share the same USB OTG hardware, the USB OTG drivers should be disabled in the catalog item when USB KITL is enabled. USB boot does not have such limitation.

29.4 Software Operation

29.4.1 Software Architecture

USB Boot and KITL are part of the eboot and KITL subsystem. A RNDIS client device is implemented to support USB Boot and KITL. Figure 29-1 illustrates the USB Boot and KITL software architecture.

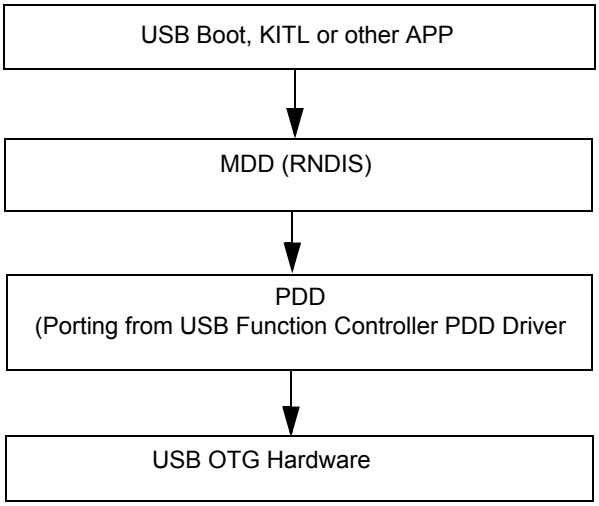

Figure 29-1. USB Boot and KITL Software Architecture Block Diagram

Microsoft has implemented a RNDIS client MDD driver in Windows CE 6.0. The code is in following location:

`%_WINCEROOT%\Public\Common\Oak\Drivers\Ethdbg\Rne_mdd`

It generates the static library `Rne_mdd.lib`.

The USB function controller PDD driver is ported to eboot and KITL to support USB Boot and KITL. For details of USB function controller PDD driver refer to the Platform Builder for Windows CE 6.0 Help under the topic:

Developing a Device Driver > Windows Embedded CE Drivers > USB Function Drivers > USB Function Controller Drivers > USB Function Controller Driver Reference > USB Function Controller PDD Functions.

Windows CE 6.0 provides an example of USB Boot. It is located at:

`%_WINCEROOT%\Platform\MainstoneIII\Src\Common\Usbfn`

29.4.2 Source Code Layout

Some files are modified or added to support USB Boot and KITL. They are as follows:

- RNDIS PDD driver
  `%_WINCEROOT%\Platform\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\MS\USBKITL\RNDIS`
- USB function controller shared with OS driver
  `%_WINCEROOT%\Platform\COMMON\SRC\SOC\<Target SOC>\USBD\COMMON`
- Add RNDIS device to EBOOT ethernet initialization routines
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Common\ether.c`
- Setup KITL device LogicalLoc and PhysicalLoc to USBOTG physical address if USB KITL option in EBOOT menu is selected by user
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Common\main.c`
- Implement private OS functions, such as `NKCreateStaticMapping(). NKCreateStaticMapping` is defined in OS. It is not defined for EBOOT while USB Boot requires this function. So it is manually defined. This function just calls OALPAtoUA()
  `%_WINCEROOT%\Platform\COMMON\SRC\SOC\<Target SOC>\USBD\KITL`
- Add USB Boot and KITL options into EBOOT menu
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Eboot\menu.c`
- Add `fsl_rne_mdd_$(_COMMONSOCDIR).lib, fsl_rne_pdd_$(_COMMONSOCDIR).lib, usb_usbfn_$(_SOCDIR).lib, usb_usbfn_eboot_$(_SOCDIR).lib`
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Eboot\sources`
- Add USB RNDIS KITL device in KITL initialization routines
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Kitl\kitl.c`
  `%_WINCEROOT%\Platform\<Target Platform>\Src\Kitl\sources`

29.4.3 Power Management

Power management is not implemented in USB Boot and KITL.

29.4.4 Registry Settings

There are no related register settings for the USB Boot and KITL.

29.4.5 DMA Support

Physical contiguous memory is required to support USB DMA. This memory region is hard coded in:
`%_WINCEROOT%\Platform\Common\SRC\SOC\<Common Soc>\ms\Usbkitl\Rndis\rndis_pdd.c`

It uses the BSP reserved IPL RAM image region (Starting from IMAGE_USB_KITL_RAM_PA_START). This region is not used by other modules in the BSP, so it can be used by USB boot and KITL.

29.5 Unit Test

29.5.1 Building the USB Boot and KITL

There is no special configuration options for building USB Boot and USB KITL. Building the BSP with default configuration includes the USB Boot and KITL support. The exception is that the USB OTG drivers should be deselected from the catalog item view before building the NK image to use USB KITL, because USB KITL and OS USB drivers share the same USB OTG hardware and they can not exist simultaneously. As a result USB KITL can not used to debug USB OTG drivers.

The USB OTG driver auto unloads when it detects USB KITL enabled.

29.5.2 Testing USB Boot and KITL on i.MX35 3-Stack

There are three Ethernet transport devices available on the 3-Stack for image download with Boot and KITL connection:

- LAN9217 (default)
- FEC
- USB RNDIS.

Follow the steps below to use USB RNDIS for Boot and KITL:

1. Connect target board to PC with USB cable and power on the board.
2. Select USB RNDIS as the Ethernet transport device in EBOOT.

```
0) IP address: 192.168.0.2
1) Subnet Mask: 255.255.255.0
3) DHCP: Disabled
6) Set MAC Address: 0-12-34-56-78-12
9) Ethernet Device: USB RNDIS
```

3. Press ‘d’ to download image over USB. If this is the first time running USB Boot or KITL with the PC, the PC shows a “Found New Hardware Wizard” dialog box and prompt you to install the driver for Microsoft Windows CE RNDIS virtual adapter on the Windows PC. Refer to

   `WINCE600\PUBLIC\COMMON\OAK\DRIVERS\ETHDBG\RNDISMINI\HOST\howto.txt`

   for how to install the driver.
4. After the driver is installed successfully, the Microsoft Windows CE RNDIS virtual adapter should be displayed in 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 Platform Builder Target > Connectivity options to make sure the target device is selected.
6. If KITL is being enabled in the run-time image, the connection is established on USB after the image downloading is finished.