i.MX53 ARD
Windows Embedded Compact 7

Reference Manual

Part Number: 924-76370
Rev. WCE700_MX53_ER_1105
06/2011

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
Backlight Driver

Chapter 5
Battery Driver

Chapter 6
Boot from Secure Digital/MultiMedia Card (SD/MMC)

Chapter 7
Camera Driver for IPUv3

Chapter 8
Controller Area Network (CAN) Driver

Chapter 9
Chip Support Package Driver Development Kit (CSPDDK)

Chapter 10
Display Driver for IPUv3

Chapter 11
Dynamic Voltage and Frequency Control (DVFC) Driver

Chapter 12
Enhanced Configurable Serial Peripheral Interface (eCSPI) Driver

Chapter 13
Enhanced Secure Digital Host Controller (eSDHC) Driver

Chapter 14
Enhanced Serial Audio Interface (ESAI) Driver

Chapter 15
Global Positioning System (GPS) Driver

Chapter 16
Graphics Processing Unit (GPU)

Chapter 17
Inter-Integrated Circuit (I^2C) Driver

Chapter 18
IIM(IC Identification Module) Driver

Chapter 19
NAND Flash Driver

Chapter 20
Power Management IC (PMIC)

Chapter 21
Serial Driver

Chapter 22

Sony/Philips Digital Interface (SPDIF) Driver

Chapter 23
Touch Panel Driver

Chapter 24
Temperature Sensor Driver

Chapter 25
Universal Serial Bus (USB) Driver

Chapter 26

USB Boot and KITL

Chapter 27
UUT(Universal Updater Tool) Driver

Chapter 28
Video Processing Unit (VPU)

Chapter 29

WLAN Driver

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® Embedded Compact 7.

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 wants to know more about Freescale's software development kit (SDK) for Microsoft Windows Embedded Compact 7.

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.MX53 ARD Release Notes for Windows Embedded Compact 7
- i.MX53 ARD User's Guide for Windows Embedded Compact 7
- Microsoft Platform Builder for Windows Embedded Compact 7 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 Compact 7 operating system. This BSP supports the following Freescale platform(s):

- i.MX53 ARD Development System

This kit supports the Microsoft Windows Embedded Compact 7 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 Compact 7 documentation, visit: http://msdn.microsoft.com/en-us/library/gg154201.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 Compact 7 Architecture

The Windows Embedded Compact 7 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.

Chapter 2
Asynchronous Sample Rate Converter (ASRC) Driver

The Asynchronous Sample Rate Converter (ASRC) converts the sampling rate of a 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.

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	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\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_v3.lib, asrc_common_fsl_v3.lib, asrcbase_<*Target SOC*>.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > ASRC
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOAUDIO= BSP_ASRC=1

2.2 Supported Functionality

The ASRC driver enables the ARD board to provide the following software and hardware support:

1. Supports standard stream interface for application usage.
2. Supports flexible 8/16/24 bit width of input data, and 16/24 bit width of output data.
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

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

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Asrc]
"Prefix"="ARC"
    "Dll"="asrc.dll"
    "Order"=dword:28
    "Index"=dword:1
    "PairIndex"=dword:0
    "MaxChnNum"=dword:2
    "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

The **PariIndex** registry key indicates which sampling rate pair of ASRC hardware module should be used by ASRC driver. In ASRC driver, pair A is fixed to be used.

The **MaxChnNum** registry key is used to restrict the maximum channel used by ASRC driver.

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 can be different from general wave file, the wave file used for ASRC test can be converted to different format. 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:

```
\WINCE700\SUPPORT\TEST\ASRC\
```

And there are three sub-directory in this directory:

```
\WINCE700\SUPPORT\TEST\ASRC\FILE_CONVERT
\WINCE700\SUPPORT\TEST\ASRC\ASRC_TEST
\WINCE700\SUPPORT\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 file containing one of the wave format (24-bit packed in 32-bit package, bit0–bit23 valid) 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 configures ASRC to work in 24-bit data fomat mode for both input and output, 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;
openParam.inputDataWidth = 32;
openParam.outputDataWidth = 32;
ASRCOpenPair(g_hASRC,&openParam );

// config the pair for conversion
configParam.clkMode = ASRC_CLK_NONE_SRC;
configParam.pairIndex = (ASRC_PAIR_INDEX)g_dwPairIndex;
configParam.inputBitClkRate = g_dwInputSampleRate*2*24;
configParam.outputBitClkRate= g_dwOutputSampleRate*2*24;
configParam.inputSampleRate = g_dwInputSampleRate;
configParam.outputSampleRate = g_dwOutputSampleRate;
configParam.inputValidBitsPerSample = ASRC_IDATA_WIDTH_16BIT;
configParam.outputValidBitsPerSample = ASRC_ODATA_WIDTH_16BIT;
configParam.inputDataAlign = ASRC_DATA_ALIGN_LSB;
configParam.outputDataAlign = ASRC_DATA_ALIGN_LSB;
configParam.outputExtension = ASRC_DATA_EXTENSION_NOSIGN;
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->ESAI 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 choose 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->ESAI 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;
configParam.inClkSrc = ASRC_SRC_ASRCK1;
configParam.outClkSrc = ASRC_SRC_ESAI_TX;
...
configParam.outputSampleRate = g_dwOutputSampleRate;
ASRCConfig(g_hASRC,&configParam );
...
```

In this mode, clk Mode is set as ASRC_CLK_ONE_SRC_OUTPUT, 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 Windows Embedded Compact 7 BSP is a block driver, used as the underlying layer for File Systems and USB mass storage. It is constructed as a stream interface driver that exposes I/O control interfaces (IOCTL_DISK_XXX, DISK_IOCTL_XXX, IOCTL_CDROM_XXX, IOCTL_DVD_XXX). The file system uses these I/O control interfaces 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	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\ATA
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ATA
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\BLOCK\ATA
Driver DLL	sata.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX53 ARD: ARMV7 > Storage Drivers > SATA Drivers > SATA Hard Disk Drive Support (ATA driver) Third Party > BSP > Freescale i.MX53 ARD: ARMV7 > Storage Drivers > SATA Drivers > SATA CD/DVD ROM Support (ATAPI driver)
SYSGEN Dependency	SYSGEN_STOREMGR_CPL,SYSGEN_MSPART,SYSGEN_FATFS,SYSGEN_EXFAT,SYSGEN_UDFS
BSP Environment Variable	BSP_NOATA= BSP_SATA=1 BSP_ATA=1 (ATA driver) BSP_ATAPI=1 (ATAPI driver)

3.2 Supported Functionality

The ATA driver provides the following 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 full sustained (media) data throughput capacity of Hitachi Travelstar 5K500.B-320 (or equivalent) at SATA 1.5-Gbps Generation 1 mode.

The ATAPI driver provides the following 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 CD/UDFS file system.
4. Supports full sustained (media) data throughput capacity of Continental DVD-ROM (or equivalent) at SATA 1.5-Gbps Generation 1 mode.

3.3 Hardware Operation

The i.MX SOC contains an on-chip SATA controller which uses DesignWare Cores SATA AHCI Core.

The DesignWare Cores SATA AHCI Core (commonly referred to as the DWC_ahsata), implements the Serial Advanced Technology Attachment (SATA) storage interface for physical storage devices.

The DWC_ahsata is an AHCI-compliant SATA AHCI Host Bus Adaptor (HBA). Together with the corresponding multi-port physical layer (PHY), it forms a complete AHCI HBA interface.

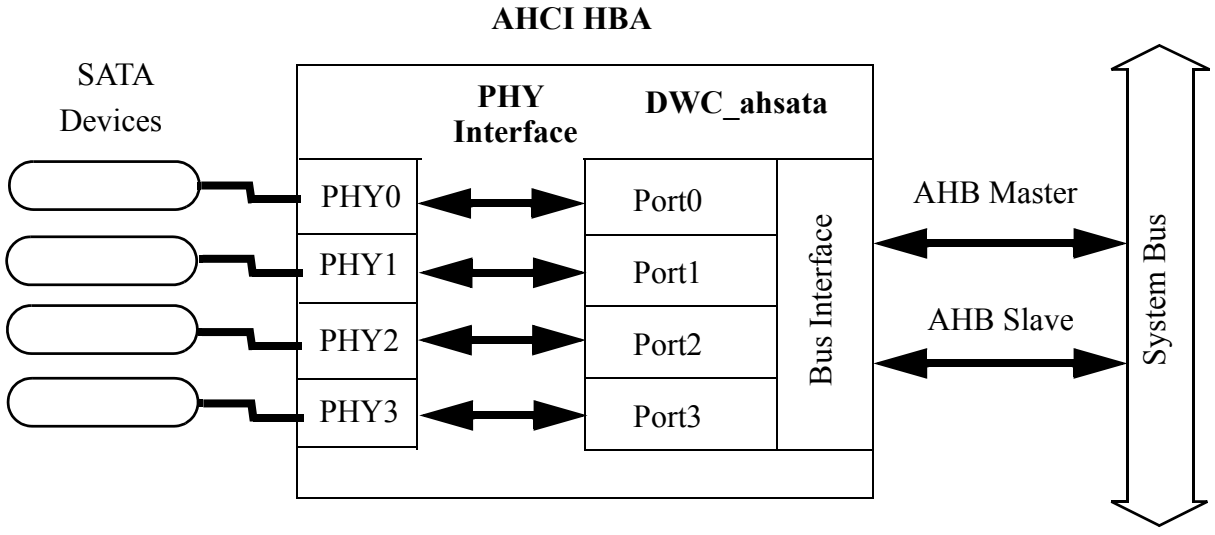


Figure 3-1. ATA Hardware Block Diagram

Although 4 ports are shown, only PORT0 and PHY0 are present.

The following are the supported features of the DWC_ahsata:

- Supports SATA 1.5-Gbps Generation 1.
- Compliant with Serial ATA Specification 2.6, and AHCI Revision 1.1 specifications.
- Supports one SATA devices (port 0).
- Internal DMA engine per port.

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

3.3.1 Conflicts with Other Peripherals

3.3.1.1 Conflicts with SoC Peripherals

No conflicts.

3.3.1.2 Conflicts with board Peripherals

No conflicts.

3.3.2 Cabling

The SATA standard defines a data cable with seven conductors (3 grounds and 4 active data lines in two pairs).

The SATA standard specifies a different power connector than the decades-old four-pin Molex connector found on pre-SATA devices. Like the data cable, it is wafer-based, but its wider 15-pin.

3.4 Software Operation

3.4.1 Application/User Interface to ATA/ATAPI drives

The ATA/ATAPI device exports a standard stream 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 IOCTLs provide interfaces to acquire disk information and to read/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, BSP_SATA, BSP_ATA for ATA driver, BSP_NOATA, BSP_SATA, BSP_ATAPI 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 working mode, and the device prefix and index, if necessary.

3.4.2.1 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 block devices can influence indexes automatically assigned by the bus enumerator.

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 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

3.4.4.1 ATA Driver

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\SATA]
```

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

Table 3-2. ATA Driver Registry Setting Values

Value	Type	Content	Description
Dll	sz	sata.dll	Driver dynamic link library
IClass	sz	"{A4E7EDDA-E575-4252-9D6B-4195D48BB865}" GUID for a power-manageable block device	
InterruptDriven	dword	01 (00)	enable interrupt driven I/O (disable interrupt; not used normally)

Standard registry entries also to be included for the ATA device after 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, automatically assigned when driver loads)
Order	dword	10	Early, to allow file system loading
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 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
```

3.4.4.2 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\SATA]
```

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

Table 3-4. ATAPI Driver Registry Setting Values

Value	Type	Content	Description
Dll	sz	sata.dll	Driver dynamic link library
IClass	sz	"{A4E7EDDA-E575-4252-9D6B-4195D48BB865}" GUID for a power-manageable block device	
InterruptDriven	dword	01 (00)	enable interrupt driven I/O (disable interrupt; not used normally)

Standard registry entries also to be included for the ATA device after the above key are shown in Table 3-5.

Table 3-5. ATAPI 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, automatically assigned when driver loads)
Order	dword	10	Early, to allow file system loading
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 ATA makes use of the CDProfile information from the StorageManager registry keys. Default values are as below:

```
"Name"="IDE CDROM/DVD Drive"
    "Folder"=LOC_STORE_CD_FOLDER
    "DefaultFileSystem"="MSIFS_CD"
    "PartitionDriver"="CDRom.DLL"

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

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\CDProfile\PartitionTable]
```

3.4.5 DMA Support

All data transfers between the device and system memory occur through the HBA, which acts as a bus master to system memory. Whether the transaction is of a DMA type or a PIO type, the HBA fetches and stores data to memory, offloading the CPU.

All transfers are performed using DMA. The use of the PIO command type is strongly discouraged. PIO has limited support for errors, for example, the ending status field of a PIO transfer is given to the HBA during the PIO Setup FIS, before the data is transferred.

For ATA driver, it always uses the scatter gather method.

If the buffer from the upper layer meets the following criteria:

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

The ATA driver builds the Scatter Gather DMA Buffer Descriptors, but does not allocate or manage DMA buffers internally. All buffers are allocated and managed by the upper layers, the details of the buffer 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.

For ATAPI driver, it allocates and manages DMA buffers internally.

3.5 Unit Test

The ATA driver is tested using the Storage Media - Hard Drive Tests included as part of the Compact Test Kit (CTK). There are no custom CTK test cases for ATA driver. The Storage Media - Hard Drive Tests used to test ATA driver include:

- File System Driver Tests for Hard Drive
- File System Performance Tests for Hard Drive
- Large File Support Tests for Hard Drive
- Partition Driver Test for Hard Drive
- Storage Block Device Driver API Test for Hard Drive
- Storage Device Block Driver Performance Test for Hard Drive
- Storage Device Block Driver ReadWrite Test for Hard Drive

The ATAPI driver is tested using the Storage Media - CD/DVD Tests included as part of the Compact Test Kit (CTK). There are no custom CTK test cases for ATAPI driver. The Storage Media - CD/DVD Tests used to test ATAPI driver include:

- Audio CD Driver Test
- CD.DVD-ROM Block Driver Test
- CD.DVD-ROM File System Driver Test

3.5.1 Unit Test Hardware

Table 3-6 lists the required hardware to run the ATA driver unit tests.

Table 3-6. ATA Driver Hardware Requirements

Requirement	Description
i.MX SOC and attached Hitachi Travelstar 5K500.B-320.	Other drives supporting SATA 1.5-Gbps Generation 1 mode may be used.

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

Table 3-7. ATAPI Driver Hardware Requirements

Requirement	Description
i.MX SOC and attached Continental DVD-ROM.	Other drives supporting SATA 1.5-Gbps Generation 1 mode may be used.

3.5.2 Unit Test Software

Table 3-8 lists the required software to run the Storage Device Tests.

Table 3-8. 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 3-9. ATA Driver Software Requirements

Requirement	Description
Fsdtst.dll	Test .dll file used to perform File System Driver Test for Hard Drive.
Fsperflog.dll, ceperf.dll, perfscenario.dll	Test .dll file used to perform File System Performance Tests for Hard Drive.
fsLargeFiles.dll	Test .dll file used to perform Large File Support Tests for Hard Drive.
Msparttest.dll	Test .dll file used to perform Partition Driver Test for Hard Drive.
Disktest.dll	Test .dll file used to perform Storage Block Device Driver API Test for Hard Drive.
PerfLog.dll, Disktest_perf.dll	Test .dll file used to perform Storage Device Block Driver Performance Test for Hard Drive.
Rwtest.dll	Test .dll file used to perform Storage Device Block Driver ReadWrite Test for Hard Drive.

Table 3-10. ATAPI Driver Software Requirements

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

3.5.3 Building the Storage Device Tests

The Storage Device Tests come pre-built as part of the CTK. No steps are required to build these tests. All the test .dll files can be found alongside the other required CTK files in the following location:

```
\Program Files\WindowsEmbeddedCompact7TestKit\tests\target\armv7
```

3.5.4 Running the Storage Media Tests

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

3.5.4.1 ATA Driver

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

The command line for running the File System Driver Test is:

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

it performs file system tests which cover all required File System API functions.

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 File System Performance Tests is:

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

The command line for running the Large File Support Tests is:

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

The command line for running the Partition Driver Test is:

```
tux -o -d msparttest -c "-store DSK1: -zorch"
```

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, CTK cases 4021 can be safely skipped.

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 Driver ReadWrite Test is:

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

NOTE

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

For detailed information on the Storage Media - Hard Drive Tests, refer to:

- **Windows Embedded Compact 7 > Compact Test Kit (CTK) > Storage Media - Hard Drive Tests > File System Driver Test for Hard Drive**
- **Windows Embedded Compact 7 > Compact Test Kit (CTK) > Storage Media - Hard Drive Tests > File System Performance Tests for Hard Drive**
- **Windows Embedded Compact 7 > Compact Test Kit (CTK) > Storage Media - Hard Drive Tests > Large File Support Tests for Hard Drive**
- **Windows Embedded Compact 7 > Compact Test Kit (CTK) > Storage Media - Hard Drive Tests > Partition Driver Test for Hard Drive**
- **Windows Embedded Compact 7 > Compact Test Kit (CTK) > Storage Media - Hard Drive Tests > Storage Block Device Driver API Test for Hard Drive**
- **Windows Embedded Compact 7 > Compact Test Kit (CTK) > Storage Media - Hard Drive Tests > Storage Device Block Driver Performance Test for Hard Drive**
- **Windows Embedded Compact 7 > Compact Test Kit (CTK) > Storage Media - Hard Drive Tests > Storage Device Block Driver ReadWrite Test for Hard Drive**

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 Storage Media - CD/DVD Tests, refer to:

- **Windows Embedded Compact 7 > Compact Test Kit (CTK) > Storage Media - CD/DVD Tests > Audio CD Driver Test**
- **Windows Embedded Compact 7 > Compact Test Kit (CTK) > Storage Media - CD/DVD Tests > CD.DVD-ROM Block Driver Test**
- **Windows Embedded Compact 7 > Compact Test Kit (CTK) > Storage Media - CD/DVD Tests > CD.DVD-ROM File System Driver Test**

3.6 Basic Elements for Driver Development

This chapter provides details of the basic elements for ATA/ATAPI driver development.

3.6.1 BSP Environment Variables

Table 3-11. BSP Environment Variables

Name	Definition
BSP_NOATA	Set to disable ATA/ATAPI driver
BSP_SATA	Set to enable ATA/ATAPI driver
BSP_ATA	Set to enable ATA driver
BSP_ATAPI	Set to enable ATAPI driver

3.6.2 Mutual Exclusive Drivers

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

3.6.3 Dependencies of Drivers

Table 3-12 summarizes the Microsoft defined environment variables used in the BSP.

Table 3-12. Microsoft Defined Environment Variables

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

Table 3-13. ATA Driver Environment Variables

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

Table 3-14. 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 performed through the Windows File System.

For reverse compatibility deprecated DISK_IOCTLs are also supported but not documented here. See Windows Embedded Compact 7 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
Backlight Driver

The Windows Embedded Compact 7 BSP backlight driver follows Microsoft recommended backlight driver PDD/MDD architecture. The backlight driver uses PWM module on the SoC, to control the backlight on the Liquid Crystal Display (LCD) panel. The backlight driver is power-manageable, and it meets the requirements of a power-manageable device by implementing the required power management I/O Controls (IOCTLs).

4.1 Backlight Driver Summary

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

Table 4-1. Backlight Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\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 <*Target Platform*>: ARMV7> Device Drivers > Smart Backlight Control > Backlight control using the PWM
SYSGEN Dependency	SYSGEN_BATTERY=1
BSP Environment Variables	BSP_BACKLIGHT_PWM=1

4.2 Supported Functionality

The backlight driver provides the following support:

1. Conforms to the Device Manager streams interface
2. Supports 0–36 level adjustment
3. Supports power management mode: full on or full off

4.3 Hardware Operation

The backlight driver operate PWM module to set backlight level. The backlight level can be configured by writing the PWM LPCCR register.For more operation and programming information, see the chapter on the PWM in the i.*MX53Applications Processor Reference Manual*.

4.4 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

4.4.1 Backlight Driver Registry Settings

This section explains about the backlight driver registry settings.

i.MX53 ARD The following registry keys are required to properly load backlight driver:

```
[HKEY_CURRENT_USER\ControlPanel\Backlight]
    "BattBacklightLevel"=dword:FF  ; Backlight level settings. 0xFF = Full On
    "ACBacklightLevel"=dword:FF    ; 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

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Backlight]
    "Dll" = "backLight.Dll"
    "Prefix" = "BKL"
    "Flags"=dword:8 ;DEVFLAGS_NAKEDENTRIES
    "Index" = dword:1
    "Order" = dword:11            ; After display driver
    "IClass" = multi_sz:"{F922DDE3-6A6D-4775-B23C-6842DB4BF094}",
                        "{0007AE3D-413C-4e7e-8786-E2A696E73A6E}"
   "MinBrightness"=dword:0 ;Off
   "MaxBrightness"=dword:FF ;On
   "SupportedDx"=dword:11 ; D0 | D4 (bitwise 00010001)

    "RegKey"=dword:80000001 ;HKEY_CURRENT_USER
    "RegSubKey"="ControlPanel\\Backlight"
```

4.4.2 Power Management

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

4.4.2.1 PowerUp

This function is not implemented for the backlight driver.

4.4.2.2 PowerDown

This function is not implemented for the backlight driver.

4.4.2.3 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

4.5 Unit Test

The backlight driver is subject to one test suites provided with the Windows Embedded Compact Test Kit (CTK): the backlight API Test. The following section explains about the hardware and software requirements for unit tests.

The Backlight API test suite is a very simple functionality test that verifies all IOCTL calls for a single backlight. The test simply calls the IOCTL and verifies that it returns. It includes power manager IOCTL calls that matters to the backlight driver.

4.5.1 Unit Test Hardware

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

4.5.1.1 i.MX53 ARD Unit Test Hardware

Table 4-2 lists the required hardware to run the backlight application test.

Table 4-2. Hardware Requirements

Requirement	Description
Toshiba XGA LVDS panel	Display panel required for display of graphics data

4.5.2 Unit Test Software

4.5.2.1 Backlight API Test

Table 4-3 lists the software required to run the backlight API tests.

Table 4-3. Software Requirements

Requirement	Description
Tux.exe	Test harness, required for executing the test .
Kato.dll	Logging engine, required for logging the test data .
backlightapitest.dll	Library containing the test cases.

4.5.3 Building the Unit Tests

4.5.3.1 Backlight API Test

The Backlight API test comes pre-built as part of the CTK. No steps are required to build these tests. For information about the tests, see the Help:

Windows Embedded Compact 7 > Compact Test Kit (CTK)

4.5.4 Running the Unit Tests

4.5.4.1 Running the Backlight API Test

The command line for running the Backlight API test is:

```
tux.exe -o -d backlightapitest.dll -c "-p"
```

The -p command line flags is included to enable Power Management test. For information about the Backlight API test and command line options, see the Platform Builder Help:

Windows Embedded Compact 7 > Compact Test Kit (CTK) > Backlight Tests

4.6 Backlight API Reference

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

Windows Embedded Compact 7 > **Device Driver** > **Streams Interface Drivers**

Chapter 5
Battery Driver

The battery driver module provides information about the battery level to the OS. The battery driver is essentially a stub in this platform.The battery driver module is used to provide information about the battery level to the OS and control the charging and discharging function. When fake battery driver selected in catalog items view, the battery driver is essentially a stub and will not support charging function.

5.1 Battery Driver Summary

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

Table 5-1. Battery Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	N/A
SOC Common Path	N/A
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\BATTDRVR\Fake
Import Library	N/A
Driver DLL	battdrvr.dll
Catalog Item	Third Party -> BSP -> Freescale <*Target Platform*>: ARMV7 -> Device Drivers -> Battery -> Fake Battery Driver
SYSGEN Dependency	SYSGEN_BATTERY
BSP Environment Variables	BSP_NOBATTERY= BSP_FAKE_BATTERY=1

5.2 Supported Functionality

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

1. Conforms to the battery driver interface

5.3 Hardware Operation

The currenti.MX53 ARD does not support battery monitoring or charging.

5.3.1 Conflicts with Other SoC Peripherals

No conflicts.

5.4 Software Operation

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

5.4.1 Battery Driver Registry Settings

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

```
; These registry entries load the battery driver. The IClass value must match
; the BATTERY_DRIVER_CLASS definition in battery.h -- this is how the system
; knows which device is the battery driver.

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Battery]
   "Prefix"="BAT"
   "Dll"="battery.dll"
   "Flags"=dword:8                      ; DEVFLAGS_NAKEDENTRIES
   "Order" = dword:9
   "IClass"="{DD176277-CD34-4980-91EE-67DBEF3D8913}"
   "BattFullLiftTime" = dword:8         ;Batt Spec defined: in unit of hr, here 8hr is assumed
   "BattFullCapacity"=dword:320     ;Batt Spec defined: in unit of mAh, here 800mAhr is assumed
   "BattMaxVoltage"=dword:1068      ;Batt Spec defined: in unit of mV, here 4200mV is assumed
   "BattMinVoltage"=dword:BB8     ;Batt Spec defined: in unit of mV, here 3000mV is assumed
   "BattPeukertNumber"=dword:73     ;Batt Spec defined, here 1.15 is assumed
   "BattChargeEff"=dword:50     ;Batt Spec defined, here 0.80 is assumed
   "Ioctl"=dword:290418                 ; IOCTL to use for PostInit callback

; HIVE BOOT SECTION

[HKEY_LOCAL_MACHINE\System\Events]
    "SYSTEM/BatteryAPIsReady"="Battery Interface APIs"
; END HIVE BOOT SECTION
```

5.4.2 Power Management

There is no additional power management implementation for battery driver.

5.5 Unit Test

5.5.1 Unit Test Hardware

The battery driver does not include any unit tests.

5.6 Battery API Reference

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

Windows Embedded Compact 7 > **Device Drivers** > **Battery Drivers**

Chapter 6
Boot from Secure Digital/MultiMedia Card (SD/MMC)

- Boot support from SD/MMC includes the following components:EBOOT (may also be referred to as bootloader in this document)
- Storage for OS binary image (NK)

NOTE:

SD/MMC boot requires a card that is at least 96 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)	iMX53_ARD
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_V3\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 96 Mbytes in size
2. Storing bootloader and SD/MMC EBOOT 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. eSD2.1 and eMMC 4.3 boot from boot partition if boot partition can be configured to be at least 96 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.

6.4 Software Operation

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 and 8-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'sGuide*.

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 96 Mbytes for boot images (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. 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.

6.4.1.1 i.MX53 Card Memory Layout

Figure 6-1 shows the card memory layout for the i.MX53.

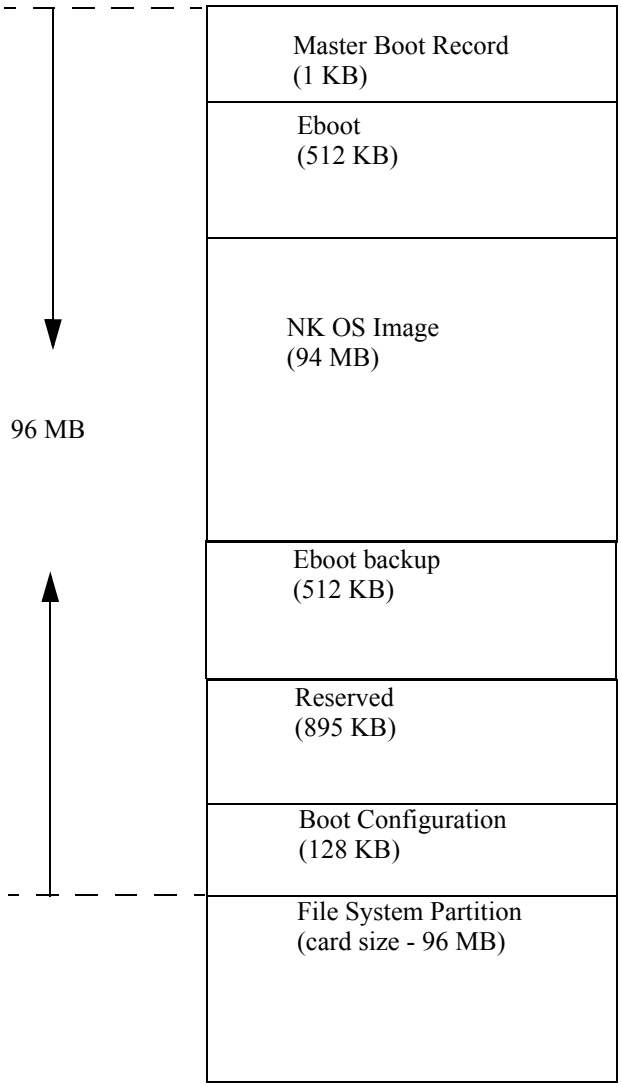


Figure 6-1. Card Memory Layout

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 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 for IPUv3

The camera driver is based on the Windows Embedded Compact 7 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 can support two camera instances, It 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 including:

- Communicating with and configuring the camera device through the I2C interface
- Configuring the submodules (CSI, SMFC and so on) of the Image Processing Unit v3 (IPUv3) for captured images
- Performing post-processing tasks with IPUv3 for the video preview data

The camera driver is compatible with the Tvin sensor ADV7180 .

The camera driver can support two camera instances. Camera1 use sensor ADV7180, Camera2 use CSI test mode. Of course, if use other sensor, the sensor special control code must be implemented and driver register item `"CameraId"` must be changed.

7.1 Camera Driver Summary

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	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\IPUV3\CAMERA
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\\<*Target Platform*>\SRC\DRIVERS\IPUV3\CAMERA
Driver DLL	camera.dll
SDK Library	N/A

Table 7-1. Camera Driver Summary (continued)

Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > Camera > CSI0 > CMOS OV3640 Support Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > Camera > CSI0 > CMOS OV5642 Support Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > Camera > CSI0 > TVin ADV7180 Support Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > Camera > CSI1 > CSI TESTMODE Support
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_I2CBUS3 = 1 BSP_PP = 1 For Camera driver 1: BSP_TVIN_ADV7180 = 1 For Camera driver 2: BSP_CSI_TESTMODE = 1

7.2 Supported Functionality

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

1. Windows Embedded Compact 7 Camera Device Driver Interface
2. Preview and Capture/Sill pins for camera1 application
3. Capture/Sill pins for camera2 application
4. ADV7180 TVin sensor for camera1 driver
5. Format from sensor output to CSI input (RGB565, YUV422)
6. Output resolution for Preview pin
   - 640×480 for VGA
   - 320×240 for QVGA
   - 160×120 for QQVGA
   - 352×288 for CIF
   - 174×144 for QCIF
7. Output resolution for Still pin
   - 720×576 for PAL
   - 720×480 for NTSC
8. Output resolution for Capture pin

   — 720×576 for PAL
   — 720×480 for NTSC
9. Output format for Preview pin (RGB565)
10. Output format for Still pin (UYVY, YV12, NV12)
11. Output format for Capture pin (UYVY, YV12, NV12)

7.3 Hardware Operation

Several hardware modules are involved in the operation of the camera driver. The input device (camera sensor) captures external image data. All other hardware elements of the camera driver are in the Image Processing Unit v3 (IPUv3). The IPUv3 Camera Sensor Interface (CSI) receives data from the sensor and converts the data into a format understood by the IPUv3. This data subsequently flows through the Sensor Multi FIFO Controller (SMFC) module for encoding or to the Image Converter (IC) for viewfinding where it undergoes post-processing. The encoding data or viewfinding data is then transferred by the IPUv3 DMA module to the final destination in the system memory .

For detailed operation and programming information, see the chapter on the Image Processing Unit (IPUv3) in the i.*MX53 Applications Processor Reference Manual*.

7.3.1 IPUv3 Overview

The low-level operation of the camera driver is based on the IPUv3. The IPUv3 is broken down into functional submodules. The following list describes the function each of these submodules:

- Camera Sensor Interface (CSI)—Gets data from the sensor and transfers data to one or more of the following: ISP, IC, SMFC
- Sensor Multi FIFO Controller (SMFC)—Controls FIFOs for the IDMAC channels related to the camera system
- Control Module (CM)—Provides control and synchronization for the entire IPUv3
- Image DMA Controller (IDMAC)—Transfers data to and from system memory
- Image Converter (IC)—Performs resizing, color conversion, combining with graphics, and horizontal inversion
- Image Rotator (IRT)—Performs rotation (90° or 180°) and inversion (vertical or horizontal)
- Post-processor Driver (PP)—General purpose image processing driver that performs the following processing tasks: color space conversion, resizing, rotation, and combining

The IPUv3 also contains the following regions of internal memory that store information used in the operation of the IPUv3:

- Task Parameter Memory (TPM)—Holds color space conversion coefficients and offsets
- Channel Parameter Memory (CPMEM)—Holds configuration information for each IDMAC channel

7.3.2 Conflicts with Other Peripherals and Catalog Items

7.3.2.1 Conflicts with SoC Peripherals

No conflicts.

7.3.2.2 i.MX53 Peripheral Conflicts

No conflicts.

7.4 Software Operation

The development concepts for camera driver is described in the Windows Embedded Compact 7 Help Documentation section under the topic:

Windows Embedded Compact 7> Device Drivers > Camera Drivers.

7.4.1 Software Architecture

7.4.1.1 Software Driver Components

Figure 7-1 shows the relationship between software components in the camera driver architecture.

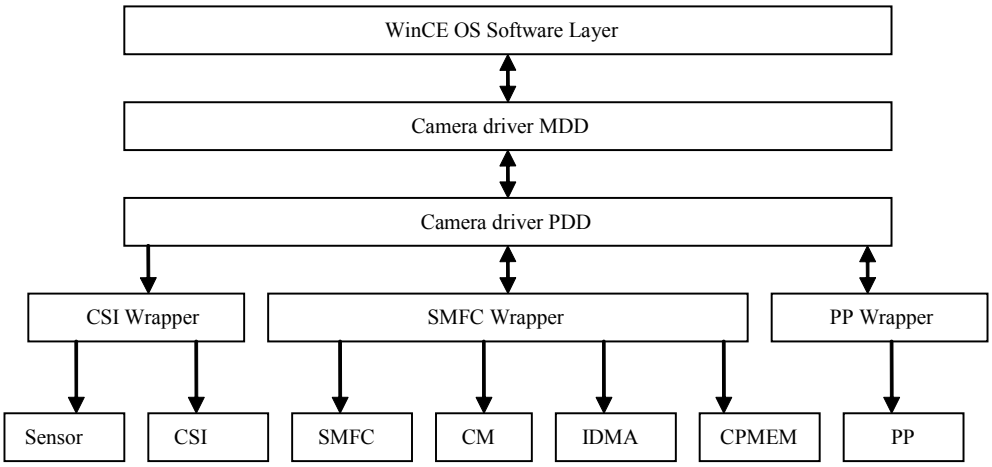


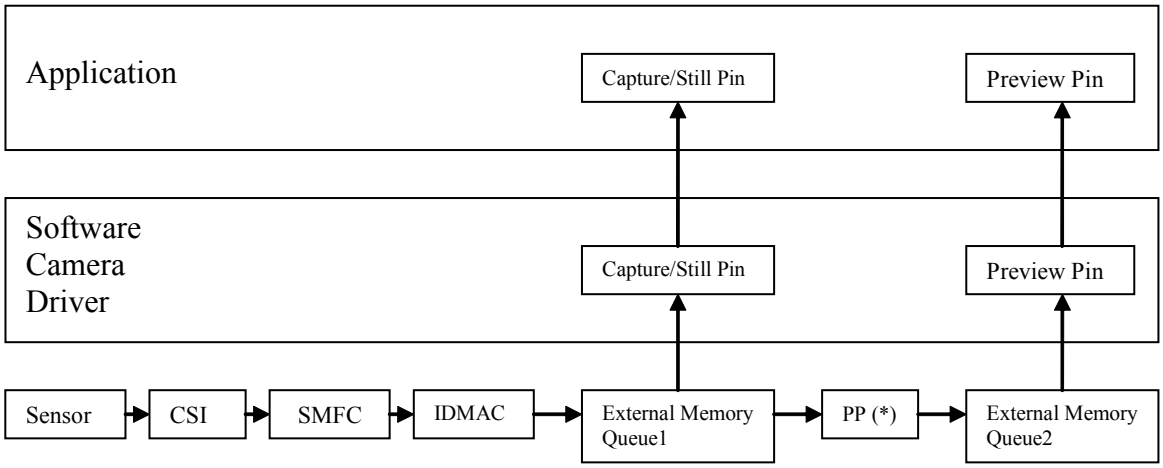
Figure 7-1. Camera Driver Architecture

Figure 7-1 shows the following main elements of the camera driver architecture:

- Camera driver MDD—Provides general interface to application
- Camera driver PDD—Implements the corresponding functions to encapsulate hardware specific code needed to write directly to the specific device
- CSI wrapper—Implements the sensor configuration and CSI module configuration
- SMFC wrapper—Implements the management of data comes from CSI
- PP wrapper—Implements the frame rotation/flip/mirror function

7.4.1.2 Data Flow

Figure 7-2 shows the data flow of the camera driver. The sensor passes the frame data to the CSI module, which then passes the data to the SMFC. The SMFC sets up the data for the IDMAC. The camera driver sets a pointer to an external memory buffer which is filled by the DMA after the IDMAC is complete. The camera driver uses the frame data in the external memory as the Capture/Still Pin output. Simultaneously, this frame data is used as the PP input for the Color Space Conversion (CSC), size change, and rotation/flip/mirror operation. The camera driver uses the PP output as the Preview Pin output. Since the frame data in the Capture/Still Pin does not pass the PP module, rotation, flip, or mirror operations cannot be achieve on the Capture/Still Pin.



Note (*): PP here is a concept, it includes many HW modules, such as IC IRT IDMAC and so on.

Figure 7-2. Camera Driver Data Flow

NOTE

The data for the Preview Pin depends on the data for Capture Pin. The hardware used by the Capture Pin must be configured and initiated before the Preview Pin to prepare the buffer. To enable these two pins, the Capture Pin must be configured before the Preview Pin to start earlier than the Preview Pin. If Preview Pin is already enabled, and then the Capture Pin should be enabled, the Preview Pin must be stopped first. Then the Capture Pin must be configured and started. Then the Preview Pin can be re-stared.

If an application uses client allocate buffer mode for the Capture Pin, then it should pay close attention to the process time required for one frame buffer. This is because data for the Preview Pin is based on data for the Capture Pin. If the application process time for one frame is too long to give the buffer back to driver, then the Capture Pin has no buffer to fill and the Preview Pin has no buffer input and output. This causes Preview frame loss.

There are two CSI interfaces, four SMFC channals. Camera1 use CSI0, SMFC(IDMAC channal0), PP; Camera2 use CSI1, SMFC(IDMAC channal2). So Camera1 can support Preview pin and Capture/Still pin, but Camera2 only support Captrue/Still pin.

7.4.1.3 Buffer Management

Buffers can be allocated either by camera driver or by the client as follows:

- Driver Allocate Buffers Mode—Buffers are allocated in hardware memory. The driver must have its own memory allocator and the client must retrieve the list of allocated buffers from the driver. A driver indicates its support for the buffer allocation model through the CSPROPERTY_BUFFER_DRIVER property. The client retrieves the list of buffers by calling DeviceIoControl with IOCTL_CS_BUFFERS and specifying CS_ALLOCATE.
- Client Allocate Buffers Mode—Buffers are allocated by the client and the client must initialize the buffers before it gives them to the driver. Once the client is done with the buffer, it must free the memory for the buffer. The driver indicates its support for the buffer allocation model through the CSPROPERTY_BUFFER_CLIENT_UNLIMITED property. The client negotiates the number of buffers by calling DeviceIoControl with IOCTL_CS_PROPERTY and specifying the property CSPROPERTY_BUFFER_COUNT. The client sends the buffers to the driver using IOCTL_CS_BUFFERS and specifying CS_ENQUEUE. The client releases the processed buffers by using IOCTL_CS_BUFFERS and specifying CS_DEALLOCATE.

7.4.1.3.1 Buffer Allocated by the Driver

If the camera pin is running under CSPROPERTY_BUFFER_DRIVER mode, buffers are allocated by the driver. The buffer state includes three mode: Idle, Busy, and Filled. The camera driver uses a queue to keep the buffer state, which means if one buffer is in the Idle Queue, it is in the Idle State. Figure 7-3 shows the buffer state diagram for this mode.

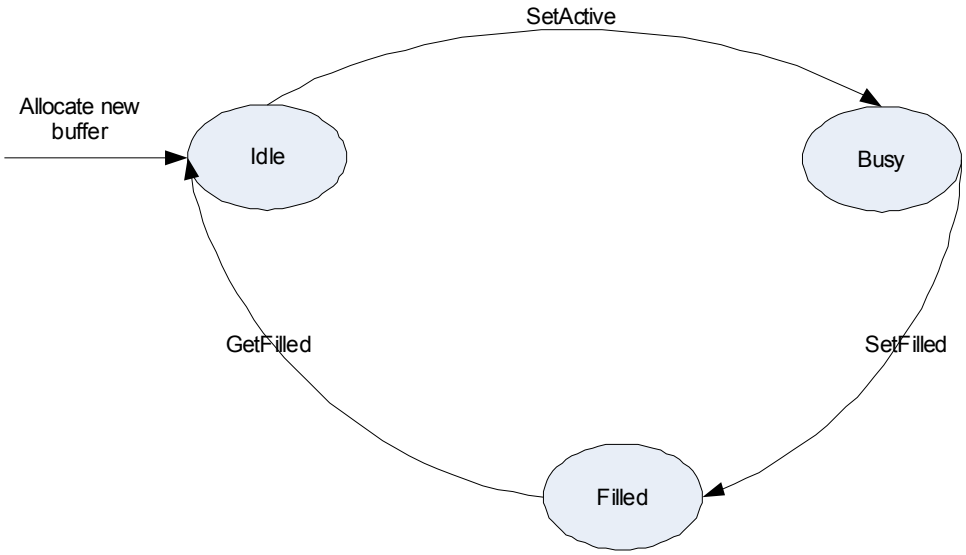


Figure 7-3. CSPROPERTY_BUFFER_DRIVER Mode Buffer State Diagram

- Idle Queue—Once a buffer is allocated by driver, it is in the idle queue. Otherwise, the filled buffer is used by user and this buffer is set to the idle queue. GetFilled or Allocate new buffer can set one buffer to Idle.
- Busy Queue—Once a buffer is set to IDMAC, it is in the busy queue and hardware begins using this buffer. SetActive can be used to transfer one buffer from Idle to Busy.
- Filled Queue—Once the IDMAC interrupt is received, the buffer is filled with frame data and it is in a filled queue. SetFilled can be used to transfer one buffer from Busy to Filled.

Once a buffer is allocated, it must be in one and only one queue, until it is free.

The following steps illustrate the process of the driver allocated buffers:

1. Application allocates a buffer using IOCTL_CS_BUFFERS and specifying CS_ALLOCATE.
2. MDD receives IOCTL, allocates buffer for the MDD layer, then calls PDD allocate interface to inform the PDD to allocate the buffer.
3. PDD calls the proper module allocate interface to allocate the buffer according to the PIN type. PDD allocated buffers are all in Idle queue.
4. When the module begins to operate, it checks if there are any buffers in the Idle queue. If true, it gets a buffer (PHY address) from Idle Queue and sets this PHY address as the hardware output address. Then is sets this buffer to Busy Queue, which means this buffer is in use by the hardware.
5. When an interrupt from hardware is received, one buffer in Busy Queue is filled with image data. The module gets this buffer from the Busy Queue and sets this buffer to the Filled Queue. At the same time, step l is repeated to pipeline the chain.
6. After the buffer enters into the Filled Queue, the MDD callback function is called to get this filled buffer.
7. The MDD callback function calls GetFilled() through the PDD interface to get the filled buffer provided by module. After GetFilled() returns, the filled buffer transfers to the Idle Queue from Filled Queue to make it available for the next iteration.
8. The module copies the image data from the filled buffer to the MDD idle buffer and sends this filled MDD buffer to MsgQ shared with the application.
9. Application receives the filled image data by calling ReadMsgQ. It may use memcpy to copy image data from the MDD buffer to the application buffer.
10. Application processes the image data.
11. Application enqueues the MDD buffer to make it available for the next iteration for MDD layer with using IOCTL_CS_BUFFERS and specifying CS_ENQUEUE.

7.4.1.3.2 Buffer Allocated by the Client

If the camera pin is running under CSPROPERTY_BUFFER_CLIENT_UNLIMITED mode, the buffers are allocated by the client. Compared to buffer allocated by driver mode, this mode adds a new state for buffer state: locked state.

- Locked Queue—Once buffers are registered by the client, they are in locked queue. Because in buffer allocated by client mode, buffers are shared between driver and application, it needs a state

to synchronize the buffer access. The locked state means the application is using this buffer and the driver cannot use it. An Enqueue interface is used to give the buffer ownership back to the driver.

Figure 7-4 shows the buffer state diagram for this mode.

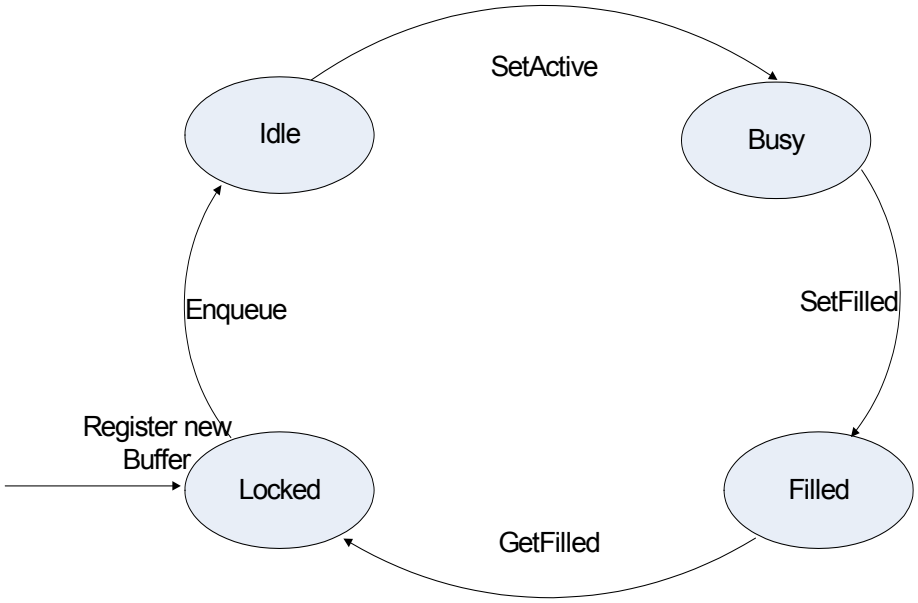


Figure 7-4. CSPROPERTY_BUFFER_CLIENT_UNLIMITED Mode Buffer State Diagram

The following steps describe the procedure of client allocating buffer:

1. Application allocates a buffer using IOCTL_CS_BUFFERS and specifying CS_ALLOCATE.
2. MDD receives the IOCTL, saves the buffer address as registered, then calls the PDD register interface to inform the PDD to register this buffer.
3. PDD calls proper module register interface to register the buffer for this module according to the PIN type. After registering, the buffer is in Locked queue and is owned by the application.
4. Application enqueues the buffer using IOCTL_CS_BUFFERS and specifying CS_ENQUEUE.
5. MDD calls the PDD Enqueue interface to enqueue the buffer.
6. PDD calls the proper module Enqueue interface to enqueue this buffer. After Enqueue, the buffer is in Idle queue, means it is owned by the driver.
7. When the module begins to operate, it checks if there are any buffers in the Idle queue. If true, it gets a buffer (PHY address) from Idle Queue and sets this PHY address as the hardware output address. Then is sets this buffer to Busy Queue, which means this buffer is in use by the hardware.
8. When an interrupt from hardware is received, one buffer in Busy Queue is filled with image data. The module gets this buffer from the Busy Queue and sets this buffer to the Filled Queue. At the same time, step l is repeated to pipeline the chain.
9. After the buffer enters into the Filled Queue, the MDD callback function is called to get this filled buffer.

10. The MDD callback function calls GetFilled() through the PDD interface to get the filled buffer provided by module. After GetFilled() returns, the filled buffer transfers to the Idle Queue from Filled Queue to make it available for the next iteration.
11. For the buffer sharing between all three layers, no memcpy from the module buffer to MDD buffer is required. The MDD determines if the buffer has been enqueued. If true, it sends this filled buffer to MsgQ shared with the application. Otherwise, it fails.
12. For the buffer sharing between all three layers, no memcpy from the MDD buffer to the application buffer is required. The application receives the filled image data by calling ReadMsgQ.
13. Application processes the image data.
14. Application calls the Enqueue interface to make it available for the next iteration for MDD.
15. MDD calls the Enqueue interface to make it available for the next iteration for PDD.
16. PDD calls the proper module Enqueue interface to make it available for the next iteration for module.

7.4.2 Communicating with the Camera

Communication with the camera driver is accomplished through Camera APIs defined by Microsoft for Windows Embedded Compact 7. Applications may access these Camera APIs directly or through the DirectShow video capture support.

7.4.2.1 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 Compact 7 Help topic:

Windows Embedded Compact 7 > **Device Drivers** > **Camera Drivers** > **Camera Driver Reference**.

7.4.2.2 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 Compact 7 Help:

Windows Embedded Compact 7 > **Audio, Graphics and Media** > **DirectShow.**

7.4.3 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.

7.4.3.1 Registry Settings

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

```
#if (defined BSP_CMOS_OV3640 || defined BSP_CMOS_OV5642 || defined BSP_TVIN_ADV7180)
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Camera1]
```

```
   "Prefix"="CAM"
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:1

IF BSP_CMOS_OV3640
   "CameraId"=dword:0
ENDIF BSP_CMOS_OV3640

IF BSP_CMOS_OV5642
   "CameraId"=dword:2
ENDIF BSP_CMOS_OV5642

IF BSP_TVIN_ADV7180
   "CameraId"=dword:4
ENDIF BSP_TVIN_ADV7180

   "CSIInterface"=dword:0
    ;CameraId default is 0.
    ;    0=0v3640;
    ;    1,2,3 are reserved for sensor support;
    ;    4,5 for TVin support
    ;    9 for CSI Test Mode
    ;CSIInterface default is 0.
    ;    0=CSI1 Interface;
    ;    1=CSI2 Interface;
    ;    2 is reserved for both CSI Interface in case of dual camere support
   "IClass"=multi_sz: "{CB998A05-122C-4166-846A-933E4D7E3C86}",
                      "{A32942B7-920C-486b-B0E6-92A702A99B35}"

[HKEY_LOCAL_MACHINE\Software\Microsoft\DirectX\DirectShow\Capture1]
   "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

#if (defined BSP_CSI_TESTMODE)
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Camera2]
   "Prefix"="CAM"
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:2

IF BSP_CSI_TESTMODE
   "CameraId"=dword:9
ENDIF BSP_CSI_TESTMODE

   "CSIInterface"=dword:1
    ;CameraId default is 0.
    ;    0=0v3640;
    ;    1,2,3 are reserved for sensor support;
    ;    4,5 for TVin support
```

```
    ;    9 for CSI Test Mode
    ;CSIInterface default is 0.
    ;    0=CSI1 Interface;
    ;    1=CSI2 Interface;
    ;    2 is reserved for both CSI Interface in case of dual camere support
   "IClass"=multi_sz: "{CB998A05-122C-4166-846A-933E4D7E3C86}",
                      "{A32942B7-920C-486b-B0E6-92A702A99B35}"

[HKEY_LOCAL_MACHINE\Software\Microsoft\DirectX\DirectShow\Capture2]
   "Prefix"="PIN"
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:2
   "PinCount"=dword:2 ;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
```

7.5 Power Management

The camera driver consumes power primarily through the operation of various IPUv3 sub-modules, such as the CSI, SMFC and the IC. The CSI, SMFC and IC modules are enabled when the camera device is set to a running state. Support for transitioning to the Suspend and Resume states is provided through the IOCTL_POWER_SET IOCTL.

7.5.1 PowerUp

This function is not implemented for the camera driver.

7.5.2 PowerDown

This function is not implemented for the camera driver.

7.5.3 IOCTL_POWER_SET

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

These states are handled in the following manner:

- D0—Action is only taken when resuming from the D4 state. If the camera is running when the transition to the D4 state occurs, the camera returns to a running state, re-enabling the sensor and IPUv3 submodules.
- 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 Compact 7, there are CTK tests written and provided by Microsoft.

The Camera CTK tests include the following:

- Camera and DirectShow Graph Integration Test — The Graph Building Tests test building a graph with all components and building graphs with single components.
- Camera Driver Data Structure Verification — The Camera Driver Data Structure Verification Test queries the camera driver for various properties and formats.
- Camera Driver Name Tests — The Camera Driver Name Tests ensure that the Camera Driver uses the right GUID and naming convention for DShow based drivers, pins and third party drivers.
- Camera Driver Preview and Capture Stream Functionality Verification — The Camera Driver Preview and Capture Stream Functionality Verification Test verifies the functionality of the still, preview and capture streams on the camera driver.
- Camera Performance Certification Test — This Test suite collects the Performance data of Camera on the Windows Embedded Compact device.
- Camera Performance Test — The Camera Performance Test requires a Windows Embedded Compact device with camera functionality. The Camera Performance Test gathers performance data for various DirectShow capture scenarios.
- Camera Quality Verification Test — The Camera Quality Verification test offers semi-automated verification of video data delivered by the camera driver. The test exercises supported resolutions, formats, and orientations as well as supported camera controls and video properties, such as controlling zoom, brightness, and contrast.
- Video Capture Filter Test — The Video Capture Filter Test tests directly the video capture filter, which exercises the camera driver. This test is ideal for component level testing of the camera driver via DirectShow, without involving the entire video capture graph.

Additionally, for Windows Embedded Compact 7, a camera application may be used to preview and capture 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	ADV7180 Tvin Sensor

7.6.2 Unit Test Software

7.6.2.1 CTK 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 CameraGrabber.dll	Library containing the “Camera and DirectShow Graph Integration Test “ cases
CamTestProperties.dll	Library containing the “Camera Driver Data Structure Verification” cases
CameraDriverNameTest.dll	Library containing the “Camera Driver Name Tests “ cases
CamIOTests.dll	Library containing the “Camera Driver Preview and Capture Stream Functionality Verification “ cases
cameraperfcerttest.dll	Library containing the “Camera Performance Certification Test “ cases
CameraPerfTests.dll	Library containing the “Camera Performance Test “ cases
cameraqualitytests.dll	Library containing the “Camera Quality Verification Test “ cases
Vidcaptest.dll	Library containing the “Video Capture Filter Test “ cases
camera.dll	Driver DLL file

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

7.6.2.2 Custom Camera Test

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

The `camapp1_preview.exe` and `camapp2_capture.exe` executable files are needed to validate dual camera driver.

7.6.2.3 Camera Application Test

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

7.6.3 Building the Unit Tests

7.6.3.1 CTK Test

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

`[Drive]:\Program Files\WindowsEmbeddedCompact7TestKit\tests\target\armv7`

7.6.3.2 Custom Camera Application Test

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

Build an OS image for the desired configuration:

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

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

CAMAPP uses GDI API to display a picture as default. CAMAPP also can support DDRAW to accelerate picture displaying. To use DDRAW, change the file `CameraWindow.cpp` under the folder `APP` as follows:

Change

```
//#define DIRECT_DRAW_MODE
```

to

```
#define DIRECT_DRAW_MODE
```

Camapp default works on Camera sensor Mode, for ARD Board, need switch camapp to TVIN mode.

To use TVIN mode, change the file `CameraWindow.cpp` under the folder `APP` as follows:

Change

```
//#define TVIN_MODE
```

to

```
#define TVIN_MODE
```

Then, repeat steps 4–6 listed above to build the custom camera application.

Another way to build the custom camera application is as follows:

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` to add the CAMAPP project

4. Right-click on the CAMAPP project and select `Rebuild`

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

When validate dual camera driver, please build the dual camera application following steps:

1. Add a new folder named `APP` under the folder ..\PLATFORM\<*Target Platform*>\SRC
2. Create an empty directory file under the folder ..\PLATFORM\<*Target Platform*>\SRC\APP
3. Copy the folders of `CAMAPP_Preview and CAMAPP_Capture` under the folder `SUPPORT\APP\Dual_Camera_App` to SRC\APP
4. Select the `Solution Explorer` of the Platform Builder Workspace window
5. Expand **Platform** > **<Target Platform>** > **Src** > **App**
6. Right-click on the CAMAPP_Preview folder and select `Rebuild`
7. Right-click on the CAMAPP_Capture folder and select `Rebuild`

The Dual camera application execution files (`camapp1_preview.exe camapp2_capture.exe`) are created copied to 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 CTK Test

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

Windows Embedded Compact 7 > **Compact Test Kit(CTK)** > **Multimedia-Camera Tests**

Use this command line to run the Camera and DirectShow Graph Integration Tests :

```
Tux -o -d cameragraphtests.dll
```

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

```
Tux -o -d camtestproperties.dll
```

Use this command line to run the Camera Driver Name Tests:

```
Tux -o -d cameradrivernametest.dll
```

Use this command line to run the Camera Driver Preview and Capture Stream Functionality Verification Tests:

```
Tux -o -d camiotests.dll
```

Use this command line to run the Camera Performance Certification Tests:

```
Tux -o -d cameraperfcerttest.dll
```

Use this command line to run the Camera Performance Test:

```
Tux -o -d cameraperftests.dll -c "-c \windows\capconfig.ini"
```

Use this command line to run the Camera Quality Verification Tests:

```
Tux -o -d cameraqualitytests.dll
```

Use this command line to run the Video Capture Filter Tests:

```
Tux -o -d vidcaptest.dll
```

NOTE

Please run camera CTK for Camera1 and Camera2 separately. If run CTK for Camera1, make sure no BSP environment variable is selected under CSI1, and when run for Camera2, make sure no BSP environment variable is selected under CSI0.

Please run camera CTK and CamApp separately. If you want to run CTK, make sure it runs before CamApp runing, and at that time, no CamApp is running.

The camera CTK requires some system DLLs and environment variables. Check that the variables listed below are selected. If these variables are not selected, select them and Sysgen the image.

```
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
```

The performance test requires the configuration file `capconfig.ini` which specifies what to test. Before testing, copy this file under the corresponding folder such as `\release` from the following location:

```
[Drive]:\Program Files\WindowsEmbeddedCompact7TestKit\tests\target\armv7
```

For ARD board, Tvin Driver is a special customized camera driver, so CTK test isn't needed.

Some CTK Camera Tests fail:

- Camera Performance Certification Tests subcase#201, #300 and #400 failed.

7.6.4.1.2 Running the Custom Camera Application Test

The following command executes the Custom Camera Application:

```
camapp.exe
```

Then application will show sensor image on the screen as default by using camera1 preview pin, and following operations are available: change resolution, rotate image, still image and save it, and so on.

Following commands for dual camera appliciation:

```
camapp1_preview.exe
```

```
camapp2_capture.exe
```

As default, `camapp1_preview.exe` will show sensor image on the screen by using camera1 preview pin, because of using CSI test mode, `camapp2_capture.exe` will show a chess board on the screen by using camera2 capture pin.

7.7 Camera Driver API Reference

For the camera driver API reference, see the Windows Embedded Compact 7 documentation. For reference information on basic camera driver functions, methods, and structures, see the Windows Embedded Compact 7 Help:

Windows Embedded Compact 7 > **Device Drivers** > **Camera Drivers** > **Camera Driver Reference**.

Chapter 8
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.

8.1 CAN Driver Summary

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

Table 8-1. CAN Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\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 -> BSP -> Freescale i.MX53 ARD:ARMV7 -> Device Drivers -> CANBUS
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_CANBUS1=1 BSP_CANBUS2=1

8.2 Supported Functionality

The CAN driver enables the Hardware 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

8.3 Hardware Operation

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

8.3.1 Conflicts with Other Peripherals and Catalog Items

8.3.1.1 Conflicts with SoC Peripherals

No conflicts.

8.3.1.2 Conflicts with ARD Peripherals

No conflicts.

8.4 Software Operation

8.4.1 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.

8.4.2 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.

8.4.3 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);
```

8.4.4 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);
```

8.4.5 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.

8.4.6 Power Management

8.4.6.1 PowerUp

This function is not implemented for the CAN driver.

8.4.6.2 PowerDown

This function is not implemented for the CAN driver.

8.4.6.3 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.

8.4.6.4 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}"
```

8.4.6.5 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.

8.4.7 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
```

8.5 Unit Test

The CAN unit 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.

8.5.1 Unit Test Hardware

The i.MX53 ARD board include CANBUS1 controller and CANBUS2 controller.So we can connected to the data exchange is tested between the two controller (one Board). The CANBUSs are not connect directly. An external transceiver on board is needed. The i.MX53 ARD board already contains this transceiver. The two controller transceiver must be connected by the CAN port (using an serial invert female-female).

8.5.2 Unit Test Software

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

Table 8-2. Software Requirements

Requirement	Description
CANApp.exe	Test file

8.5.3 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: `\WINCE700\SUPPORT\TEST\CANBUS\CANApp`
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 CANApp.exefile is located in the $(_FLATRELEASEDIR) directory.

8.5.4 Running the Unit Tests

On the tested board run the application with this command `CANApp.exe -r and CANApp.exe -s`

Chapter 9
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)

9.1 CSPDDK Driver Summary

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

Table 9-1. CSPDDK Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFROM\COMMON\SRC\SOC\COMMON_FSL_V3\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=

9.2 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)

9.3 Hardware Operation

Refer to the *i.MX53 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

Refer to the *i.MX53 Applications Processor Reference Manual* for possible conflicts

9.3.1.2 Conflicts with Board Peripherals

No conflicts.

9.4 Software Operation

9.4.1 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.

9.4.2 Compile-Time Configuration Options

The CSPDDK exposes compile-time options for configuring the SDMA support. In some cases, these compilation variables are also leveraged by driver code to expose a central point of controlling SDMA functionality. Table 9-2 describes the available CSPDDK compile options.

Table 9-2. CSPDDK Configurable 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 9-2. CSPDDK Configurable 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.Set to IMAGE_WINCE_IRAM_SDMA_PA_START to use internal RAM or IMAGE_WINCE_DDKSDMA_PA_START to use external RAM.
BSP_SDMA_CHNPRI_xxx	bsp_cfg.h	Assigns a SDMA channel priority to the respective peripheral. Refer to the individual driver chapters for more information on the specific priorities.
BSP_SDMA_SUPPORT_xxx	bsp_cfg.h	Boolean to specifies if SDMA-based transfers are enabled for each respective peripheral. Refer to the individual driver chapters for more information on the DMA support provided.

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

To decrease power consumption in system uses cases such as audio playback, it is beneficial to configure BSP_SDMA_MC0PTR to point to a reserved internal RAM (IRAM) region and allocate the audio buffers in IRAM. This configuration does not require external memory cycles in the data flow from the audio buffers to the SSI and allows the CSPDDK to utilize EMI clock gating to significantly reduce the power consumption. Refer to Chapter 5, “Audio Drivers“,” for more information on configuring audio DMA support.

9.4.3 Registry Settings

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

9.4.4 Power Management

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

9.5 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.

9.5.1 Unit Test Hardware

Table 9-3 lists the required hardware to run the unit tests.

Table 9-3. Hardware Requirements

Requirement	Description
No additional hardware required	

9.5.2 Unit Test Software

Table 9-4 lists the required software to run the unit tests.

Table 9-4. 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

9.5.3 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: `\WINCE700\SUPPORT\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.

9.5.4 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 9-5 describes the test cases contained in the DDK_SDMA tests.

Table 9-5. 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.

9.6 CSPDDK DLL Reference

9.6.1 CSPDDK DLL System Clocking (DDK_CLK) Reference

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

9.6.1.1 DDK_CLK Enumerations

Table 9-6. 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_CKO1_SRC	Clock output source one (CKO1) signal selections
DDK_CLOCK_CKO2_SRC	Clock output source two (CKO2) signal selections
DDK_CLOCK_CKO_DIV	Clock output source (CKO) divider selections
DDK_CLOCK_OVERRIDE_ENABLE_INDEX	Index for referencing the corresponding clock enable signal to be overridden

Table 9-6. DDK_CLK Enumerations (continued)

DDK_CLOCK_OVERRIDE_MODE	Clock enable signal override mode supported by CCM Enable Override Register
DDK_CLOCK_BRM_INDEX	Index for BRM
DDK_DVFC_SETPOINT	Frequency/voltage setpoints supported by the DVFC driver

9.6.1.2 DDK_CLK Functions

9.6.1.2.1 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

9.6.1.2.2 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

9.6.1.2.3 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

9.6.1.2.4 DDKClockSetFreq

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

```
BOOL DDKClockSetFreq(
```

```
            DDK_CLOCK_SIGNAL sig,
            UINT32 freq)
```

Parameters

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

9.6.1.2.5 DDKClockConfigBaud

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

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

Parameters

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

9.6.1.2.6 DDKClockSetCKO1

This function configures the clock output source 1 (CKO1) signal.

```
BOOL DDKClockSetCKO1(
            BOOL bEnable,
            DDK_CLOCK_CKO1_SRC index,
            DDK_CLOCK_CKO_DIV div)
```

Parameters

bEnable	[in] Set to TRUE to enable CKO1 output; set to FALSE to disable CKO1 output
index	[in] Selects the CKO1 source signal
div	[in] Specifies the CKO1 divide factor
Return Values	Returns TRUE if successful, otherwise returns FALSE

9.6.1.2.7 DDKClockSetCKO2

This function configures the clock output source 2 (CKO2) signal.

```
BOOL DDKClockSetCKO2(
            BOOL bEnable,
            DDK_CLOCK_CKO2_SRC index,
            DDK_CLOCK_CKO_DIV div)
```

Parameters

bEnable	[in] Set to TRUE to enable CKO2 output; set to FALSE to disable CKO2 output

index [in] Selects the CKO2 source signal

div [in] Specifies the CKO2 divide factor

Return Values Returns TRUE if successful, otherwise returns FALSE

9.6.1.2.8 DDKClockSetOverride

This function sets the override mode for clock enable mode.

```
BOOL DDKClockSetOverride(
        DDK_CLOCK_OVERRIDE_ENABLE_INDEX index,
        DDK_CLOCK_OVERRIDE_MODE mode)
```

Parameters

index [in] Index for referencing the clock enable signal

mode [in] Requested override mode for the clock enable signal

Return Values Returns TRUE if successful, otherwise returns FALSE

9.6.1.2.9 DDKClockGetOverride

This function gets the override mode for clock enable mode.

```
BOOL DDKClockGetOverride(
        DDK_CLOCK_OVERRIDE_ENABLE_INDEX index,
        DDK_CLOCK_OVERRIDE_MODE *mode)
```

Parameters

index [in] Index for referencing the clock enable signal

pMode [out] Pointer to the buffer to save current override model for clock enable signal

Return Values Returns TRUE if successful, otherwise returns FALSE

9.6.1.2.10 DDKClockSetBRM

This function setting BRM value.

```
BOOL DDKClockSetBRM(
        DDK_CLOCK_BRM_INDEX index,
        UINT32 brmVal)
```

Parameters

index [in] Specifies the clock to set BRM value

`brmVal` [in] Value of BRM.

Return Values Returns TRUE if successful, otherwise returns FALSE

9.6.1.2.11 DDKClockGetBRM

This function retrieve the BRM Setting Value

```
BOOL DDKClockGetBRM(
        DDK_CLOCK_BRM_INDEX index,
        UINT32 *pbrmVal)
```

Parameters

index [in] Specifies the clock to set BRM value

`pbrmVal` [out] Point to Value of BRM

Return Values Returns TRUE if successful, otherwise returns FALSE

9.6.1.2.12 DDKClockSetpointRequest

This function requests the DVFC driver to transit 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

9.6.1.2.13 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

9.6.1.2.14 DDKClockGetSharedConfig

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

```
PDDK_CLK_CONFIG DDKClockGetSharedConfig(VOID)
```

Parameters None

Return Values Returns a pointer to the clock configuration data structure

9.6.1.2.15 DDKClockLock

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

```
VOID DDKClockLock(VOID)
```

Parameters None

Return Values None

9.6.1.2.16 DDKClockUnLock

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

```
VOID DDKClockUnLock(VOID)
```

Parameters None

Return Values None

9.6.1.3 DDK_CLK Examples

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

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

UINT32 freq;

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

9.6.2 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.

9.6.2.1 DDK_GPIO Enumerations

Table 9-7. 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

9.6.2.2 DDK_GPIO Functions

9.6.2.2.1 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

9.6.2.2.2 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

9.6.2.2.3 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

9.6.2.2.4 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

9.6.2.2.5 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

9.6.2.2.6 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

9.6.2.2.7 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

9.6.2.2.8 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

9.6.2.3 DDK_GPIO Example

Example 9-3. 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);
```

9.6.3 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.

9.6.3.1 DDK_IOMUX Enumerations

Table 9-8. 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
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.

Table 9-8. DDK_IOMUX Enumerations (continued)

Programming Element	Description
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_INMODE	Specifies the CMOS/open drain mode for a pad; if no DDR_INPUT bit for a PAD, the DDK_IOMUX_PAD_INMODE_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_OUTVOLT	Specifies the output voltage mode for a pad; if no HVE bit for a PAD, the DDK_IOMUX_PAD_OUTVOL_NULL should be set
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

9.6.3.2 DDK_IOMUX Functions

9.6.3.2.1 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

9.6.3.2.2 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

9.6.3.2.3 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_INMODE inputMode,
        DDK_IOMUX_PAD_OUTVOLT outputVol)
```

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
inputMode	[in] Input mode (CMOS/DDR) configurationoutputVolt[in] Output voltage configuration
Return Values	Returns TRUE if successful, otherwise returns FALSE.

9.6.3.2.4 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_INMODE *pInputMode,DDK_IOMUX_PAD_OUTVOLT *pOutputVol)
```

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
pInputMode	[out] Input mode (CMOS/DDR) configurationpOutputVolt[out] Output voltage configuration
Return Values	Returns TRUE if successful, otherwise returns FALSE.

9.6.3.2.5 DDKIomuxSetGpr0

This function writes a value into IOMUX GPR0.

```
BOOL DDKIomuxSetGpr0(UINT32 data)
```

Parameters

data [in] Data to be written

Return Values Returns TRUE if successful, otherwise returns FALSE

9.6.3.2.6 DDKIomuxGetGpr0

This function read a value from IOMUX GPR0.

```
UINT32 DDKIomuxGetGpr0(VOID)
```

Return Values Returns IOMUX GPR0 value

9.6.3.2.7 DDKIomuxSetGpr1

This function writes a value into IOMUX GPR1.

```
BOOL DDKIomuxSetGpr1(UINT32 data)
```

Parameters

data [in] Data to be written

Return Values Returns TRUE if successful, otherwise returns FALSE

9.6.3.2.8 DDKIomuxGetGpr1

This function read a value from IOMUX GPR1.

```
UINT32 DDKIomuxGetGpr1(VOID)
```

Return Values Returns IOMUX GPR1 value

9.6.3.2.9 DDKIomuxSetGpr2

This function writes a value into IOMUX GPR2.

```
BOOL DDKIomuxSetGpr2(UINT32 data)
```

Parameters

data [in] Data to be written

Return Values Returns TRUE if successful, otherwise returns FALSE

9.6.3.2.10 DDKIomuxGetGpr2

This function read a value from IOMUX GPR2.

```
UINT32 DDKIomuxGetGpr2(VOID)
```

Return Values Returns IOMUX GPR2value

9.6.3.2.11 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

9.6.3.3 DDK_IOMUX Examples

Example 9-4. 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 9-5. 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, and remainder fields are invalid for GPIO1_5.
DDKIomuxSetPadConfig(DDK_IOMUX_PIN_GPIO1_5, DDK_IOMUX_PAD_SLEW_FAST,
DDK_IOMUX_PAD_DRIVE_HIGH, DDK_IOMUX_PAD_OPENDRAIN_NULL, DDK_IOMUX_PAD_PULL_NULL,
DDK_IOMUX_PAD_HYSTERESIS_NULL, DDK_IOMUX_PAD_INMODE_NULL,
DDK_IOMUX_PAD_OUTPUT_NULL);
```

9.6.4 CSPDDK DLL SDMA (DDK_SDMA) Reference

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

9.6.4.1 DDK_SDMA Enumerations

Table 9-9. DDK_SDMA Enumerations

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

9.6.4.2 DDK_SDMA Functions

9.6.4.2.1 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

9.6.4.2.2 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

9.6.4.2.3 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

9.6.4.2.4 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

9.6.4.2.5 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

9.6.4.2.6 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

9.6.4.2.7 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

9.6.4.2.8 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

9.6.4.2.9 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

9.6.4.2.10 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

9.6.4.2.11 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

9.6.4.2.12 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

9.6.4.2.13 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 10
Display Driver for IPUv3

The Windows Embedded Compact 7 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 v3 (IPUv3).

The MX53 ARD supports the following display types:

- Toshiba XGA LVDS panel
- HannStar XGA LVDS panel
- VGA analog output
- D1 TV Output following the NTSC or PAL television standard
- 720p TV Output following the 720p60 or 720p50 television standard
- 1080i TV Output following the 1080i30 television standard

10.1 Display Driver Summary

Table 10-1 identifies the source code location, library dependencies and other BSP information.

Table 10-1. Display Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\IPUV3\DISPLAY
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SoC*>\IPUV3\DISPLAY
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\IPUV3\DISPLAY
Driver DLL	ddraw_ipu.dll
SDK Library	N/A
Catalog Items	Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > Display > Display Port0 > IPU Support for the LVDS1 Panel Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > Display > Display Port1 > IPU Support for the LVDS2 Panel Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > Display > Display Port1 > IPU Support for VGA output Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > Display > Display Port1 > TVE Output Support

Table 10-1. Display Driver Summary (continued)

SYSGEN Dependency	SYSGEN_DDRAW=1
BSP Environment Variables	BSP_NODISPLAY= BSP_DISPLAY_LVDS1 = 1 for LVDS Panel on display port 0 BSP_DISPLAY_LVDS2 = 1 for LVDS Panel on display port 1 BSP_DISPLAY_VGA = 1 for VGA Output BSP_DISPLAY_TVE = 1 for TV Encoder support

10.2 Supported Functionality

The display driver provides the following software and hardware support:

1. Variety of display types and resolutions (see Section 10.3.2, “Display Configurations.”)
2. Dual simultaneous output for two display devices (see Section 10.4.3.2, “Dual Display Support.”)
3. RGB565, RGB888 and RGB8888 frame buffer pixel format
4. DirectDraw Hardware Abstraction Layer (DDHAL)
5. Up to three overlay surfaces
6. One overlay surface on each display when two displays are on (two active overlay surfaces total)
7. Video overlays containing image data in any of the following FOURCC pixel formats:
   - RGB565
   - UYVY
   - YV12
   - NV12
8. Hardware-accelerated color space conversion in video overlays
9. Hardware-accelerated image resizing in video overlays, resizing ratios ranging from 1:8 to 1000:1
10. Overlay surface color keying
11. Alpha blending with an overlay surface, through use of a global alpha value
12. Alpha blending with an overlay surface containing per-pixel alpha data (only ARGB8888 format)
13. Cropping of an overlay surface
14. Screen rotation of 0°, 90°, 180°, or 270°
15. Gamma correction support for dumb display device (DVI, LVDS)
16. De-interlacing of a video overlay
17. CVBS, S-Video, YPbPr and RGB TV output mode

The following limitations apply to the display driver overlay support:

1. The dimensions of the overlay surface may not exceed 2048x2048
2. The width of the overlay surface must conform to an 8-pixel alignment restriction
3. The minimum width (or height if screen is rotated) of an overlay surface is 8 pixels
4. The minimum height (or width if screen is rotated) of an overlay surface is 8 pixels
5. Overlays are not supported when using a rotated screen with a resolution larger than 1024×768

6. When using the cropping feature, the x coordinate position must conform to the 8-pixel alignment restriction
7. When using the cropping feature with a surface using the YV12 pixel format, the x coordinate position must conform to 16-pixel alignment restriction and the y coordinate position must conform to 4-pixel alignment restriction
8. For a display using interlaced output (for example NTSC/PAL TV), the target overlay surface must have an even surface height
9. The area of overlay surface must be divisible by 32 for YV12 format

While the display driver is in output mode whose resolution is larger than 1024x1024, the following supported features become unavailable due to the limited bandwidth and increased system loading associated with these modes:

- Dual simultaneous output to an LCD
- Support for more than one active overlay surface
- Screen rotation of 90°, 180°, or 270°
- Cropping of an overlay surface

10.3 Hardware Operation

For operation and programming information, see the chapter on the IPUv3 in the i.*MX53Applications Processor Reference Manual*.

10.3.1 IPUv3 Overview

The low-level operation of the display driver is based on the IPUv3. The IPUv3 is broken down into functional submodules. The following list describes the function each of these submodules:

- Control Module (CM)—Provides control and synchronization for the entire IPUv3
- Image DMA Controller (IDMAC)—Transfers data to and from system memory
- Display Processor (DP)—Performs the processing required for data sent to display, including color space conversion and image combining
- Image Converter (IC)—Performs resizing, color conversion, combining with graphics, and horizontal inversion
- Image Rotator (IRT)—Performs rotation (90° or 180°) and inversion (vertical or horizontal)
- Video De-Interlacer (VDI)—Performs de-interlacing of interlaced video content
- Display Interface 0 and 1 (DI0/DI1)—Provides interface to displays, display controllers, and related devices
- Display Controller (DC)—Controls the display ports
- Display Multi-FIFO Controller (DMFC)—Controls FIFOs for IDMAC channels related to the display system

The IPUv3 also contains regions of internal memory that store information used in the operation of the IPUv3.

- Task Parameter Memory (TPM)—Holds color space conversion coefficients and offsets
- Channel Parameter Memory (CPMEM)—Holds configuration information for each IDMAC channel
- Look-Up Table (LUT)—Holds a table of look-up values, providing support for palettized pixel formats

10.3.2 Display Configurations

The IPUv3 features two display ports each capable of generating output for one display. The platform catalog allows for the selection of only one display type for each display port—Display Port 0 and Display Port 1. Choosing a configuration that includes a display for both Display Port 0 and Display Port 1 allows the use of dual display mode.When a display is selected for both display ports, the display device on Display Port 0 is the default display device and is the only display that will be active when the system boots up (the display device on Display Port 1 will be turned off by default). See Section 10.4.2.1.2, "Changing To Dual Display Mode," and Section 10.4.3.2, "Dual Display Support," for details on configuring and changing to dual display mode. The catalog, and thus the OS image, may also be configured to select a display from only one of the display ports. Choose this configuration to switch between different display types or supporting multiple simultaneous displays.

10.3.2.1 i.MX53 ARD

The following displays and resolutions may be selected for the i. MX53 ARD:

- Display Port 0
  - Toshiba 8.4" XGA LVDS display (LT084AC37100)—1024×768
  - HannStar 10" XGA LVDS display (HSD100PXN1)—1024×768
- Display Port 1
  - HannStar 10" XGA LVDS display (HSD100PXN1)—1024×768
  - VGA analog output—800×600, 1024×768, 1280×1024, 1680×1050
  - TV Output support for NTSC and PAL standard televisions and 720p and 1080i HDTVs—720×480, 720×576, 1280×720, 1920×1080

10.3.3 Conflicts with Other Peripherals and Catalog Items

10.3.3.1 Conflicts with SoC Peripherals

No conflicts.

10.3.3.2 i.MX53 ARD Peripheral Conflicts

There are pin confliction between VGA port, Ethernet port and Nor Flash. When VGA port is selected, LAN9220 component must be disabled and Nor Flash can not be used as storage. Also the jumpers J11 and J12 on ARD CPU board must jump to [1,2].

Note: The detail hardware information must be adjusted according to hardware update. The information above is from ARD CPU board version 1.0.

10.4 Software Operation

10.4.1 Software Architecture

10.4.1.1 Software Driver Components

Figure 10-1 shows the relationship between software components in the display driver architecture.

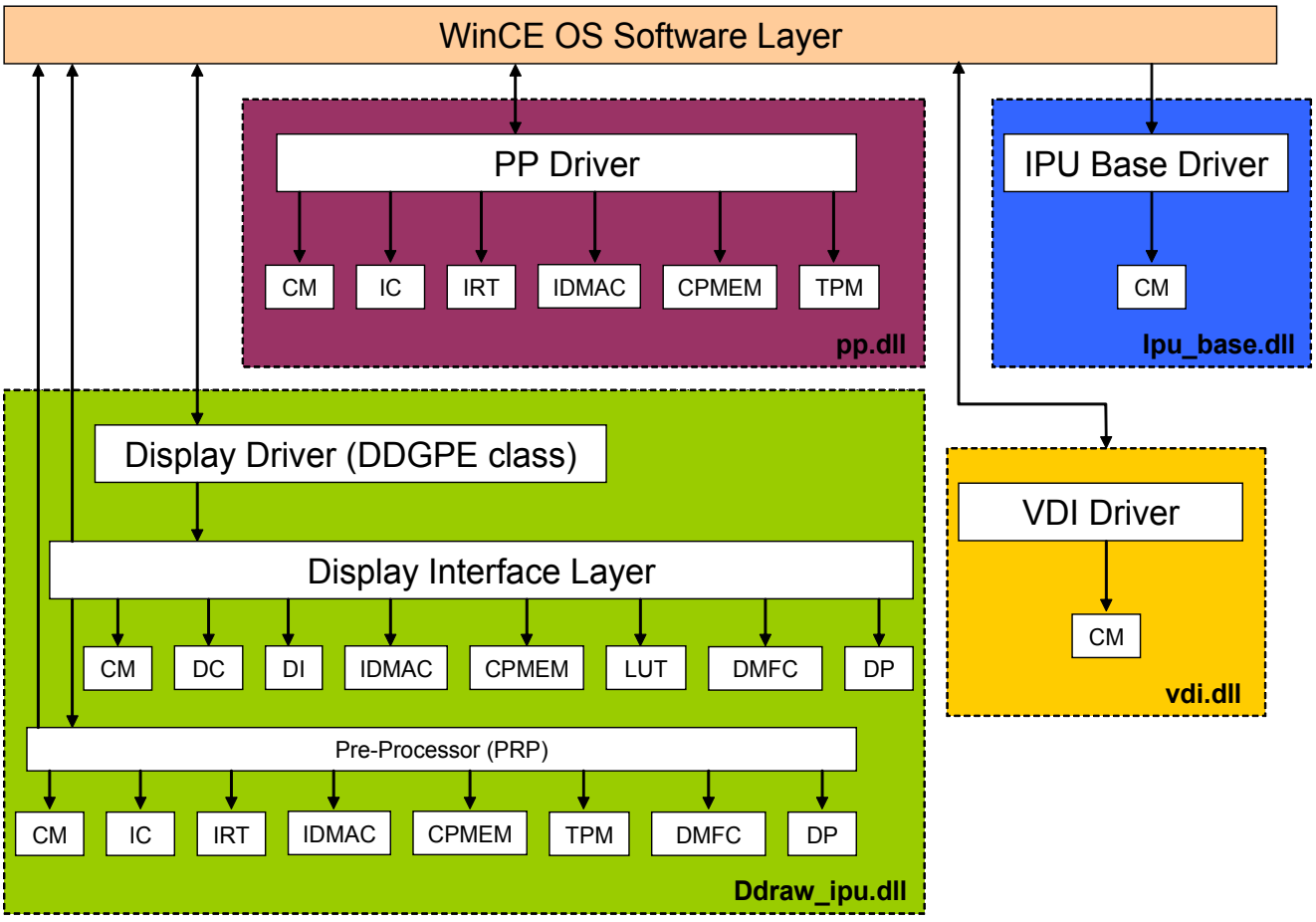


Figure 10-1. Software Architecture

Figure 10-1 shows the main elements of the display driver architecture:

- Display Driver—The high level DDGPE-based display driver. Contains implementations for DirectDraw APIs
- Display Interface Layer—Set of functions that performs high-level display operations (DisplaySetSrcBuffer, DisplayUpdate) and retrieves display information (DisplayGetPixelDepth, DisplayGetSupportedModes)
- Pre-processor Driver (PRP)—Sub-driver dedicated to the display driver that performs the following processing tasks: color space conversion, resizing, rotation, and combining
- Post-processor Driver (PP)—General purpose image processing driver that performs the following processing tasks: color space conversion, resizing, rotation, and combining
- Video De-Interlacer Driver (VDI)—Driver for the IPUv3 video de-interlacing hardware block, which processes interlaced video fields and outputs progressive video frames
- IPUv3 Base Driver —Stream interface driver that controls the allocation of buffers from video memory. This driver also completes all IPUv3 interrupt handling
- Low-Level APIs (IPUv3 Submodules)—Functions that provide access to IPUv3 registers and internal memories

10.4.1.1.1 Display Driver

The display driver is the top level interface between the display driver and the Windows CE OS or a calling application. This top level software component is composed of the DDIPU class, which is derived from the public DDGPE class and inherits the underlying GPE driver functionality. Graphics Device Interface (GDI) and DirectDraw APIs are implemented at this level, and calls are made into the Display Interface Layer to retrieve display information, enable and disable the display, and control what is sent to the display.

10.4.1.1.2 Display Interface Layer

The Display Interface Layer provides the main parts of the display driver. It handles requests from the Display Driver and manages a number of IPUv3 submodules in order to control what is sent to the active display devices. The tasks that this component performs include the following:

- Retrieving display information (for example supported modes, pixel formats)
- Handling requests to allocate video memory
- Initializing, enabling, and disabling display panels
- Initializing, enabling, and disabling IPUv3 submodules
- Handling requests to update the UI contents on the display
- Handling requests to update the overlay contents on the display
- Managing processing tasks for an overlay surface

The Display Interface Layer interfaces with the IPUv3 driver, through the stream interface to handle requests to allocate video memory buffers. The Display Interface Layer interfaces with the CM to control flow of data to the display. The DI and DC are called to configure the display ports. The IDMAC, CPMEM, DMFC, and LUT are called to control the transfer of display data to through the IDMAC. The DP and PRP are accessed to process overlay surfaces and combine with the UI when an overlay surface is active.

10.4.1.1.3 Pre-Processor Driver

The Pre-Processor (PRP) driver provides the display driver with the means for performing the following processing tasks on an overlay surface:

- Resizing
- Combining of video and graphics data
- Rotation (90°)
- Vertical and horizontal flipping
- Color Space Conversion (CSC)
- Cropping

The PRP driver uses the IC and IRT submodules to perform these processing tasks. The PRP driver also accesses the DP to configure the processing flow through the IPUv3.

The PRP driver is the primary means for performing resizing, rotation, and cropping on an overlay surface. Although the PRP driver is capable of CSC and combining, this task is typically left to the DP submodule, which can more effectively perform these tasks.

10.4.1.1.4 Post-Processor Driver

The Post-Processing (PP) driver provides a general resource capable of performing a set of processing tasks on a surface. The PP is capable of performing the same set of processing tasks as the PRP driver:

- Resizing
- Combining of video and graphics data
- Rotation (90°)
- Vertical and horizontal flipping
- Color Space Conversion
- Cropping

The PP driver also uses the IC and IRT submodules to perform these processing tasks. The IC and IRT submodules provide time-sharing of tasks between the PRP and PP, so both drivers can perform a processing task simultaneously.

The PP driver is currently used within the display driver to aid in the resizing and combining of overlay surfaces when multiple overlay surfaces are active.

10.4.1.1.5 Video De-Interlacer Driver

The Video De-Interlacer (VDI) driver handles the task of converting de-interlaced video content into progressive video content. The VDI hardware applies a high-quality three-field motion-adaptive filter, which retains the full image resolution for slow motion video, while preventing motion artifacts in dynamic, fast motion video.

10.4.1.1.6 IPUv3 Base Driver

The IPUv3 base driver is accessed through a stream interface and serves two primary purposes in the operation of the display driver:

- Provides centralized management of video memory
- Provides centralized interrupt handling for the entire IPUv3

10.4.1.1.7 Low Level APIs

At the lowest level of software in the display driver architecture, register accesses exert direct control over the IPUv3 submodules. A library is created for each IPUv3 submodule containing the functions providing access to its registers. These functions are called from the Display Interface Layer, the PRP, the PP, and the IPUv3 Base driver.

10.4.1.2 Video Memory Requirements

Memory must be reserved for the following types of surfaces:

- UI Surfaces (Primary Surfaces)—Primary surface holding the graphics data that makes up the main User Interface screen, along with back buffers for the primary surface.
- Video Processing Surfaces, Stage 1—Internal buffers used by the display driver when processing video frames or other overlay surfaces. These buffers hold the output from the first processing task.
- Video Processing Surfaces, Stage 2—Additional buffers used for processing video frames. These buffers are only used in cases in which both rotation and resizing are required. In this case, a second set of buffers is needed to hold the output from the second processing task.
- Application Surfaces—Includes all surfaces created by applications, including buffers used in decoding video frames. The number and size of buffers in this category can vary greatly, so we attempt to construct a worst-case scenario, and add some additional buffering to that case.

Figure 10-2 shows the amount of memory required for each of the surfaces based on assumptions about how many surfaces are needed and the maximum resolutions for LCD and video content that are used in a hypothetical embedded system. For the application surfaces, an estimate has been made based on the size and number of surfaces needed to decode worst-case video content, and some additional buffering has been added to that to ensure space for additional surfaces. It is also important to note that the number of video processing surfaces multiplies with the number of simultaneous overlays that are used. Therefore, when developing a system that uses three simultaneous overlay surfaces, the number of video processing buffers increases proportionally.

Table 10-2. Surface Memory Requirements

Surface Type	Number of Surfaces	Maximum Size	Bytes
UI (Primary Surface)	1–3	LCD Size (VGA)	640x480x2x3 = 1.8 Mbytes
Video Processing, Stage 1	2	Max Video Size (D1)	720x576x2x2 = 1.6 Mbytes
Video Processing, Stage 2	2	LCD Size (VGA)	640x480x2x2 = 1.2 Mbytes
Application	N/A	N/A	~ 6 Mbytes
		Total	10.6 Mbytes

10.4.2 Communicating with the Display

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

10.4.2.1 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

10.4.2.1.1 Changing the Display Mode

The GDI function ChangeDisplaySettingsEx is used to change the display mode. For information and syntax on this function, see the Windows Embedded Compact 7 documentation:

Windows Embedded Compact 7> Shell and UI > Graphics, Windowing and Events (GWES) > GWES Reference > GDI Reference > GDI Functions > ChangeDisplaySettingsEx

In order to transition between display (e.g. LCD and TV) modes, ChangeDisplaySettingsEx must be called with the target width (dmPelsWidth) and target height (dmPelsHeight) equal to that for the desired display mode. If the target width and height do not match the width and height of one of the supported display modes, the ChangeDisplaySettingsEx call fails.

For example, when attempting to switch from LCD mode to NTSC TV mode, the dmPelsWidth should be set to 720 and the dmPelsHeight should be set to 480.

NOTE

There may be multiple display modes supported by the display driver that support the same resolution. To distinguish between these modes, the calling application should use the display frequency (no two display modes have the same resolution and frequency). The display frequency is set using the SET_DISPLAY_FREQUENCY DrvEscape code (see Section 10.4.2.3.1, “Setting the Display Frequency,”), and must be set before calling ChangeDisplaySettingsEx to change the mode.

10.4.2.1.2 Changing To Dual Display Mode

Display mode transitions may also trigger the enabling of dual display mode. In order for the display driver to allow a transition to dual display mode, the display driver must be configured and built with dual display supported (see Section 10.4.3.2, “Dual Display Support,”). Once a device with dual display support transitions from a display mode associated with Display Port 0 to a display mode associated with Display Port 1, dual display mode becomes active. At this point, the secondary primary surface is shown on the secondary display (the LCD transitioned from), and may be accessed through the steps described in Section 10.6.3, “Dual Display API.”

10.4.2.2 Using DirectDraw

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

Windows Embedded Compact 7 > Audio, Graphics and Media > DirectDraw

The following DirectDraw features are supported in the display driver by the IPUv3 hardware:

- Page flipping with one backbuffer
- Overlay surfaces using the RGBA, RGB, YV12, NV12, or UYVY pixel formats
- Multiple overlay surfaces, up to a maximum of five simultaneous surfaces
- 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
- Overlaying using a global alpha value
- Stretching of overlay surfaces

The IPUv3 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 is 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).

10.4.2.3 Using Display Driver Escape Codes

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

Windows Embedded Compact 7 > Device Drivers > Display Drivers > Display Driver Reference > Display Driver Functions > DrvEscape

10.4.2.3.1 Setting the Display Frequency

The display driver provides the following two driver escape codes to allow applications to set and query the display frequency:

- DISPLAY_SET_OUTPUT_FREQUENCY
- DISPLAY_GET_OUTPUT_FREQUENCY

The display frequency must be set in order to disambiguate between display modes that use the same resolution (for example 720p50 and 720p60). The display frequency should be set before calling ChangeDisplaySettingsEx to set the display mode. See Section 10.6.2.1, “DISPLAY_SET_DISPLAY_FREQUENCY Escape Code,” for information about how to use these APIs.

10.4.2.3.2 Video De-Interlacing

Since there is no other way to pass information about whether an overlay surface is interlaced through the DirectDraw API, video de-interlacing is enabled through the DISPLAY_IS_VIDEO_INTERLACED driver escape code. Video de-interlacing is primarily used when decoding and playing back interlaced video content, so the video playback application must use the driver escape code to request that the display driver enable interlaced video mode. Refer to Section 10.6.2.1, "DISPLAY_SET_DISPLAY_FREQUENCY Escape Code," for information about how to use the API to enable video de-interlacing.

10.4.2.4 Using The Display Driver Control Panel Application

A control panel application provides access to additional display driver functionality. Look for the icon shown in Figure 10-2 in Windows CE control panel.

Figure 10-2. Display Driver Icon

The control panel application supports the following display driver features:

- Rotation between 0°, 90°, 180°, and 270°
- Gamma correction configuration for a synchronous display device, The gamma value may be set between 0.5 and 3.5. The default gamma value is 1.0.
- Display mode configuration with a drop-down box listing all of the display modes supported by the display driver. Each display mode is listed as a combination of the mode width, height, and frequency. For example, 800×480@60Hz represents the WVGA panel LCD mode, and 720×480@50Hz represents NTSC TV mode. The resolution of the current mode is displayed in the box.

The GUI of the display driver control panel application is shown in Figure 10-3.

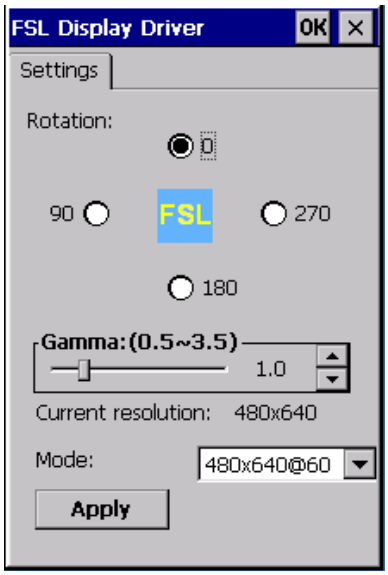


Figure 10-3. Display Driver GUI

NOTE

Because Windows Embedded Cmpact 7 only identifies display modes from the width, height, and frequency, the least significant digit of the frequency is varied to distinguish otherwise identical modes. For example, both the VGA and DVI display types support a 1024×768@60Hz mode, so the DVI mode is represented as 1024×768@60Hz and the VGA mode is represented as 1024×768@61Hz.

10.4.3 Configuring the Display

The primary means for configuring the display is through the selection of a display panel type in the Platform Builder catalog. The selection of a panel in the catalog causes a BSP environment variable to be selected, which ultimately leads to the inclusion in the OS image of a **PanelType** registry key. The **PanelType** registry key, which is described in Section 10.4.3.4, "16BPP and 32BPP are supported. Display Registry Settings," specifies the display panel that is being used to the display driver. The PanelType provides the display driver an index into an array containing all of the main display configuration information for the panel—panel resolution, timings, pixel mappings, and additional information.

10.4.3.1 Rotation Support

The DirectDraw display driver may be configured to allow screen rotation through a parameter in the bspdisplay.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

The rotation feature is disabled when the panel resolution is larger than 1024×768 or more than one panel is enabled.

10.4.3.2 Dual Display Support

The DirectDraw display driver may be configured to support dual independent displays. In dual display mode, a secondary display device may be enabled to display contents from a secondary frame buffer, which is independent from the primary frame buffer. Dual display support is configured through a parameter in the bspdisplay.h file. If the BSP_ENABLE_SECONDARY_PRIMARY_SURFACE parameter is set to TRUE, the DirectDraw display driver supports dual displays. If it is set to FALSE, it does not support dual displays.

NOTE

Due to a system bandwidth loading limitation, the dual display support feature is automatically disabled when one of display device's resolution is larger than 1024×768.

10.4.3.3 Display Driver Blit Acceleration

Two on-chip Graphics Processing Unit (GPU) cores, GPU2D and GPU3D, 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 these two GPU cores, and how to configure the BSP to enable acceleration through these GPU cores.

NOTE

GPU2D Graphic acceleration is enabled in default BSP. If customer doesn't need the hardware acceleration, BSP_DISPLAY_Z160 should be removed in platform environment variable table to disable it. And for no hardware GDI acceleration system, setting flag BSP_VID_MEM_CACHE_WRITETHROUGH can get additional graphic performance boost. But application needs to be careful about cache maintainance for all video memory access at that time.

10.4.3.3.1 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.

10.4.3.3.1.3 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.

10.4.3.3.1.4 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 CTK 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 CTK reference image(case #218,222).
- GPU2D fails the AlphaBlend CTK test(case #231). The color output after an alpha blend blit operation may have a single-bit mismatch when compared with the reference image.

10.4.3.3.2 GPU3D Graphics Acceleration

GPU3D 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 GPU3D component by setting the platform environment variable BSP_DISPLAY_Z430=1. This may be achieved by navigating to the project properties, and adding the environment variable in the Configuration Properties->Environment window.

10.4.3.3.2.5 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. PatBlt() - Pattern copy blits are accelerated.
5. The following accelerated ROP operations: BLACKNESS, PATCOPY, SRCCOPY, WHITENESS.

10.4.3.4 16BPP and 32BPP are supported. Display Registry Settings

Depending on the display panel catalog item(s) included in the OS design, a series of registry keys are optionally included in the OS image. These keys provide information to the display driver about the panel type, frame buffer format, and video memory size.

The following is a sample set of registry keys that might be included for a given display panel:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
   "Bpp"=dword:10                  ; RGB565
   "VideoBpp"=dword:20             ; RGB666 (32bpp internal)
   "VideoMemSize"=dword:2000000    ; 32MB

[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU\DI0]
   "PanelType"=dword:2
   "EnableOnBoot"=dword:1          ; TRUE
```

If a secondary display panel is selected from Display Port 1 (DI1), a similar set of registry keys is added under the [HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU\DI1] subkey. When panels from both DI0 and DI1 are selected, the panel under DI0 is the default panel upon device boot-up. If only a panel connected to DI1 is selected, that panel is the default panel upon device boot-up.

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

The C2DThreshold key controls the size of graphical blit operaitons that are accelerated by the GPU. When C2DThreshold is set, only size larger than C2DThreshold×C2DThreshold will be accelerated by the GPU.

In the following example, acceleration is enabled for pattern fill, line draw, stretchblt, and rotated screen cases whose operation size is larger than 100×100, while acceleration is disabled for solid color fill, simple bitblt, and maskblt:

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU]
       "C2DFlag"=dword:12a     ; Flag for c2d
       "C2DThreshold"=dword:64 ; 100
```

10.4.3.4.1 TV Out Register Setting

For TVE out register setting there have a specifical variable for TV output mode.

```
[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU\DI1]
         "TVOutputMode"=dword:6     ; TVE Component YPrPb for both SDTV and HDTV
```

The TVOutputMode settings might be configured as follows:

- 0 is for TVE standby
- 1 is for TVE composite on channel #0
- 2 is for TVE composite on channel #2
- 3 is for TVE composite on channel #0 and #2
- 4 is for TVE S-video on channel #0 and #1
- 5 is for TVE S-video on channel #0 and #1, and composite on channel #2
- 6 is for TVE component YPrPb on channel #0, #1 and #2
- 7 is for TVE component RGB on channel #0, #1 and #2

(Note: The default setting for TVOutputMode in TVEv2 is 6, i.e. component output).

10.4.4 Power Management

The display driver consumes power primarily through the operation of the display panel, and through the following IPUv3 sub-modules:

- Image Converter (IC)—performs color conversion and resizing on video data
- Image Rotation (IRT) submodule—performs rotation
- Display Processor—performs color space conversion and combining of video and graphics data

The display driver also controls the operation of TVE, LVDS module, calling to enable or disable the corresponding module and its clocks. 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.

10.4.4.1 PowerUp

This function is not implemented for the display driver.

10.4.4.2 PowerDown

This function is not implemented for the display driver.

10.4.4.3 IOCTL_POWER_SET

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

- D0—The display panel is enabled. The IPUv3 DI and DC modules are enabled to send data to the display. If video is active, additional submodules may also be enabled to process and convert video data. If TV output mode is active, enable the TVE module and its clocks.
- D4—The DI and DC submodules of the IPUv3 are disabled. The display panel is disabled. If TV output mode was enabled, disable the TVE and its clocks.

10.5 Unit Test

The display driver is subject to two test suites provided with the Windows Embedded Compact Test Kit (CTK): the Graphics Device Interface (GDI) Tests and the DirectDraw Tests. Additionally, video playback may be verified by using the PlayWnd application. The video de-interlacing functionality of the display driver may be tested through a custom CTK test suite.

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.

PlayWnd 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.

The Video De-Interlacing test reads from a sample input file containing interlaced video frames. These frames are de-interlaced and displayed to the screen.

10.5.1 Unit Test Hardware

The display driver unit tests require the inclusion of a display panel to display graphics and video data.

10.5.2 Unit Test Software

10.5.2.1 GDI Tests

Table 10-6 lists the software required to run the GDI tests.

Table 10-6. Software Requirements

Requirement	Description
Tux.exe	Test harness, required for executing the test .
Kato.dll	Logging engine, required for logging the test data .
Gdit.dll	Main test library .
Ddi_test.dll	Graphics Primitive Engine (GPE)®Cbased display driver that the GDI API uses to verify the success of each test case. If Ddi_test.dll is unavailable, run the test with manual verification.

10.5.2.2 DirectDraw Tests

Table 10-7 lists the software required to run the DirectDraw tests.

Table 10-7. Direct Draw Software Requirements

Requirement	Description
Tux.exe	Tux test harness, required for executing the test
Kato.dll	Kato logging engine, required for logging test data
Ddautoblt.dll	Test library for DDraw Blt test .
DDfunc.dll	Test library for DirectDraw Functionality Test
ddi.dll.cfg	Configuration file for supported driver capabilities. It is used in DirectDraw Functionality Test.
DDints.dll	Test library for DirectDraw Interface Tests
DDrawTK.dll	Test library for DirectDraw Verification Tests

10.5.2.3 Windows Media Player Tests

Table 10-8 lists the software required to perform WMV playback with Windows Media Player.

Table 10-8. Windows Media Player Software Requirements

Requirement	Description
PlayWnd. exe	Movie Player sample application
*.wmv sample video files	Sample windows media files

10.5.2.4 Multiple Overlay Custom Test

A custom test is provided to test the display driver support for multiple overlays. Table 10-9 lists the software required to run the MultipleOverlay test.

Table 10-9. Multiple Overlay Software Requirement

Requirement	Description
MultipleOverlay.exe	Multiple overlay sample test application

10.5.2.5 Video De-Interlacing Custom CTK Test

A custom test is provided to test the de-interlacing functionality of the display driver. Table 10-10 lists the software required to run the VDI test.

Table 10-10. Video De-Interlacing Software Requirement

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
Vdi_test.dll	VDI test .dll file
stefan_interlaced_320x240_30frames.yv12	Test input file containing Interlaced video input

10.5.3 Building the Unit Tests

10.5.3.1 GDI/DirectDraw Tests

The GDI and DirectDraw tests come pre-built as part of the CTK. No steps are required to build these tests. For information about the tests, see the Help:

Windows Embedded Compact 7 > Compact Test Kit (CTK)

10.5.3.2 PlayWnd Tests

For PlayWnd testing, there are no build steps required. playwnd.exe always is included in the image. Additionally, sample WMV files must be included in the image to demonstrate playback.

10.5.3.3 Multiple Overlay Custom Test

The MultipleOverlay application is included with the BSP release. To build the application complete the following steps:

1. Open the BSP sample solution in Microsoft Visual Studio
2. Click **Build OS** > **Open Release Directory** to open the command prompt
3. Navigate to the test directory: `\WINCE700\SUPPORT\APP\MultipleOverlay`
4. Build the application with the command **build -c**
5. The binary `MultipleOverlay.exe` is automatically copied into the release directory

10.5.3.4 Video De-Interlacing Custom CTK Test

To build the VDI test, 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. Navigate to the VDI test directory: `\WINCE700\SUPPORT\TESTS\VDI`
3. Enter **set WINCEREL=1** on the command prompt and press return.
   This copies the DLL to the flat release directory.
4. Execute the **build -c** to build the VDI test.

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

10.5.4 Running the Unit Tests

10.5.4.1 Running the GDI Tests

The command line for running the GDI tests is:

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

The NoResize and NoRotate command line flags must be included to prevent test failures caused by illegal mode changes. For information about the GDI tests and command line options, see the Platform Builder Help:

Windows Embedded Compact 7 > Compact Test Kit (CTK) > Display - GDI Tests

10.5.4.2 Running the DirectDraw Tests

Table 10-11 lists the command line for running the DirectDraw tests.

Table 10-11. The command line for running the DirectDraw tests

Command	Description
tux -o -d ddautoblt.dll	DDraw Blt test .
tux -o -d ddfunc.dll	DirectDraw Functionality Test
tux -o -d ddints	DirectDraw Interface Tests
`tux –o –d ddrawtk`	DirectDraw Verification Tests

NOTE

The display driver fails in the following DirectDraw functionality testcases : 304,404,526,540,556 and 640. The failure occurs for case 304, 404 because Ddraw capability flag can not cover some surface types hardware supported. The failure occurs for case 526 because YUV offscreen surface is not supported. The failure occurs for case 540, 640 is in Microsoft middleware code. The failure occurs for case 556 because the test case expected result is conflict with MSDN description which driver design follows.

The display driver is possible to fail in the following DirectDraw interface testcases : 518, 523, 771 and 1821. The failure may occur for case 518 when the test resolution is large than 1600x1200 which test case can't handle. The failure occurs for case 523 because NV12 fourcc value can't be recognized by test case but supported by hardware. The failure may occurs for case 771 when two display mode with the same resolution is supported by display driver. The failure may occur for case 1821 because hardware doesn't support 8x8 overlay surface size, the minimum overlay surface hardware can fully support is 16x16.

The display driver fails or aborts in the following DirectDraw verification testcases: 400, 410, 420, 430, 500, 502, 504, 506, 508, 510, 512, 514, 516, 518, 520, 1240, 1250, 1340 and 1350. The failure 400 ~ 520 occurs because flag DDSCAPS_VIDEOPORT is used for secondary panel support. DDSCAPS_VIDEOPORT is enabled; however, a real videoport feature is not implemented. This causes all CTK test cases involving videoport to fail. The failure 1240, 1250, 1340, 1350 occurs because hardware doesn't support 8x8 overlay surface size, the minimum overlay surface hardware can fully support is 16x16.

10.5.4.3 Running the PlayWnd tests

The command line for starting playback of a WMV test video clip in PlayWnd is:

```
playwnd [wmv test file]
```

For example, `playwnd motocross_208x160_30fps.wmv`

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.

10.5.4.4 Running the Multiple Overlay Custom Test

In the CE target shell window, execute the following command to start the MultipleOverlay application:

```
s MultipleOverlay.exe
```

The correct operation of this application is to create several mosquito images, which float around the main display screen area. The topmost mosquito shows no bordering area, while the other is contained in a black box (this is due to source color keying only working for the topmost overlay surface).

Press the ESC key on the keypad to end the application.

10.5.4.5 Running the Video De-Interlacing CTK Custom Test

Before executing the VDI CTK test, the VDI test input file, `stefan_interlaced_320x240_30frames.yv12`, must be copied from the VDI test directory into the $(_FLATRELEASEDIR) directory.

The command line for running the VDI test is

```
tux -o -d vdi_test
```

The VDI test does not require any test specific command line options.

Once executed, a sequence of 30 video frames are shown on the display panel. Each frame is shown for approximately one second. There should be no artifacts, and each image should be of good image quality. If successful, the test completes and the debug output shows that it has passed.

10.6 Display Driver API Reference

For information about the display driver APIs, see the CE Help. The only display driver feature that requires a customized API is dual display support, where a custom API is required to access the secondary primary surface.

10.6.1 GDI and DirectDraw APIs

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

Windows Embedded Compact 7 > Device Drivers > Display Drivers > Display Driver Reference

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

Windows Embedded Compact 7 > Audio, Graphics and Media > DirectDraw

10.6.2 Driver Escape Code Extensions

Driver escape codes may be added and used by the display driver to provide access to display driver functionality beyond what is provided through GDI and DirectDraw. The display driver achieves this by defining driver escape codes, along with any structures needed to pass parameters to the display driver. These driver escape code extensions are detailed in the following sections.

10.6.2.1 DISPLAY_SET_DISPLAY_FREQUENCY Escape Code

The DISPLAY_SET_DISPLAY_FREQUENCY escape code must be used with the **ExtEscape**() driver escape function in order to set the display frequency in the display driver. The display frequency should be set before calling ChangeDisplaySettingsEx to set the display mode. The combination of the resolution parameter from the ChangeDisplaySettingsEx and the frequency set through DISPLAY_SET_DISPLAY_FREQUENCY allows the display driver to choose the correct display mode to enable. In the case where multiple display modes share the same resolution but different frequencies, this function must be used to help select the correct display mode.

The parameters listed below are for the **ExtEscape**() function and must be set in order to enable or disable interlaced video mode.

Parameters

cbInput	A pointer to a DWORD containing the desired display frequency, in Hz
lpszInData	Must be equal to the size of a DWORD or the function call fails

10.6.2.2 DISPLAY_GET_DISPLAY_FREQUENCY Escape Code

The DISPLAY_GET_DISPLAY_FREQUENCY escape code must be used with the **ExtEscape**() driver escape function in order to retrieve the display frequency in the display driver.

The parameters listed below are for the **ExtEscape**() function and must be set in order to enable or disable interlaced video mode.

Parameters

cbOutput	A pointer to a DWORD that holds the current display frequency, in Hz
lpszOutData	Must be equal to the size of a DWORD or the function call fails

10.6.2.3 DISPLAY_IS_VIDEO_INTERLACED Escape Code

The DISPLAY_IS_VIDEO_INTERLACED escape code must be used with the **ExtEscape**() driver escape function in order to enable or disable interlaced video mode in the display driver. Interlaced video mode ensures that the display driver treats incoming video overlay surfaces as interlaced video frames. After calling this function to enable interlaced video mode, all subsequent video frames undergo video de-interlacing to convert those frames into progressive frames before being displayed, until this function is called again to disable the mode.

The parameters listed below are for the **ExtEscape**() function and must be set in order to enable or disable interlaced video mode.

Parameters

cbInput	Pointer to an InterlacedVideoData structure, containing information about whether to enable or disable interlaced video mode and which field is the top field
lpszInData	Must be equal to the size of the InterlacedVideoData structure or the function call fails

10.6.2.4 DISPLAY_SETCRRECT Escape Code

The DISPLAY_SETCRRECT escape code must be used with the **ExtEscape**() driver escape function in order to setup the conditional read area in the display driver. Conditional read is a feature for reducing bus bandwidth through mask some primary surface data. Application can set one or more rectangles in which data will be masked before sending to the screen. These rectangles can be overlapped or not. This feature will not impact overlay surface data transfering.

The parameters listed below are for the **ExtEscape**() function and must be setup for configuring conditional read feature.

Parameters

cbInput	Pointer to an SetCRRectData structure, containing information about the position and size of an rectangle in which the data will be drawn or not drawn.
lpszInData	Must be equal to the size of the SetCRRectData structure or the function call fails

10.6.3 Dual Display API

Microsoft does not provide native support for dual independent displays in Windows Embedded Compact 7. The Windows Embedded Compact 7 documentation describes support for Multiple Screens and for a Secondary Display Driver, but both of these features are critically limited and insufficient to provide dual independent display support. As a result, a custom extension to the DirectDraw APIs is required to allow an application to access a secondary primary surface for the secondary display.

10.6.3.1 Dual Display Interface

When the display is configured to support dual display, a secondary display primary surface is only created after the secondary display device is enabled, and is deleted once the secondary display device is disabled. A custom flag has been created to allow applications to access this primary surface. This flag, DDSCAPS_PRIMARYSURFACE2, must be used when calling the DirectDraw `CreateSurface()` function to create a handle to the secondary primary surface. Once the application has a handle to the DirectDraw surface for the secondary primary surface, the DirectDraw `Blt()` function may be used to render into the surface and onto the secondary display. The secondary display can also support one overlay surface, which must be created based on the secondary primary surface.

The follow code fragment shows how an application might draw to the secondary display:

```
#define DDSCAPS_PRIMARYSURFACE2 (DDSCAPS_PRIMARYSURFACE|DDSCAPS_VIDEOPORT)
// Create DirectDraw object
hRet = DirectDrawCreate(NULL, &g_pDD, NULL);
if (hRet != DD_OK)
        return InitFail(hWnd, hRet, TEXT("DirectDrawCreate FAILED"));

// Set Level to DDSCL_NORMAL, or else main display primary may be wiped out!
hRet = g_pDD->SetCooperativeLevel(hWnd, DDSCL_NORMAL);
if (hRet != DD_OK)
        return InitFail(hWnd, hRet, TEXT("SetCooperativeLevel FAILED"));

// Get a pointer to the secondary display primary surface
memset(&ddsd, 0, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.dwFlags = DDSD_CAPS;
ddsd.ddsCaps.dwCaps = DDSCAPS_PRIMARYSURFACE2;
hRet = g_pDD->CreateSurface(&ddsd, &g_pDDSPrimary, NULL);
if (hRet != DD_OK)
        return InitFail(hWnd, hRet, TEXT("CreateSurface FAILED"));

// Create back buffer with size equal to primary to blit from
memset(&ddsd, 0, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.ddsCaps.dwCaps = DDSCAPS_SYSTEMMEMORY;
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH | DDSD_PIXELFORMAT;
ddsd.dwWidth = 480;
ddsd.dwHeight = 640;
```

```
ddsd.ddpfPixelFormat = ddpfOverlayFormats[0];
hRet = g_pDD->CreateSurface(&ddsd, &g_pDDSBack[0], NULL);

// Load image onto backbuffer
LoadImageOntoSurface(g_pDDSBack[0], szImg1);

// Blt from back buffer onto secondary primary surface
g_pDDSPrimary->Blt(&rd, g_pDDSBack[i++], &rs, NULL, NULL);
```

10.6.3.2 Dual Display API Limitations

The dual display API limitations are as follows:

- The Windows manager has no awareness of the secondary display primary surface and cannot draws windows, menus, buttons, and other objects to the secondary display. Therefore, a custom application must handle all drawing to the secondary display, using the interface described in this section.
- Due to incompatibilities between DirectDraw middleware and the customized secondary display primary surface the `Flip()` function cannot be used with the secondary primary surface.
- The custom flag created to allow access to the secondary display primary surface reuses the DirectDraw DDSCAPS_VIDEOPORT flag. As a result, attempts to create and use video ports result in failures. Additionally, the DirectDraw Verfication CTK tests related to video ports return as FAILED (these tests previously returned SKIPPED).
- There is no touch support for the secondary display device when in dual display mode.
- Due to system bus bandwidth limitation, some display features are limited when two display devices are on.

Chapter 11
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.

11.1 DVFC Driver Summary

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

Table 11-1. DVFC Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\DVFC
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\DVFC
Driver DLL	dvfc.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > DVFC driver support using the on-board PMIC
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOPMIC = BSP_DVFC = 1

11.2 Supported Functionality

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

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

4. Supports peripheral setpoint requests initiated by CSPDDK clock management code.

11.2.1 i.MX53 ARD Supported Functionality

1. Supports DVFS for CPU (GP) and peripheral (LP) power domains
2. Exposes separate Power Manager stream interfaces for CPU and peripheral domains to provide individual control of setpoint for each domains
3. Supports reactive CPU load tracking to control setpoint based on system performance requirements. Current release uses software load tracking algorithm.
4. Provides voltage control using LTC3598 PMIC

11.3 Hardware Operation

This section describes about the DVFC hardware operation.

11.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

11.3.2 i.MX53 ARD Configuration

The DVFC driver is dependent upon the LTC3598 PMIC interface for dynamic voltage control through the I2C2 port. The LTC3598 SDK import library is used by the DVFC driver to access the PMIC interface.

11.4 Software Operation

This section describes about the registry settings.

11.4.1 i.MX53 ARD Registry Settings

The following registry keys are required to properly load the i.MX53 ARD DVFC module.

```
IF BSP_NOPMIC !
IF BSP_DVFC
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\DVFC1]
   "Prefix" = "DVF"
   "Index" = dword:1
   "Dll"="dvfc.dll"
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"    ; PMCLASS_GENERIC_DEVICE

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\DVFC2]
   "Prefix" = "DVF"
   "Index" = dword:2
   "Dll"="dvfc.dll"
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"    ; PMCLASS_GENERIC_DEVICE
```

```
[HKEY_LOCAL_MACHINE\SYSTEM\CEDDK]
    "CalibrateStallCounter"=dword:0
ENDIF DVFC
ENDIF BSP_NOPMIC !
```

11.4.2 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.

11.4.3 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.

11.4.3.1 i.MX53 ARD Voltage/Frequency Setpoints

The i.MX53 ARD DVFC driver supports mutually exclusive voltage and frequency setpoints for the CPU and peripheral power domains. Table 11-2 and Table 11-3 provide the voltage/frequency characteristics for these setpoints.

Table 11-2. i.MX53 ARD DVFC Setpoints for CPU Domain

Setpoint Name	CPU Frequency (MHz)	Core Voltage
DDK_DVFC_SETPOINT_HIGH	800	1.1 V
DDK_DVFC_SETPOINT_MEDIUM	400	0.95 V
DDK_DVFC_SETPOINT_LOW	166	0.95 V

Table 11-3. i.MX53 ARD DVFC Setpoints for Peripheral Domain

Setpoint Name	AXI/AHB/DDR Frequency (MHz)	Core Voltage
DDK_DVFC_HIGH2	200/133/400	1.300 V
DDK_DVFC_SETPOINT_HIGH	200/66/400	1.300 V
DDK_DVFC_SETPOINT_MEDIUM	166/66/333	1.300 V
DDK_DVFC_SETPOINT_LOW	42/42/333	1.300 V

The setpoint attributes are controlled by the definitions in the platform-specific DVFS header file (found in `\PLATFORM\<Target Platform>\SRC\INC\dvfs.h`). The DVFC driver uses these definitions to populate a global setpoint array (g_SetPointConfig) that is referenced during setpoint transitions.

11.4.3.2 i.MX53 ARD 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. The setpoint requirements for drivers are expressed in terms of the the following parameters:

- Internal AHB bus frequency requirement
- Internal AXI peripheral bus frequency requirement
- Peripheral domain (LP) voltage requirement

These parameters are statically mapped to clock nodes managed by drivers through the CSPDDK. Each time a driver enables module clocks using DDKClockSetGatingMode, the CSPDDK maps the voltage/frequency requirements for the specified clock node to a supported peripheral domain setpoint that meets those requirements. The static mapping can be changed by modifying the **periphSetpointReq** elements of the globally shared **DDK_CLK_CONFIG** data structure. This mapping occurs in `\PLATFORM\<Target Platform>\SRC\COMMON\BSPCMN\bspargs.c`.

WARNING

Do not map a peripheral to a set of voltage/frequency requirements that violate the electrical specification or do not provide adequate clocking for the peripheral protocol specification.

The DVFC driver advertises support for IOCTL_POWER requests from the 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\dvfc.c`). The DVFC driver exposes two separate stream interfaces to allow individual control of the CPU and peripheral power domain setpoints. Stream DVF1 is mapped to the CPU domain and DVF2 is mapped to the peripheral domain. To change the setpoint mapping for a specific device power state (D0–D4), modify the code in **DvfcMapDevPwrStateToSetpoint**.

11.4.4 DDK Interface

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

- DDKClockSetpointRequest, Section 9.6.1.2.12, “DDKClockSetpointRequest.”
- DDKClockSetpointRelease, Section 9.6.1.2.13, “DDKClockSetpointRelease.”

11.4.5 Power Management

The DVFC is an integral part of the power management supported by the BSP. However, as 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.

11.4.5.1 PowerUp

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

11.4.5.2 PowerDown

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

11.4.5.3 IOCTL_POWER_CAPABILITIES

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

11.4.5.4 IOCTL_POWER_SET

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

11.4.5.5 IOCTL_POWER_GET

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

11.5 Unit Test

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

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

   NOTE

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

   NOTE

   Due to the hardware dependency, USBH1, SATA, and TPS drivers need to be removed to achieve the lowest power consumption.

   If BSP_DISPLAY_Z160 is set in the workspace, the VCC power will remain high after resuming from suspend due to unknown reason. Removing this variable from the workspace could temporarily workaround this issue.

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

4. Launch the stress test application. From the CE shell, the stress test can be launched with the following command line:

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

5. Debug messages to indicate setpoint transitions can be enabled using the DVFC_VERBOSE macro found in `\PLATFORM\<Target Platform>\SRC\DRIVERS\DVFC\COMMON\dvfc.`

Chapter 12
Enhanced Configurable Serial Peripheral Interface (eCSPI) Driver

The Enhanced Configurable Serial Peripheral Interface (eCSPI) module provides master functionality of a Enhanced CSPI bus. The eCSPI module includes larger receive and transmit buffers than the CSPI and also includes more flexible tail data operations.

12.1 eCSPI Driver Summary

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

Table 12-1. eCSPI Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\ECSPI
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ECSPI
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\ECSPI
SDK Library	ecspisdk.lib
Driver DLL	ecspi.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>:ARMV7 > Device Drivers > CSPI Bus
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOCSPI =

12.2 Supported Functionality

The eCSPI driver supports the following features:

1. eCSPI master mode of operation
2. eCSPI configurable bus feature
3. eCSPI multiple channel method
4. DMA exchange mode
5. Configurable access method of interrupt mode and DMA mode
6. Buffering exchange for asychronous SPI access
7. Stream interface

8. Two power management modes, full on and full off

12.2.1 Conflicts with Other Peripherals and Catalog Items

12.2.1.1 Conflicts with SoC Peripherals

The i.MX53 eCSPI1 conflicts with EIM module.

12.2.1.2 Conflicts with Board Peripherals

The i.MX53 ARD eCSPI1 conflicts with LAN9220 module.

12.3 Software Operation

12.3.1 Registry Settings

12.3.1.1 i.MX53 Registry Settings

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

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ECSPI1]
   "Prefix"="SPI"
   "Dll"="ecspi.dll"
   "Index"=dword:1
   "Order"=dword:1
   "IClass"=multi_sz:"{A32942B7-920C-486b-B0E6-92A702A99B35}"
```

12.3.2 Communicating with the eCSPI

The eCSPI is a stream interface driver, and is thus accessed through the file system APIs. To communicate using the eCSPI, 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 in the following sections.

12.3.3 Creating a Handle to the eCSPI

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

To open a handle to the eCSPI:

1. Insert a colon after the eCSPI port for the first parameter, *lpFileName*.
   — For example, specify `SPI1:` as the eCSPI port.

2. Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to an eCSPI 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 eCSPI 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)
```

12.3.4 Data Transfer Operations

The eCSPI driver provides one command, CSPIExchange, that facilitates performing both reads and writes through the eCSPI bus. The basic unit of data transfer in the eCSPI 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 the process of performing write and read operations through the eCSPI bus.

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

To perform an eCSPI 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 CHANNEL SELECT, DATA RATE, BURST LENGTH, SSPOL, POL, DRCTL, and specify the method parameters for using or not using the DMA. The BURST LENGTH unit is bit.
   b) Set the *pTxBuf* field to the user buffer with the transmit data.
   c) Set the *pRxBuf* field to the user buffer which receives data. If there is no receive data, set the field to NULL.
   d) Set the *xchCnt* field. The *xchCnt* unit is 32-bit. *xchCnt* must equal BurstLength/32 or BurstLength/32 +1 (if BurstLength is not multiple of 32-bits).
   e) Specify the *xchEvent* parameter and the *xchEventlength* including the tail zero character. Otherwise, set *xchEvent* to NULL and *xchEventlength* to 0. When using *xchEvent*, the XCH data is queued inside the driver.
2. Set the *hDevice* parameter to the previously acquired eCSPI port handle.
3. Set *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*64,   //Burstlength 64 DWORDS
        FALSE, // SSPOL: Active LOW
        FALSE, // POL: Active high polarity
        0,     // DRCTL: Don't care SPI_RDY
        FALSE}; //Don't use DMA

DWORD TxData[1024];
DWORD RxData[1024];

CSPI_XCH_PKT_T xchPkt =
{       &buscnfg,
        TxData,
        RxData,
        64,                                          // DWORD, Equal Burstlength/32
        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 eCSPI
CSPIExchange(hCSPI, &xchPkt);
// optional
WaitForSingleObject(hEvent, INFINITE);
// Code for dealing received DATA
```

12.3.5 Closing the Handle to the eCSPI

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

12.3.6 Power Management

The primary method for limiting power consumption in the eCSPI module is to gate off the input clock to the module when the input eCSPI clock is not needed. This is accomplished through the **DDKClockSetGatingMode** function call. In the all Windows Compact 7 BSP use cases, the eCSPI controller acts as a master device. As a result, the eCSPI clock can be turned off, whenever the module is not processing eCSPI packets.

12.3.6.1 PowerUp

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

12.3.6.2 PowerDown

This function is not implemented for the eCSPI driver.

12.3.6.3 IOCTL_POWER_SET

This function is implemented for the eCSPI driver. When D4 power mode is set, the driver enters into D4 mode after finishing the last running burst transmission. When the driver leaves D4 power mode, it recovers its original operating mode.

12.4 Unit Test

The eCSPI is used for PMIC or SPIFlash, the following methods are used to test it:

- Loopback test
- Access SPI flash on board through the eCSPI port

A eCSPI sample test application may be found at directory: `\WINCE700\SUPPORT\TEST\ECSPI`

12.5 eCSPI Driver API Reference

12.5.1 eCSPI Driver IOCTLS

This section consists of descriptions for the eCSPI I/O control codes (IOCTLs). These IOCTLs are used in calls to **DeviceIoControl** to issue commands to the eCSPI device. Descriptions are provided only for relevant parameters of the IOCTL.

12.5.1.1 CSPI_IOCTL_EXCHANGE

This **DeviceIoControl** request performs the transfer of data to a target device. An SPI_XCH_PKT object is required, which contains the eCSPI 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

12.5.1.2 CSPI_IOCTL_ENABLE_LOOPBACK

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

12.5.1.3 CSPI_IOCTL_DISABLE_LOOPBACK

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

12.5.2 eCSPI Driver SDK Wrapper

12.5.2.1 CSPIOpenHandle

This function retrieves the eCSPI device handle.

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

Parameters

lpDevName — eCSPI device name for retrieving handle from CreateFile()

Return Values — Returns handle for eCSPI driver, returns INVALID_HANDLE_VALUE if failure

12.5.2.2 CSPICloseHandle

This function closes a handle of the eCSPI stream driver.

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

Parameters

hDev — eCSPI device handle retrieved from CreateFile()

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

12.5.2.3 CSPIEnableLoopback

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

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

Parameters

hDev — eCSPI device handle retrieved from CreateFile()

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

12.5.2.4 CSPIEnableLoopback

This function sets the eCSPI controller work in loopback mode. So that inspect if data value during XCH is right.

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

Parameters

hDev — The eCSPI device handle retrieved from CreateFile().

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

12.5.2.5 CSPIExchange

This function performs XCH operations.

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

Parameters

hDev	eCSPI 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

12.5.3 eCSPI Driver Structures

12.5.3.1 CSPI_BUSCONFIG_T

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

```
// eCSPI bus configuration
typedef struct
{
    UINT8         ChannelSelect;      //CS0, CS1, CS2, CS3
    UINT32        Freq;
    UINT32        BurstLength;        //bitcount, recommend 32bit as unit.
    BOOL          SSPOL;
    BOOL          SCLKPOL;
    BOOL          SCLKPHA;
    UINT8         DRCTL;
    BOOL          usedma;
} CSPI_BUSCONFIG_T, *PCSPI_BUSCONFIG_T;
```

Table 12-2. CSPI_BUSCONFIG_T Structure Members

Member	Description
ChannelSelect	Select XCH channel, range 0–3
Freq	Data bandrate
BurstLength	Define bits used in a single XCH, range 1–32×64
SSPOL	SPI SS Polarity Select FALSE: active low TRUE: active high
SCLKPOL	SPI Clock Polarity Control FALSE: active high polarity (0 = Idle) TRUE: active low polarity (1 = Idle)
SCLKPHA	SPI Clock/Data Phase Control FALSE: phase 0 operation TRUE: phase 1 operation

Table 12-2. CSPI_BUSCONFIG_T Structure Members (continued)

Member	Description
DRCTL	DRCTL of eCSPI XCH operation 00: Do not care SPI_RDY 01: Burst triggered by failing edge of SPI_RDY 10: Burst triggered by low level of SPI_RDY 11: Reserved
usedma	True: use DMA mode

12.5.3.2 CSPI_XCH_PKT_T

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

```
// eCSPI 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 12-3. CSPI_XCH_PKT_T Structure Members

Member	Description
pBusCnfg	Pointer to eCSPI bus configuration object
pTxBuf	Pointer to Tx data buffer
pRxBuf	Pointer to Rx data buffer
xchCnt	Amount of XCH operation to SPI device. xchCnt is 32-bit unit and must equal BurstLength/32 or BurstLength/32 +1 (if BurstLength is not multiple of 32-bit)
xchEvent	Asychronous access using the internal exchange queue
xchEventLength	Event name length including tailing Zero

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	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\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_v3.lib, esdhcbase_<*Target SOC*>.lib, sdcardlib.lib, sdhclib.lib, sdbus.lib
Catalog Item	Third Party > BSP > Freescale i.MX53 ARD: ARMV7 > Device Drivers > SD Host Controller > Enhanced SD Host Controller 1 (ESDHC1) Support Third Party > BSP > Freescale i.MX53 ARD: ARMV7 > Device Drivers > SD Host Controller > Enhanced SD Host Controller 2 (ESDHC2) Support
SYSGEN Dependency	SYSGEN_SD_MEMORY=1
BSP Environment Variables	BSP_NOSDHC=1 BSP_ESDHC1=1 BSP_ESDHC2=1

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. Fast Path
3. DMA or PIO modes of data transfers based on value of eSDHC driver registry key, DisableDMA
4. SD, SD High Capacity (up to spec v2.1), MMC (up to spec v4.3), and SDIO cards (up to spec v2.0).

5. One host supports only one card connected to it
6. DLL supports multiple instances of the eSDHC controller
7. Configuration of the block sizes from 1–4096 bytes in single and multi-block modes
8. Insertion and removal of card, even when system is suspended; when the system resumes, the card (if present) is remounted
9. Power states on(D0) and off (D4), D1–D3 states are treated as D4
10. Clocks are gated in D4 state, and ungated in D0 state, except for SDIO cards for which clocks are never gated. Clocks are never gated if BSP_CLK_GATING_BETWEEN_CMDS_SDHC macro is FALSE or undefined in the bsp_cfg.h file
11. Write protect switch on SD cards
12. Combo cards (for example, SD memory + WiFi functionality on same card)
13. MMC cards in 1-bit/4-bit mode and SD/SDIO cards in 4-bit modes

13.3 Hardware Operation

Refer to the chapter on the eSDHC in the *i.MX53 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

13.3.1.2 Conflicts with Other Board Peripherals

13.4 Software Operation

The eSDHC driver follows the Microsoft-recommended architecture (standard host controller driver) for Secure Digital Host Controller drivers, whenever possible.

13.4.1 Required Catalog Items

13.4.1.1 SD and MMC Memory Card Support

Core OS > Windows Embedded Compact > Device Drivers > SD > SD Clients > SD Memory

13.4.1.2 SDIO Wifi Card Support

Third Party > BSP > Freescale i.MX53 SMD: ARMV7 > Device Drivers > WiFi > Atheros AR6102 (SDIO) Driver

13.4.2 eSDHC Registry Settings

13.4.2.1 i.MX53 SDHC Registry Settings

```
; @CESYSGEN IF CE_MODULES_SDBUS

IF BSP_ESDHC1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ESDHC1]
    "Order"=dword:21
    "Dll"="esdhc.dll"
    "Prefix"="SHC"
    "Index"=dword:1
    ;"DisableDMA"=dword:1                                    ; Use this reg setting to disable
both internal and external DMA
    "MaximumClockFrequency"=dword:3197500           ; 52 MHz max clock speed
    ;"WakeupSource"=dword:1                                   ; this will enable system wakeup
when card is inserted or removed during suspend state
ENDIF BSP_ESDHC1

IF BSP_ESDHC2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ESDHC2]
    "Order"=dword:21
    "Dll"="esdhc.dll"
    "Prefix"="SHC"
    "Index"=dword:2
    ;"DisableDMA"=dword:1                                    ; Use this reg setting to disable
both internal and external DMA
    "MaximumClockFrequency"=dword:3197500           ; 52 MHz max clock speed
    ;"WakeupSource"=dword:1                                   ; this will enable system wakeup
when card is inserted or removed during suspend state
ENDIF BSP_ESDHC2

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

; @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.MX53 DMA Support

For the i.MX53, both DMA and non-DMA mode are supported by the driver. DMA mode is used by default, and user can change the *DisableDMA* value in registry file `esdhc_mx53.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 (ADMA). The driver supports only ADMA mode.

For ADMA1, the upper layer should ensure that the start of the buffer is a page aligned address (4096 bytes). Due to cache coherency issues arising from the processor and DMA access of the memory, the above criteria is further restricted for the read or receive operation (it is not applicable for write or transmit) such that the number of bytes to transfer in the last buffer should be cache line size (64 bytes) aligned.

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

ADMA2 removes these restrictions, so all types of buffer addresses and sizes can be supported. However, cache line alignment for address of the starting buffer and the length of the last buffer are required.

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.MX53 Power Management

The eSDHC driver conserves power by making calls to the clock management hardware, CCM, to gate clocks to eSDHC module in between commands sent to the card. Clocks are also turned off when there is no card present in the socket. Clock gating can be turned off by setting the BSP_CLK_GATING_BETWEEN_CMDS_SDHC macro to FALSE or if it is undefined in `bsp_cfg.h` file. Clocks cannot be gated at the CCM when a SDIO card is plugged in because the eSDHC is unable to wake up upon interrupt from the SDIO card. In this case, clocks are only gated at the eSDHC, not at the CCM.

The power supply for the SD ports is shared with other peripherals. The PMIC driver determines when it is safe to grant the eSDHC driver request to turn off the supply. The eSDHC driver requests to turn off this supply whenever there is no card plugged in or when the system is suspended. When a card is inserted or the system resumes from suspend, the eSDHC driver requests to turn on this supply. The SHC_PowerDown and SHC_PowerUp APIs calls down to the BspESDHCSetSlotVoltage function, which actually handles the communication with the PMIC driver.

13.5 Unit Test

The eSDHC driver is tested using the following tests included as part of the Windows Embedded Compact 7 Test Kit (CTK).

- File System Driver Test
- File System Performance Test
- Storage Device Block Driver Read/Write Test
- Storage Device Block Driver API Test
- Storage Device Block Driver Performance Test
- SD Bus FunctionalTest

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
sdmemtux.dll	Library that contains the sdbus test cases
sdtst.dll	Test client driver
fsdtst.dll	File System Driver Test .dll file
fsperflog.dll	File system performance test library
ceperf.dll	Module containing functions that monitor and log performance
perfscenario.dll	Module containing functions that monitor and log performance
btsperflog.dll	Generate ceperfformatted output
perflog.dll	Logging library that provides functionality for timing and logging the performance data generated by the test dll.

Table 13-3. Software Requirements

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

13.5.3 Building the Unit Tests

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

```
[Drive]:\Program Files\WindowsEmbeddedCompact7TestKit\tests\target\armv7
```

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 **CTK Tests** in the Platform Builder Help, or view the online documentation at the following URL:
http://msdn.microsoft.com/en-us/library/gg154684.aspx

13.5.4.1 File System Driver Test

Use command line

```
tux –o –d fsdtst –c "-p SDMemory –zorch"
```

to run the tests on an SDSC card.

For SDHC card, use

```
tux –o –d fsdtst –c "-p SDHCMemory –zorch"
```

For MMC cards, use

```
tux –o –d fsdtst –c "-p MMC –zorch"
```

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:

For SDSC:

```
tux –o –d rwtest –c "-profile SDMemory -zorch"
```

For SDHC:

```
tux –o –d rwtest –c "-profile SDHCMemory -zorch"
```

For MMC:

```
tux –o –d rwtest –c "-profile MMC -zorch"
```

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

For SDSC:

```
tux –o –d disktest –c "-profile SDMemory -zorch"
```

For SDHC:

```
tux –o –d disktest –c "-profile SDHCMemory -zorch"
```

For MMC:

```
tux –o –d disktest –c "-profile MMC -zorch"
```

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

For SDSC:

```
tux –o –d disktest_perf –c "-profile SDMemory -zorch"
```

For SDHC:

```
tux –o –d disktest_perf –c "-profile SDHCMemory -zorch"
```

For MMC:

```
tux –o –d disktest_perf –c "-profile MMC -zorch"
```

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

13.5.4.5 SD Bus Level Funcational Test

Use command line

For SDSC:

```
tux –o –d sdmemtux.dll –c "/device sd"
```

For SDHC:

```
tux –o –d sdmemtux.dll –c "/device sdhc"
```

For MMC:

```
tux –o –d sdmemtux.dll –c "/device mmc" -x !1011 -x !1012
```

to run the tests.

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://msdn.microsoft.com/en-us/library/gg157412.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 eight 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	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\ESAI ..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\ESAI2
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ESAI
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\ESAI ..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\ESAI2
Driver DLL	esai_cs42888.dll
SDK Library	esai_common_fsl_v3.lib, esai2_common_fsl_v3.lib, esai_<*Target SOC*>.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > ESAI Support Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > ESAI stereo + 5.1 Channel output Support
SYSGEN Dependency	SYSGEN_AUDIO
BSP Environment Variables	BSP_NOAUDIO= BSP_ESAI=1 for ESAI support BSP_ESAI2=1 for ESAI stereo + 5.1 Channel output Suppor BSP_ASRC=1

14.2 Supported Functionality

The ESAI audio driver enables the ARD System to provide the following software and hardware support:

1. Conforms to the audio driver architecture as defined for Windows Embedded Compact 7 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.
6. Supports 1–8 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 CS42888 multi-channel audio Codec
9. Supports playback without ASRC support at sample rate 48KHz (according to the external oscillator).
10. Supports playback with ASRC support from sample rate 32K–192K
11. Supports record function with Freescale hardware platform that includes the CS42888 Codec
12. Supports record sample rate 48KHz (according to the external oscillator)

14.3 Hardware Operation

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

14.3.1 Conflicts with Other Peripherals and Catalog Items

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 in sequence, such as TX0, TX1, TX2, TX3, TX0, TX1, TX2, TX3, 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 four transmitters are used for 1–8 channels of audio playback:

Channel Number = 8: CH0,CH4->TX0, CH1,CH5->TX1, CH2,CH6->TX2, CH3,CH7->TX3

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 Sample Rate

The ESAI module only supports 48KHz recording. Therefore, playback and record can only be performed under 48kHz sample rate at 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>:ARMV7 > Device Drivers >I2C Bus > I2C Bus2

14.4.2 Scenario Settings

Software implements two scenarios for different use cases.

One is generic ESAI multi-channel output and input case. ESAI will enable the relative output and input channels according to the channel number of original bit stream, and it doesn't support audio mixer. This case is the default setting of ESAI driver, which can be enabled by selecting the calatlog item of 'ESAI Support'. And all the description in this chapter is referring to this case.

The other is two zone output and input case, which can be enabled by selecting the calatlog item of 'ESAI stereo + 5.1 Channel output Support'. In this case, ESAI supports two zones output including one stereo and one 5.1 channel, and supports two zones input including two individual stereo. Application can use wFormatTag member in WAVEFORMATEX structure to designate different zones while opening ESAI device, WAVE_FORMAT_PCM for front zone and WAVE_FORMAT_EXTENSIBLE for back zone. In this scenario, software is involved for data position arrangement in DMA buffer for different channel, sample rate conversion and audio mixer. ASRC hardware is not used.

14.4.3 ESAI Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ESAI]
"Prefix"="WAV"
    "Dll"="esai_cs42888.dll"
    "Index"=dword:3
    "Order"=dword:8
    "Priority256"=dword:95
    "AudioProtocol"=dword:2
          "SupportWAVEFORMATEX"=dword:1
    "IClass"=multi_sz:"{A32942B7-920C-486b-B0E6-92A702A99B35}",
                      "{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 8-channel wave format is supported. To keep the mapping relationship between slot number and transmitters, the audio data needs to be packed to 8-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.

The SupportWAVEFORMATEX key is used to determine whether ESAI driver supports WAVEFORMATEX data structure. When the registry key is set to 1, this format structure is supported.

14.4.4 Supported Wave Data Format

In general, to access the ESAI audio interface, the WAVEFORMATEXTENSIBLE data structure is 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 8, but when AudioProtocol is set to 2 in the registry file, only even number can be used (such as 2, 4, 6,8). For 24-bit audio data packed in bits 23–0 of the 32 bits, set Samples.wValidBitsPerSample to 24 and Format.wBitsPerSample to 32. The SubFormat member is not used since only PCM data is supported.

14.4.5 DMA Support

14.4.5.1 DMA Support

Double-buffer is used for audio data transfer.

14.4.6 Power Management

The primary method for limiting power consumption in ESAI driver is to stop all active audio DMA operations, and to disable all audio hardware components at the end of each audio stream.

14.4.6.1 PowerUp

This function resumes an audio I/O operation that was previously terminated by calling the PowerDown() API. It begins by 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 functionality is currently handled by IOCTL_POWER_SET and the function is just a stub.

14.4.6.2 PowerDown

This function suspends all currently active audio I/O operations just before the entire system enters the low power state. It stops all active audio DMA operations and to disalbe 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.

14.4.6.3 IOCTL_POWER_CAPABILITIES

This function return the power capability of the ESAI driver that it support D0 and D4 status.

14.4.6.4 IOCTL_POWER_GET

This function is not implemented for the ESAI driver.

14.4.6.5 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.

14.5 Unit Test

If the SupportWAVEFORMATEX key is set to 0, 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: `\WINCE700\SUPPORT\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: `\WINCE700\SUPPORT\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
8. Change to the D_Player Tests directory: `\WINCE700\SUPPORT\TEST\ESAI\D_PLAYER`
9. Enter **set WINCEREL=1** on the command prompt and press return.
   This copies the .exe file to the flat release directory
10. Enter **build -c** at the prompt and press return

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

14.5.2 Hardware Setup

N/A

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 [-n channel_number] [-m channel_mask] [-ne]`

Example: `m_player temp\source.wav -n 6 -m 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 data file.

To run the application within VS2008, 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.

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.

To run the application within VS2008, 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.

14.5.3.3 Full Duplex Function Test

The d_player application is used for full duplex function test. The sound from the audio input line is recorded in memory temporarily and meanwhile the memory audio data is output through earphone or speakers.

Usage: `d_player [-n chn_num][-s sample_rate][-b bit_depth][-t seconds]`

Example: `d_player -n 4 -s 48000 -b 16 -t 60`

In this example, the full-duplex works over ESAI for 60 seconds, the wave format is 4 channels, 16-bit bitdepth, with sample rate at 48000 HZ for both playback and record.

To run the application within VS2008, 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.

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.

In general, ESAI driver needs wave data that conforms with the WAVEFORMATEXTENSIBLE structure for operation. Also ESAI driver doesn’t support audio mixer. Hence audio driver CTK is not appropriate for ESAI driver testing.

Chapter 15
Global Positioning System (GPS) Driver

The Global Positioning System (GPS) enables a GPS receiver to determine its location, speed/direction, and time.

15.1 GPS Driver Summary

This platform supports the Atheros AR1520A Single Chip Assisted-GPS (*A-GPS*) solution. AR1520A™ 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. AR1520A delivers high sensitivity, 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.

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

Table 15-1. GPS Driver Summary

Driver Attribute	Definition
Target Platform	
Target SOC	N/A
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\GPS
Driver DLL	athrgpsctrl.dll, OrionSvc.dll, OrionSys.dll, OrionVSP.dll
SDK Library	N\A
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*>: ARMV7 > Device Drivers > GPS
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_GPS=1 BSP_SERIAL_UART2 =1

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 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 SMD platform. Resetting and power on/power off to the GPS module are controlled by the 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.

The following table shows the architecture of GPS driver. Three layers in the GPS software system.

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

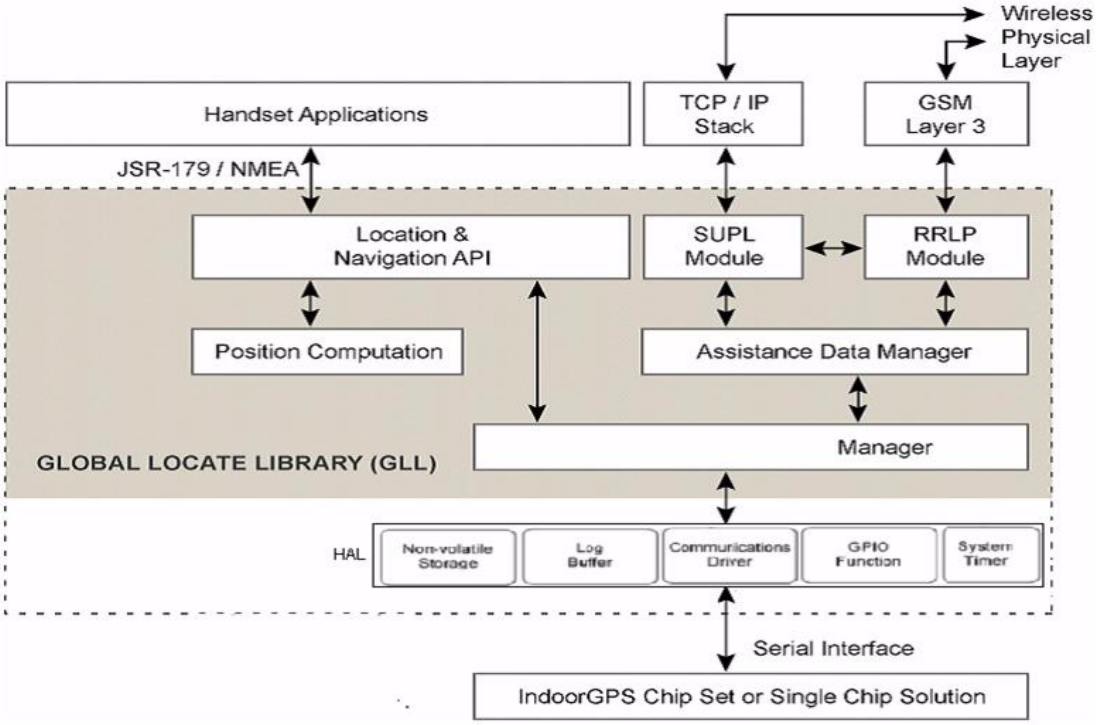


Figure 15-1. Software Architecture of GPS Driver

15.1.1 Application Layer

Handset applications, TCP/IP stack and the GSM layer3 belong to the application layer. Handset applications, such as VisualGpsce.exe, MobileOA.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.

15.1.2 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.

15.1.3 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 athrgpsctrl.dll, and source code can be found at:

PLATFORM\<Target Platform>\SRC\DRIVERS\GPS\GPSCTROL.

15.2 Supported Functionality

The GPS driver provides the following software and hardware support:

1. Integrates the AR1520A GPS module from Atheros company
2. Supports power management mode full on/full off

15.3 Hardware Operation

The GPS driver exchanges data and command between GPS application layer and hardware module via UART2 port

15.3.1 Conflicts with Other Peripherals and Catalog Items

15.3.1.1 Conflicts with SoC Peripherals

GPS uses UART2 port as communication port which confilicts with PATA module

15.3.1.2 Conflicts with Board Peripherals

None.

15.3.2 Hardware Operation

15.3.2.1 UART Port

For i.MX53 SMD board, UART2 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\FILSE\Orion.ini:

"ReceiverComPortName"="COMx:", where “x” should be specified according to the UART actually used ("COM2:").

15.3.2.2 GPIO Control

The GPIO pins are used to control the GPS module as shown in The following table.

Table 15-2. GPIO Control

GPIO Name	PIN	Value Description
GPS_RESET_B	PATA_DATA12	0: Reset of GPS module is asserted 1: Reset of GPS module is de-asserted

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\GPS\GPSCTROL\gpsctrl.cpp

15.4 Software Operation

15.4.1 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.

15.4.2 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.

15.4.3 GPS Driver Registry Settings

15.4.3.1 Configuration Registry Keys

The following configuration is to load GPSCtrol driver.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\GPSReset]
   "Prefix"="AGC"
   "Dll"="athrgpsctrl.dll"
   "Index"=dword:0
   "Order"=dword:20
```

15.5 Unit Test

A navigation application is necessary to test the GPS driver. There are a simple GPS test application located as Control Panel -> Orion which is provided by Atheros. Freescale does not provide a navigation application. The customer may contact Atheros for navigation application and more information.

Chapter 16
Graphics Processing Unit (GPU)

The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D/3D graphics applications. The GPU3D (3D graphics processing unit) is based on the AMD Z430 core, which is an embedded engine capable of DirectX9 Shader Model 3.0+ program execution and accelerates user level graphics APIs such as OpenGL ES 1.1 and 2.0.. 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 setThe GPU driver is delivered only as binary code.

16.1 GPU Driver Summary

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

Table 16-1. GPU Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\\<*Target Platform*>\SRC\DRIVERS\GPU
Driver DLL	amdgslldd.dll essc.dll lib2d-z160.dll lib2dz160k.dll lib2dz430k.dll lib2d-z430.dll libEGL.dll libGLES_CM.dll libGLESv1_CM.dll libGLESv2.dll libgsl.dll ibgslmemcfg.dll libgsluser.dl l libgslWrapperk.dll libkos.dll libOpenVG.dll libos.dll libpanel.dll librenderboy.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <*Target Platform*> > Device Drivers > GPU > Graphics Processing Unit > OpenGL ES support Third Party > BSP > Freescale <*Target Platform*> > Device Drivers > GPU > Graphics Processing Unit > OpenVG support
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_GPU_BASE=1 BSP_GPU_OPENGLES=1 BSP_GPU_OPENVG=1

16.2 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. OpenGL® ES (royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems) 1.1 API defined by Khronos Group
3. OpenGL ES 2.0 API defined by Khronos Group
4. 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
5. D0 (Full On) and D4 (Off) power states

16.3 Hardware Operation

Refer to the GPU chapter in the*i.MX53Applications Processor Reference Manual* for detailed hardware operation and programming information.

16.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

16.4 Software Operation

16.4.1 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
- OpenGL ES 1.1 and 2.0 API

16.4.2 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
  - lib2dz430k.dll—contains Z430 c2d 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
  - libEGL.dll—contains EGL implementation
  - libOpenVG.dll—contains OpenVG 1.1 implementation
  - essc.dll—contains shader compiler logic
  - librenderboy.dll—contains the logic of rendering framework
  - lib2d-z430.dll—contains Z430 c2d helper functions
  - libGLES_CM.dll—contains OpenGL ES 1.1 implementation
  - libGLESv1_CM.dll—contains OpenGL ES 1.1 implementation, different wrapper
  - libGLESv2.dll—contains OpenGL ES 2.0 implementation

16.4.3 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.

16.4.4 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
"IClass"="{8DD679CE-8AB4-43c8-A14A-EA4963FAA715}"    ; PMCLASS_GENERIC_DEVICE

[HKEY_LOCAL_MACHINE\Drivers\GPU]
```

```
;"MemSize"=dword:2000000  ; 32MB
In above Registry setting, key "MemSize" is used to set the gpu memory pool size. Customer can
set it according to different board and 3D/2D use cases requirement.

;"ISTPriorityLevel"=dword:98  ; used for adjust GPU interrupt thread priority level.
;"EnableFlipping"=dword:0  ; non-zero indicates enabled
;"WaitForVB"=dword:0  ; non-zero indicates enabled
```

The GPU driver exports Common 2D (C2D) APIs to accelerate 2D operations. Two C2D implementations, C2D Z430 and C2D Z160, are provided. They use different GPU cores and expose the same APIs.

Currently reference codes are added into the display driver to accelerate following GDI operations using C2D Z430 or C2D Z160. For information, please refer to 10.4.3.3, "Display Driver Blit Acceleration"

16.5 Unit Test

The following sections describe the unit tests for the GPU driver.

16.5.1 Unit Test Hardware

No special requirements.

16.5.2 Unit Test Software

The following sections describe the software for the GPU driver unit tests.

16.5.2.1 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 follow figure.

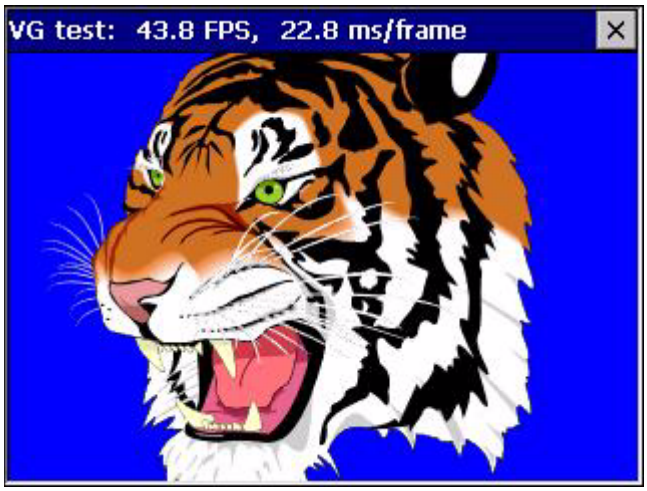


Figure 16-1. Cube Test

16.5.2.2 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.

16.5.2.3 Cube Test

This test application verifies the basic functionality of OpengGL ES 1.1. It is included in the release image and is located under `\Windows\cube.exe`. Click to launch this test and a rotating cube appears on the screen as shown in follow figure. Press ESC to exit this application.

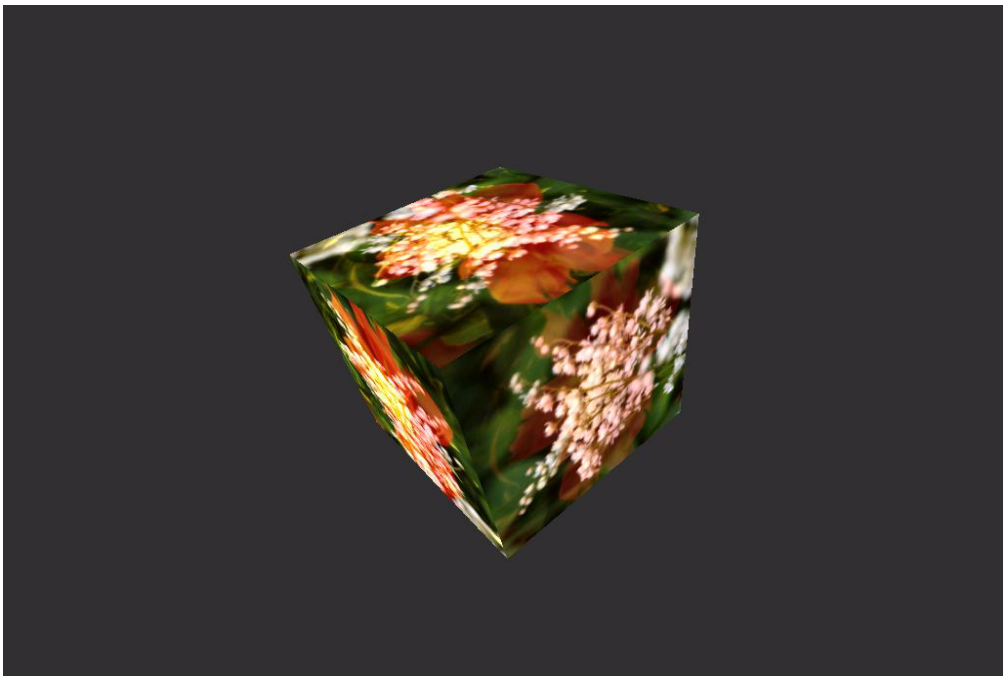


Figure 16-2. Cube Test

16.5.2.4 Triangle Test

This test application verifies the basic functionality of OpengGL ES 2.0. It is included in the release image and is located under `\Windows\triangle.exe`. Click to launch this test and a triangle appears on the screen as shown in follow firgure. Press ESC to exit this application.

Figure 16-3. Triangle Test

16.5.2.5 OpenGL ES 1.1/2.0 Conformance Test

The OpenGL ES 1.1 and 2.0 conformance tests are standard OpenGL ES 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 these tests.

16.5.2.6 Known Issues

- Refer to the release notes for up-to-date known issue list

16.6 GPU Driver API Reference

- For OpenGL ES 1.1 and 2.0 API refer to http://www.khronos.org/opengles/ for detailed specifications
- 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 17
Inter-Integrated Circuit (I^2C) Driver

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

17.1 I^2C Driver Summary

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

Table 17-1. I^2C Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\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_I2CBUS2=1 or BSP_I2CBUS3=1

17.2 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. Stream interface
5. Two power management modes: full on and full off

17.3 Hardware Operation

17.3.1 Conflicts with Other Peripherals and Catalog Items

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

17.3.1.1 Conflicts with SoC Peripherals

No conflicts.

17.3.1.2 Conflicts with Board Peripherals

No conflicts.

17.4 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.

17.4.1 Registry Settings

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

17.4.1.1 i.MX53 Registry Settings

The following registry keys are required to properly load the I^2C module.

```
IF BSP_I2CBUS2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\I2C2]
   "Prefix"="I2C"
   "Dll"="i2c.dll"
   "Index"=dword:2
   "Order"=dword:2
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"          ; PMCLASS_GENERIC_DEVICE
ENDIF    ; BSP_I2CBUS2

IF BSP_I2CBUS3
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\I2C3]
   "Prefix"="I2C"
   "Dll"="i2c.dll"
   "Index"=dword:3
   "Order"=dword:2
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"          ; PMCLASS_GENERIC_DEVICE
ENDIF    ; BSP_I2CBUS3
```

17.4.2 Communicating with the I^2C

The I^2C is a stream interface driver, and is thus accessed through the file system APIs. To communicate using the I^2C, a handle to the device must first be created using the **CreateFile** function. Subsequent

commands to the device are issued using the **DeviceIoControl** function with IOCTL codes specifying the desired operation. The following are the basic steps. The I^2C driver is provided to hide all the IOCTL calls from the calling application.

17.4.3 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. 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.

Example 17-1shows how to open an I^2C port.

Example 17-1. Code to Open 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.

17.4.4 Configuring the I^2C

Configuring the I^2C port for communications involves two main operations:

- Setting the master 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.

Example 17-2 shows the code to configure an I^2C port:

Example 17-2. Code to Configure 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;
}
```

17.4.5 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 required 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

   c) 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.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
```

17.4.5.1 }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.

17.4.6 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.

17.4.7 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.

17.4.7.1 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.

17.4.7.2 PowerDown

This function is not implemented for the I^2C driver.

17.4.7.3 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.

17.5 Unit Test

The functionality of this driver is tested by other drivers, like eCompass, Accelerometer, SSI drivers and so on.

So far, the CTK I2C test is not applicable.

17.6 I^2C Driver API Reference

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

17.6.1 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.

17.6.1.1 I2C_IOCTL_GET_CLOCK_RATE

This **DeviceIoControl** request retrieves the clock rate

Parameters

lpOutBuffer	Pointer to the divisor index clock rate. The true clock frequency is platform dependent. See the I^2C specification for more information
nOutBufferSize	Size in bytes of the divisor indexclock rate

17.6.1.2 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

17.6.1.3 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

17.6.1.4 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

17.6.1.5 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.

17.6.1.6 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 clock rate divisor index.divisor index. See the I^2C specification to obtain the true clock frequency
nInBufferSize	Size in bytes of the clock rate divisor index divisor index

17.6.1.7 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, see 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

17.6.1.8 I2C_IOCTL_SET_MASTER_MODE

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

17.6.1.9 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.

17.6.1.10 I2C_IOCTL_SET_SLAVE_MODE

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

17.6.1.11 I2C_IOCTL_TRANSFER

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

Parameters

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

nInBufferSize Size in bytes of the I2C_TRANSFER_BLOCK

17.6.1.12 I2C_IOCTL_ENABLE_SLAVE

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

17.6.1.13 I2C_IOCTL_DISABLE_SLAVE

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

17.6.1.14 I2C_IOCTL_GET_SLAVESIZE

This DeviceIoControl request gets the interface buffer size of the I^2C device for slave mode.

17.6.1.15 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.

17.6.1.16 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.

17.6.1.17 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.

17.6.2 I^2C Driver SDK Encapsulation

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

17.6.2.1 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

17.6.2.2 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

17.6.2.3 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

17.6.2.4 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

17.6.2.5 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

17.6.2.6 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

17.6.2.7 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. See the I^2C specification to obtain the true clock frequency
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

17.6.2.8 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. See the I^2C specification to obtain the true clock frequency
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

17.6.2.9 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, see 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

17.6.2.10 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

17.6.2.11 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

17.6.2.12 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

17.6.2.13 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

17.6.2.14 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

17.6.2.15 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

17.6.2.16 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

17.6.2.17 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

17.6.2.18 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

17.6.2.19 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

17.6.3 I^2C Driver Structures

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

17.6.3.1 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;
```

Parameters

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

17.6.3.2 I2C_TRANSFER_BLOCK

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

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

Parameters

pI2CPackets Pointer to an array of I2C_PACKET objects

iNumPackets Number of I2C_PACKET objects pointed to by pI2CPackets

Chapter 18
IIM(IC Identification Module) Driver

The IC Identification Module (IIM) provides an interface for reading and in some cases programming and/or overriding identification and control information stored in on-chip fuse elements. The module supports laser fuses (L-Fuses) and/or electrically-programmable poly fuses (e-Fuses). The IIM driver only supports e-Fuses operation.

18.1 IIM Driver Summary

The IIM driver provide three basic functionalities for IIM operation: fuse reading, fuse sensing and fuse programming.

Both fuse reading and fuse sensing are used to read the value of a fuse, still, there are differences between them: fuse reading is used to read the value of software fuse value in shadow cache, while fuse sensing is used to read the fuse elements themselves. The reasons for caching the fuse values are to reduce the risk of accidental programming of e-Fuses due to repeated reads, and to reduce power consumption associated with sense cycles.

Table 18-1 provides a summary of source code location, library dependencies and other BSP information.

Table 18-1. Flash Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\IIM
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\IIM
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\IIM
Driver DLL	iim.dll
SDK Library	N\A
Catalog Item	N\A
SYSGEN Dependency	N\A
BSP Environment Variables	BSP_IIM=1

18.2 Supported Functionality

Fuse reading, fuse sensing and fuse programming.

18.3 Hardware Operation

Refer to the chapter of IIM in the *User guide or Reference Manual* for detailed operation and programming information.

18.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

18.4 Software Operation

IIM module is packed to a stream interface module. This means one can easily use the functionality by passing IOCTL requests into it.

Table 18-2. IOCTL request

IOCTL code	functionality
IOCTL_IIM_FUSE_READ	fuse reading
IOCTL_IIM_FUSE_SENSE	fuse sensing
IOCTL_IIM_FUSE_PROGRAM	fuse programming

Those IOCTL requests can be passed into IIM.dll by calling DeviceIoControl. There is an example about how to do this. Please refer to ..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\UUT\bspuut.cpp for details.

18.4.1 Fuse reading

Input parameter: DWORD FuseAddr.

FuseAddr is the address of the fuse to read.

Output parameter:BYTE FuseValue.

FuseValue is the value of the fuse to read.

18.4.2 Fuse reading

Input parameter: FuseSenseAddr Addr;.

FuseSenseAddr is a struct:

typedef struct {

DWORD AddrOffset;

BYTE Bit;

}FuseSenseAddr, *PFuseSenseAddr;

AddrOffset is the address of the fuse to sense.

Bit is the bit offset of the fuse to sense.

Output parameter:BYTE FuseValue.

FuseValue is the value of the fuse to sense.

18.4.3 Fuse reading

Input parameter: FuseProg Fuse;.

FuseProg is defined like this:

typedef struct {

DWORD AddrOffset;

BYTE val;

}FuseProg, *PFuseProg;

AddrOffset is the address of the fuse to program.

val is the value of the fuse to program.

Output parameter: None.

18.5 Unit Test

There is no CTK provided for this module. The best way to test the module is to create a testbench which calls the IOCTL of the module.

Another way is to use MfgTool to do the test. Regarding usage of MfgTool, you can refer to MfgTool chapter.

Regarding to the fuse address and bit definition, please refer to i.MX53 fuse mapping table if you are authenticated to get it.

18.5.1 Fuse reading

Add below sentense to ucl.xml:

<CMD type="push" body="FuseRead:0x00000860">read fuse value.</CMD>

Here 0x00000860 is the address of fuse, change it to what you want.

You can open a console to get the value of the fuse to read.

18.5.2 Fuse Sensing

Add below sentense to ucl.xml:

<CMD type="push" body="FuseSense:0x00000860:0x00000003">read fuse value.</CMD>

Here 0x00000860 is the address of fuse, 0x00000003 is the bit offset of the address. Change them to what you want.

You can open a console to get the value of the fuse to sense.

18.5.3 Fuse Programming

Add below sentense to ucl.xml:

<CMD type="push" body="FuseProgram:0x00000860:0x00000010">read fuse value.</CMD>

Here 0x00000860 is the address of fuse, 0x00000010 is the value of the fuse to program. Change them to what you want.

You can open a console to check the result.

Warning:the operation will lead to un-resumable result since a fuse is one-time programmable element. Please make sure you fully understand the meaning of the fuse to be programmed.

Chapter 19
NAND Flash Driver

The NAND flash driver provides the functionality of NAND storage accessing. The flash driver follows Windows Embedded Compact 7 flash driver PDD/MDD architecture.

19.1 NAND Flash Driver Summary

Windows CE provides driver support for flash media devices using MDD or PDD architecture. The MDD allows NAND flash storage to be exposed as a block driver that is accessed by file system. The PDD wraps FMD layer 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 19-1 provides a summary of source code location, library dependencies and other BSP information.

Table 19-1. Flash Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\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> Storage Drivers > MSFlash Drivers> Samsung K9LBG08U0D / Micron MT29F16G08ABACA NAND Flash support
SYSGEN Dependency	sysgen_flashmdd=1
BSP Environment Variables	BSP_NONAND_FMD=

19.2 Supported Functionality

The Flash driver provides the following support:

1. Supports the Windows CE MDD or PDD interface
2. Supports both MLC and SLC NAND

3. Supports both 2 Kbyte and 4 Kbyte page size NAND

4. Supports MLC NAND Flash K9LBG08U0D as default

5. Supports SLC NAND Flash MT29F16G08ABACA

19.3 Hardware Operation

The Flash driver use NFC module to operate NAND flash chips. For detailed operation and configuration information, see the NFC chapter in the i.*MX53 Applications Processor Reference Manual*.

19.4 Software Operation

The development concepts for flash media drivers are described in the Windows Embedded Compact 7 Help Documentation section under the topic Windows Embedded Compact 7> Device Drivers> Flash Drivers >. The NAND FMD supported in the i.MX BSP implements the required FMD functions for interfacing to NAND Flash devices.

19.4.1 NAND Flash Driver Registry Settings

19.4.1.1 This section explains about the Flash driver registry settings.**NAND Flash Driver Registry Setting**

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\FlashPDD]
    "Dll"="flashmdd.dll"
    "FlashPddDll"="flashpdd_nand.dll"
    "Prefix"="DSK"
    "Profile"="MSFlash"
    "IClass"=multi_sz:"{A4E7EDDA-E575-4252-9D6B-4195D48BB865}",
                      "{8DD679CE-8AB4-43c8-A14A-EA4963FAA715}"
    "FriendlyName"="NAND Flash Driver"
    "Order"=dword:20
    "Priority256"=dword:76
; @CESYSGEN IF FILESYS_FSREGHIVE
    ;"Flags"=dword:1000
; @CESYSGEN ENDIF FILESYS_FSREGHIVE

[HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\MSFlash]
    "DefaultFileSystem"="FATFS"
    "PartitionDriver"="FLASHPART.dll"
    "AutoFormat"=dword:1
    "AutoPart"=dword:1
    "AutoMount"=dword:1
    "Name"="NAND FLASH Storage"
    "Folder"="NANDFlash"
    ;"FormatExfat"=dword:1
; @CESYSGEN IF FILESYS_FSREGHIVE
    "FormatTfat"=dword:1
    ;"MountAsBootable"=dword:1
    ;"MountPermanent"=dword:1
; @CESYSGEN ENDIF FILESYS_FSREGHIVE
```

Note: AutoFormat and AutoPart are set to 1 to enable automatical formating and partitioning operation for NAND flash disk. If one needs to manually do this, please set them to 0.

19.4.2 NAND Flash Driver Optimization

There are performance and power comsumption optimization in NAND flash driver, to get this, below code in public directory is moved into COMMON_FSL_V3\Nand:

..\WINCE700\public\COMMON\oak\drivers\block\msflash\FlashPddDispatch

..\WINCE700\public\COMMON\oak\drivers\block\msflash\FlashPddFmdWrapper

In COMMON_FSL_V3\Nand, there are two method are provided: big sector interleave and OCQ interleave to implement interleaving operation. Only OCQ interleave method is guaranteed to work, big sector interleave method is provided as a reference.

19.5 Power Management

The NAND FMD currently does not support power management.

19.6 Unit Test

The Flash driver is subject to one test suites provided by Windows Embedded Compact Test Kit (CTK). It can be found in CTK catalog: storage media\Flash.

CTK Test	Command Line
File System Driver Test for Flash	tux -o -d fsdtst -c "-p MSFlash -zorch"
File System Performance Test for Flash	tux -o -d fsperflog -c "-p MSFlash -zorch"
Partition driver Test for Flash	tux -o -d msparttest.dll -c "-profile MSFlash -zorch -store"
Storage Device Block Driver API Test for Flash	tux -o -d disktest -c "-p MSFlash -zorch -part -sectors 256"
Flash Driver PDD validation Test for Flash	tux -o -d flashpddtest -c "-p MSFlash -zorch"
Flash Memory Read Write Performance Test	tux -o -d flshwear -c "/profile MSFlash /store /flash"

Note: Please skip Storage Device Block Driver ReadWrite Test for Flash item in the catalog because the test doesn't use IOCTL_FLASH_XXX interface to access NAND flash driver.

19.6.1 System Testing

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

Use the Start –> Settings –> Control Panel –> Storage Manager to format and create partitions on the mounted NAND device .

Establish ActiveSync connection over USB and transfer files to/from the NAND storage.

Write media files to NAND storage. Use Windows Media Player to playback media files from NAND storage.

19.6.2 Performance Testing

The below tool is provided to test NAND flash driver performance:

..\WINCE700\support\TEST\FLASHRW

The test is based on file system level and result shows the performance of file read/write.

Chapter 20
Power Management IC (PMIC)

The LTC3589 PMIC is power management and user interface power Management and User Interface component to support for controlling the PMIC multi-function device.

20.1 PMIC Summary

This chapter provides information to develop:

- Device drivers that interface directly with the power management IC (PMIC) hardware components. The PMIC that is specifically referenced in this document is the LTC3589.
- 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 pmic driver which allows complete access to the functionality of the PMIC. This document is intended for device driver and application developers who need to understand and gain access to the functionality provided by the PMIC. Table 20-1 provides a summary of source code location, library dependencies and other BSP information.

Table 20-1. PMIC Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\PMIC\LTC3589
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\PMIC\LTC3589
Driver DLL	pmicPdk_ltc3589.dll
SDK Library	pmicSdk_ltc3589.lib
Catalog Item(s)	N/A
SYSGEN Dependency	N/A
BSP Environment Variable(s)	BSP_PMIC_LTC3589= 1 BSP_NOPMIC=

20.2 Supported Functionality

The PMIC device driver framework for Windows CE is a stream interface driver .It provides a SDK dll.A description of the stream interface driver will be found in the Windows CE Platform Builder documentation at **Windows Embedded Compact 7 > Device 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. Regulators

20.3 Hardware Operation

Refer to the LTC3589 document for details on the LTC3589 PMIC.

20.3.1 Conflicts with Other On-Chip Peripherals

20.3.1.1 i.MX53 Peripheral Conflicts

No conflicts.

20.3.2 Conflicts with Other ARD Peripherals

No conflicts.

20.4 Software Operation

20.4.1 Configuring the PMIC

The PMIC module 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.

20.4.2 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 pmic hardware. If the pmic 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")));
}
```

20.4.3 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.

20.4.4 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.

20.4.5 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 is the handle returned by the CreateFile function call that opened the PMIC port.

20.4.6 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.

20.4.6.1 PowerUp

This function is not implemented for the PMIC driver.

20.4.6.2 PowerDown

This function is not implemented for the PMIC driver.

20.4.6.3 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.

20.4.6.4 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.

20.4.6.5 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.

20.4.7 PMIC Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PMI]
   "Prefix"="PMI"
   "Dll"="pmicpdk_ltc3589.dll"
   "Index"=dword:1
   "Order"=dword:2
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"     ; PMCLASS_GENERIC_DEVICE
```

20.4.8 DMA Support

No support.

20.5 Unit Test

20.5.1 Unit Test Hardware

The LTC3589 PMIC ARD boards are required.

20.5.2 Unit Test Software

No software is necessary for this test.

20.5.3 Running the PMIC Tests

The PMIC driver has no CTK test case.The pmic fuction is tested by other driver.

Chapter 21
Serial Driver

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

21.1 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 21-1 provides a summary of source code location, library dependencies and other BSP information. Supported Functionality

Table 21-1. Serial Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\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*>: ARMV7 −> Device Drivers > Serial −> UART2 serial port support Third Party −> BSP −> Freescale <*Target Platform*>: ARMV7 −> Device Drivers −> Serial −> UART3 serial port support
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_SERIAL_UART2 =1 BSP_SERIAL_UART3 =1

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 baud rate up to 4 Mbps

NOTE

For low power consideration, the input clock of the UART driver is 24 MHz, other than 66.5 MHz, so the actual max baudrate is 1.5 Mbps.

21.2 Hardware Operation

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

21.2.1 Conflicts with Other Peripherals and Catalog Items

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

21.2.1.1 Conflicts with SoC Peripherals

All UART pins can be configured for alternate functionality (PATA, USB, CAN) using the i.MX53 IOMUX. The configuration is specified by the BSP serial driver. Changing this configuration can result in a conflict and prevent proper operation of the UART.

21.2.1.2 Conflicts with Board Peripherals

NA

21.3 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 MSDN Help documentation at the following location:

Windows Embedded Compact 7 -> Device Drivers -> Serial Port Drivers.

21.3.1 Registry Settings

This section explains the registry settings used to load the serial driver.

21.3.2 Power Management

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

21.4 Unit Test

The serial driver is tested using the Serial Port Driver Test and the command line is following:

tux -o -d serdrvbvt -c "-p COMn:"

Note: n is COM number

The Serial Port Test assesses if 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.

21.4.1 Unit Test Hardware

The following hardware is used for the unit test:

- i.MX53 ARD board

21.4.2 Unit Test Software

Table 21-2 lists the required software to run the unit tests.

Table 21-2. 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
SerDrvBvt.dll	Test.dll file for Serial Port Driver Test

21.4.3 Building the Unit Tests

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

```
[Drive]:\Program Files\WindowsEmbeddedCompact7TestKit\tests\target\armv7
```

21.4.4 Running the Unit Tests

The Serial Port Driver Test executes the `tux -o -d serdrvbvt -c "-p COMn:"` command line on default execution.

For detailed information on the Serial Port Tests, see

Windows Embedded Compact 7 -> Compact Test Kit(CTK) -> Communication Bus Tests -> Serial Port Driver BVT Test > Serial Port Driver in the MSDN 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 21-3 describes the Serial Port driver test cases.

Table 21-3. 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.

Table 21-3. Serial Port Driver Test Cases (continued)

Test Case	Description
1007	Makes the **WaitCommEvent** function return a value when the handle for the current COM port is closed. This test fails if the **WaitCommEvent** function behaves improperly.
1008	Tests the **SetCommTimeouts** function and verifies that the **ReadFile** function properly times out when no data is received. This test fails if the COM timeouts do not function correctly.
1009	Verifies that previous Device Control Block (DCB) settings are preserved when the **SetCommState** function call fails with DCB settings that are not valid. This test fails if the serial port driver does not keep previous DCB settings when DCB settings that are not valid are passed to the driver.

21.5 Serial Driver API Reference

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

Windows Embedded Compact 7 -> Compact Test Kit(CTK) -> Communication Bus Tests -> Serial Port Driver BVT Test > Serial Port Driver

21.5.1 Serial PDD Functions

Table 21-4 shows a mapping of Serial PDD functions to the functions used in the serial driver.

Table 21-4. 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

Table 21-4. Serial PDD Functions (continued)

PDD Function Pointer	Serial Driver Function
HWSetRTS	SL_SetRTS
HWEnableIR	SerEnableIR
HWDisableIR	SerDisableIR
HWClearBreak	SL_ClearBreak
HWSetBreak	SL_SetBreak
HWXmitComChar	SL_XmitComChar
HWGetStatus	SL_GetStatus
HWReset	SL_Reset
HWGetModemStatus	SL_GetModemStatus
HWGetCommProperties	SerGetCommProperties
HWPurgeComm	SL_PurgeComm
HWSetDCB	SL_SetDCB
HWSetCommTimeouts	SL_SetCommTimeouts

21.5.2 Serial Driver Structures

This section explains the serial driver structures.

21.5.2.1 UART_INFO

This structure contains information about the UART Module.

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

```
    CRITICAL_SECTION    RegCritSec
    ULONG    ChipID;
 } UART_INFO, * PUART_INFO;
```

Parameters

pUartReg	Pointer to UART Hardware registers
sUSR1	This value contains the UART status register
sUSR2	This value contains the UART status register
bDSR	This boolean value keeps the DSR state
UartType	This value contains the type of UART like DCE or DTE
UlDiscard	This is used to discard the echo characters in IrDa Mode
UseIrDA	This boolean value determines the driver is in IR mode or not
HwAddr	This value contains the hardware address of the UART Module
EventCallback	This is a callback to the Model Device Driver
pMDDContext	This contains the context of the UART, which is the first parameter to the callback function
dcb	This value contains the copy of Device Control Block
CommTimeouts	This contains the copy of CommTimeouts structure used to get and set the time-out parameters for a communication device
pBaudTable	Pointer to baud rate table
DroppedBytes	This value contains the number of bytes dropped
FlushDone	Handle to the flush done event
CTSFlowOff	This boolean value is used to store the CTS flow control state
DSRFlowOff	This boolean value is used to Store the DSR flow control state
AddTXIntr	This boolean value is used to fake a Tx interrupt
Status	This value contains the comm status
CommErrors	This value contains Win32 comm error status
ModemStatus	This value shows the Win32 Modem status
TransmitCritSec	This value is used as Critical Section for UART registers
RegCritSec	This value is used as Critical Section for UART
ChipID	This value contains Chip identifier (CHIP_ID_16550 or CHIP_ID_16450)

21.5.2.2 SER_INFO

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

```
typedef struct __SER_INFO {
    UART_INFO    uart_info;
    BOOL    fIRMode;
    DWORD    dwDevIndex;
    DWORD    dwIOBase;
```

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

Parameters

uart_info This structure contains information about UART

fIRMode This boolean value determines the module is FIR or serial

dwDevIndex This static value contains the device index value which is read from registry

dwIOBase This static value contains the I/O Base address of UART module which is read from registry

dwIOLen This static value contains the I/O length of UART Module which is read from registry

pBaseAddress Pointer to the start address of the UART registers mapped

cOpenCount Contains count of the concurrent open

CommProp Pointer to CommProp structure

pHWObj Pointer to PDDs HWObj structure

useDMA This boolean flag indicates if SDMA is to be used for transfers through this UART

SerialDmaReqTx SDMA request line for Tx

SerialDmaReqRx SDMA request line for Rx

SerialPhysTxDMABufferAddr Physical address of Tx SDMA address

SerialPhysRxDMABufferAddr Physical address of Rx SDMA address

pSerialVirtTxDMABufferAddr Virtual address of Tx SDMA address

pSerialVirtRxDMABufferAddr Virtual address of Rx SDMA address.

SerialDmaChanRx SDMA virtual channel indices for Rx

SerialDmaChanTx SDMA virtual channel indices for Tx

currRxDmaBufId	Index of the buffer descriptor next expected to complete its SDMA in the Rx SDMA buffer descriptor chains
currTxDmaBufId	Index of the buffer descriptor next expected to complete its SDMA in the Tx SDMA buffer descriptor chains
dmaRxStartIdx	Keeps the start index of byte to be delivered to MDD for Read
availRxByteCount	This variable keeps the remaining bytes in the Rx SDMA buffer
awaitingTxDMACompBmp	Indicates if an SDMA request is in progress on Tx SDMA buffer descriptor
dmaTxBufFirstUseBmp	Indicator for first time use of a Tx SDMA buffer descriptor
rxDMABufSize	Receive DMA buffer size
txDMABufSize	Transfer DMA buffer size

Chapter 22
Sony/Philips Digital Interface (SPDIF) Driver

The Sony/Philips Digital Interface (SPDIF) audio module is a stereo transceiver that allows the processor to transmit and receive digital audio.

22.1 SPDIF Driver Summary

The SPDIF driver module (`spdifdev.dll`) provides receiver (RX)functions as a waveform audio driver. For more information about the waveform audio driver, see the MSDN Help topic:

Windows Embedded Compact 7 -> Audio, Graphics and Media -> Waveform Audio

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

Table 22-1. SPDIF Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\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*>:ARMV7 > Device Drivers > SPDIF > SPDIF Input support
SYSGEN Dependency	SYSGEN_AUDIO
BSP Environment Variables	BSP_NOAUDIO= BSP_SPDIF=1

22.2 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 Compact 7 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. RX function support
7. Support 44.1KHz, 48 KHz sample rate

22.2.1 Conflicts with Other Peripherals and Catalog Items

22.2.1.1 Conflicts with SoC Peripherals

No conflicts

22.2.1.2 Conflicts with board Peripherals

No conflicts.

22.2.2 Known Issues

The SPDIF driver may cause the audio playback driver CTK to fail. To run the audio playback driver CTK, remove the SPDIF driver from the catalog temporarily or run the AudioRouting application to select Audio Output/Input as the default device.

Because there is not exact 22.5792MHz oscillator on board, 44.1KHz sample rate has a little inaccuracy, and some TV/monitor with poor compatibility may not play the audio with that samplerate.

22.3 Software Operation

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

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

22.3.1 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.

22.3.2 Compile-Time Configuration Options

The following table shows the compile-time configuration options.

Table 22-2. 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_RX_WATERMARK	The receiver watermarks that are to be used with SPDIF RX FIFO. The default is 16.
SPDIF_RX_ENABLED	Enable/Disable SPDIF RX module by define/undef this macro.

22.3.3 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:2
"Order"=dword:6
;"Priority256"=dword:99
"IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"       ; PMCLASS_GENERIC_DEVICE
```

22.3.4 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 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 Compact 7 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.

22.4 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.

22.4.1 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.

22.4.1.1 i.MX53 PowerUp Support

Power enables the clock and exits the SPDIF from lower-power mode.

22.4.2 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.

22.4.2.1 i.MX53 Power Down Support

Power gates the clock and sets the SPDIF to lower-power mode.

22.4.2.2 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
```

22.5 Unit Test

22.5.1 Unit Test Hardware

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

Table 22-3. Hardware Requirements

Requirement	Description
M-Audio Card on PC	M-Audio Card to send/receive SPDIF digital data

22.5.2 Unit Test Software

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

Table 22-4. 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

22.5.3 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: `\WINCE700\SUPPORT\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.

22.5.4 Running the Unit Tests

The command line for running the SPDIF test is:

```
tux –o –n –d spdiftest.dll
```

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

22.6 System Testing

In addition to running the SPDIF driver tests, 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 `WINCE700\PUBLIC\COMMON\SDK\SAMPLES\AUDIO`).

```
pwfx->wFormatTag = WAVE_FORMAT_WMASPDIF; // SPDIF FORMAT
```

For perform this testing, a SPDIF receiver device which can be used to receive audio data from the i.MX53 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 [out] <—> line dual-optical interface <—> i.MX53 SPDIF RX optical port

Then Spectralab can be used play audio data to the ARD SPDIF device.

22.7 SPDIF Driver API Reference

SPDIF driver is a standard waveform audio driver. For detailed reference information for the SPDIF driver, see the MSDN Help:

Windows Embedded Compact 7 -> Device Drivers -> Audio Drivers -> Waveform Audio Driver Reference

Chapter 23
Touch Panel Driver

The touch screen interface can be provided I2C multi-finger Touch controller(HSD100).The soc use I2C get the touch panel point. And if the multi-finger Touch controller has any touch data to be sent to host computer,the controller pulls the interrupt pin low to generate an interrupt to host computer .

23.1 Touch Panel Driver Summary

Table 23-1 provides a summary of source code location, library dependencies, and other BSP information.

Table 23-1. Touch Panel Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\MTOUCH
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.MX53 ARD:ARMV7 -> Device Drivers -> TOUCH -> HSD100 Multi-finger TOUCH
SYSGEN Dependency	SYSGEN_TOUCH = 1
BSP Environment Variables	BSP_NOTOUCH= BSP_TOUCH_HSD100=1

23.2 Supported Functionality

The touch panel should conform to the standards as explained in the documentation below:

Windows Embedded Compact 7 > Device Drivers > Touch Screen Drivers

23.3 Hardware Operations

The hardware consists of a LCD Panel with a touch screen and a HSD100 touch controller and LVDS panel. The I^2C module sends control information to the HSD100 and reads back the touch samples. From the touch sample ,the touch controller will provide single touch point and multi touch point different report.More details about the I^2C can be found in “Inter-Integrated Circuit(I2C) Driver”.

23.3.1 Conflicts with SOC Peripherals

NO Conflict.

23.4 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 has to 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 is removed. The calibrated coordinates must be reported to the nearest one-quarter of a pixel.

The wince700 touch screen stream interface driver support Multi-finger touch function.When two fingers continue press,touch screen driver can get two points.The driver will report the two points to the Graphics, Windowing, and Events Subsystem (GWES).

23.4.1 Touch Driver Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Touch]
    "Prefix"="TCH"
    "Dll"="touch.dll"
    "Flags"=dword:8                ; DEVFLAGS_NAKEDENTRIES
    "Index"=dword:1
    "Order"=dword:25
    ; IClass = touch driver class & power managed device
    "IClass"=multi_sz:"{25121442-08CA-48dd-91CC-BFC9F790027C}",
                      "{A32942B7-920C-486b-B0E6-92A702A99B35}"
    "Priority256"=dword:6D         ; touch ist priority = 109
    ;Below values are not used; Kept just to keep the Touch BVT passing!
    "SysIntr"=dword:0

    "InitialSamplesDropped"=dword:2     ; Number of samples to be dropped after pen down
    "SampleRate"=dword:C8               ; samples per second, default is 200


; how long touch proxy will wait for touch driver to load
[HKEY_LOCAL_MACHINE\SYSTEM\GWE\TouchProxy]
    "DriverLoadTimeoutMs"=dword:1388 ; 5 seconds

[HKEY_LOCAL_MACHINE\SYSTEM\GWE\UserInput]
    "TouchInputTimeout"=dword:3E8 ; 1 second

[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]

IF BSP_TOUCH_HSD100
   "CalibrationData"="2164,1560 875,587 887,2496 3460,2519 3453,584 "
ENDIF

; For double-tap default setting
[HKEY_CURRENT_USER\ControlPanel\Pen]
```

```
"DblTapDist"=dword:18
"DblTapTime"=dword:637
```

23.5 Unit Tests

This section explains the unit tests.

23.5.1 Unit Test Hardware

Table 23-2 lists the hardware required to run the unit tests.

Table 23-2. Hardware Requirements

Requirement	Description
LCD panel	Display panel required for displaying graphics data.

23.5.2 Unit Test Software

Table 23-3 lists the software required to run the unit tests.

Table 23-3. 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
touchfunc.dll	The Test.dll File
touchbvt.dll	TBVT Test dll file

23.5.3 Running the Touch Panel Tests

The touch panel test cases can be run by entering the following:

```
tux -o -n -d touchfunc.dll -x <Test case id>
tux -o -n -d touchbvt.dll
```

The test case IDs are described in the documentation at:

Windows Embedded Compact 7>Compact Test Kit (CTK)>Input - Touch Tests

23.6 Touch Panel API Reference

The complete API reference is available in the documentation at:

Windows Embedded Compact 7 > Device Drivers > Touch Screen Drivers >Touch Stream Interface Driver Reference

Chapter 24
Temperature Sensor Driver

Temperature Sensor driver in Windows Embedded Compact 7 BSP is constructed as a stream interface driver that exposes I/O control codes (IOCTL_TPS_READ, IOCTL_TPS_WRITE, IOCTL_POWER_CAPABILITIES, IOCTL_POWER_SET, IOCTL_POWER_GET). The application uses these I/O control codes to access the Temperature Sensor function.

24.1 Temperature Sensor Driver Summary

Table 24-1 provides a summary of source code location, library dependencies and other BSP information.

Table 24-1. Temperature Sensor Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\TPS
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\TPS
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\TPS
Driver DLL	tps.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <Target Platform>: ARMV7 > Device Drivers > TPS > Temperature Sensor
SYSGEN Dependency	N/A
BSP Environment Variable	BSP_TPS=1

24.2 Supported Functionality

The Temperature Sensor driver provides the following support:

1. Real-time die temperature.
2. No extra pins required, access is through JTAG interface or Parallel CR Control port.
3. Supports two power management modes, full on and full off.

24.3 Hardware Operation

The i.MX SOC contains an on-chip Temperature Sensor which is included in SATA controller, i.e. the Synopses DesignWare Cores SATA AHCI Core.

The die's temperature is derived using measured voltages from the internal analog circuitry. These voltages are first converted to a digital value through an ADC, then these values are simply read from register. Refer to the SATA chapter in the hardware specification document for measuring and calculating the temperature.

Synopsys has verified the temperature sensor's function in the lab environment. The equation for the temperature calculation is a quadratic curve fit to simulation and experimental data. The temperature measurement has been proven to be supply-independent, with an absolute error of less than 5^{o}C. The temperature measurement can be made without disturbing data traffic.

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

24.3.1 Conflicts with Other Peripherals and Catalog Options

24.3.1.1 Conflicts with SoC Peripherals

No conflicts.

24.3.1.2 Conflicts with board Peripherals

No conflicts.

24.4 Software Operation

24.4.1 Application/User Interface to Temperature Sensor drives

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

The user software which requires access to the Temperature Sensor, does so through the self defined IOCTLs, such as IOCTL_TPS_READ. They provide interface to read temperature data.

24.4.2 Temperature Sensor Driver Configuration

The driver is configured into the BSP build by check the catalog item listed in Table 24-1. It defines the environment variable/configuration option: BSP_TPS for Temperature Sensor driver. Configuration for the Temperature Sensor is then provided through registry settings imported from platform.reg. These settings can be modified to select the Temperature Sensor prefix and index.

24.4.2.1 Prefix and Index

The default device prefix is “TPS”.

The default device index is 1.

24.4.3 Power Management

The Temperature Sensor 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 power state, according to inactivity settings configured in Power Manager. As for standard stream interface driver, PowerUp and PowerDown functions are called by the Device Manager.

The primary method for limiting power consumption in the Temperature Sensor 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 is turned off after initialization is completed. The Temperature Sensor driver turns on the clock and enables the Temperature Sensor module before processing any temperature measurement. After the measurement, the Temperature Sensor module is disabled and the clock is turned off.

24.4.3.1 PowerUp

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

24.4.3.2 PowerDown

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

24.4.3.3 IOCTL_POWER_SET

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

24.4.4 Registry Settings

24.4.4.1 Temperature Sensor driver

The Temperature Sensor 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\TPS]
```

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

Table 24-2. Temperature Sensor driver Registry Setting Values

Value	Type	Content	Description
Dll	sz	tps.dll	Driver dynamic link library

Standard registry entries also to be included for the Temperature Sensor after the above key are shown in Table 24-3.

Table 24-3. Temperature Sensor driver Registry Setting Values

Value	Type	Content	Description
Prefix	sz	“TPS”	Device identifier (combined with Index for TPS1 for example)
Index	dword	1	Instance of Temperature Sensor

24.5 Unit Test

The Temperature Sensor driver is tested using self defined test application.

24.5.1 Unit Test Hardware

Table 24-4 lists the required hardware to run the Temperature Sensor driver unit tests.

Table 24-4. Temperature Sensor driver Hardware Requirements

Requirement	Description
i.MX platform.	The i.MX SOC contains an on-chip Temperature Sensor which is included in SATA controller.

24.5.2 Unit Test Software

Table 24-5 lists the required software to run the Temperature Sensor driver unit tests.

Table 24-5. Software Requirements

Requirement	Description
tps.exe	Self defined test application, which read the temperature data and output.

24.5.3 Building the Temperature Sensor Tests

The source code for the Temperature Sensor driver unit tests can be found under the directory:

`\WINCE700\SUPPORT\TEST\TPS\`

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.

24.5.4 Running the Storage Media Tests

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

The command line for running the File System Driver Test is:

```
tps.exe
```

This application read the temperature data and output.

24.6 Basic Elements for Driver Development

This chapter provides details of the basic elements for driver development.

24.6.1 BSP Environment Variables

Table 24-6. BSP Environment Variables

Name	Definition
BSP_TPS	Set to enable Temperature Sensor driver

24.6.2 Mutual Exclusive Drivers

N/A.

24.6.3 Dependencies of Drivers

N/A.

24.7 Device API Reference

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

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 Temperature Sensor are correct, these functions are handled automatically, and need no further documentation here.

24.7.1 IOCTL_TPS_READ

This DeviceIoControl request returns the temperature data.

Parameters

hDevice — [in] Handle to the Temperature Sensor. The following code example shows how to open the Temperature Sensor.

HANDLE hTPS = CreateFile(TEXT("TPS1:"), GENERIC_READ | GENERIC_WRITE, 0, NULL, OPEN_EXISTING, 0, NULL);

lpOutBuffer — [out] Set to the address of an allocated UINT32 variable. This variable receives the temperature data when the IoControl call returns

nOutBufferSize	[out] Set to the size of the allocated UINT32 variable.
lpBytesReturned	[out] Pointer to a DWORD to receive the total number of bytes returned.

Chapter 25
Universal Serial Bus (USB) 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 25.1, "USB OTG Driver Summary.").

25.1 USB OTG Driver Summary

25.1.1 USB OTG Client Driver Summary

Table 25-1 provides a summary of source code location, library dependencies and other BSP information for the USB OTG client driver.

Table 25-1. OTG Client Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
Common SOC	COMMON_FSL_V3
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*>.lib usb_usbfn_os_<*Target SOC*>.lib usb_ufnmddbase_<*Common Soc*>.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBD
Import Library	N/A
Driver DLL	usbfn.dll

Table 25-1. OTG Client Driver Summary (continued)

Driver Attribute	Definition
Catalog Item	High Speed OTG: Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > 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 25.5.7, "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.

25.1.2 OTG Host Driver Summary

Table 25-2 provides a summary of source code location, library dependencies and other BSP information for the USB OTG host driver.

Table 25-2. OTG Host Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX53_ARD
Target SOC (TGTSOC)	MX53_FSL_V3
Common SOC	COMMON_FSL_V3
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*>.lib usbh_ehcdpdd_<*Common SOC*>.lib usbh_usb2com_<*Common SOC*>.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*>: ARMV7 > 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 25.5.8, "Class Drivers," for class driver information.

25.1.3 OTG (Pin-Detection) Driver Summary (For High-Speed Only)

Table 25-3 provides a summary of source code location, library dependencies and other BSP information for the USB OTG transceiver driver.

Table 25-3. OTG Transceiver Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX53_ARD
Target SOC (TGTSOC)	MX53_FSL_V3
Common SOC	COMMON_FSL_V3
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<Common Soc>\MS\USBOTG\MDD
CSP Static Library	usbotgcm_<Common SOC>.lib
Platform Driver Path	PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBOTG
Import Library	N/A
Driver DLL	fsl_usbotg.dll
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*>: ARMV7 > 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_USBOTG=1

25.2 USB Host Driver Summary

25.2.1 HS Host1 Driver Summary

Table 25-4 provides a summary of source code location, library dependencies and other BSP information for the HS host driver.

Table 25-4. HS Host1 Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX53_ARD
Target SOC (TGTSOC)	MX53_FSL_V3
Common SOC	COMMON_FSL_V3
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*>.lib usbh_ehcdpdd_<*Common SOC*>.lib usbh_usb2com_<*Common SOC*>.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBH\HSH1

Table 25-4. HS Host1 Driver Summary (continued)

Import Library	N/A
Driver DLL	hcd_hsh1.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV7 > Device Drivers > USB Devices > USB High Speed Host1 To support high speed host, choose .. >High Speed Host1
SYSGEN Dependency	SYSGEN_USB=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSH1=1

The host driver requires a set of class drivers to be loaded. See Section 25.5.8, “Class Drivers,” for more information.

25.2.2 HS Host2 Driver Summary

Table 25-5. HS Host2 Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX53_ARD
Target SOC (TGTSOC)	MX53_FSL_V3
Common SOC	COMMON_FSL_V3
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*>: ARMV7 > Device Drivers > USB Devices > USB High Speed Host2 To support only host mode, choose .. >High Speed Host Function.
SYSGEN Dependency	SYSGEN_USB=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSH2=1

The host driver requires a set of class drivers to be loaded. See Section 25.5.8, “Class Drivers,” for more information.

25.3 Supported Functionality

The OTG driver provides the following software and hardware support:

1. High Speed OTG/Host driver supports USB specification 2.0.
2. When a cable is not connected or a mini-B cable is connected (in either of these cases, the ID pin is pull up), OTG driver selects the peripheral driver to be in charge. On attaching a mini-A cable (in this case, the ID pin is pull down), OTG driver selects the host driver to be in charge.
3. OTG port as client/peripheral supports mass storage, RNDIS and serial clients
4. OTG port as host or HS Host supports mass storage, HID and RNDIS classes
5. When nothing is attached to the OTG/Host port, the driver configures the controller and transceiver into a low power state
6. When the system is suspended with nothing attached to the Host port, while a device attaching to Host port, the behavior of the system, remain suspended or resume can be configured by the compile option.
7. When the system is suspended while a device attached to the Host port, while it is unplugged, the behavior of the system, remain suspended or resume can be configured by the compile option
8. When the system resumes after suspend, any attached devices are enumerated and their class drivers loaded appropriately
9. Data transfer rates on the client port exceeds 40 Mbits/sec in Mass Storage client

25.4 Hardware Operation

25.4.1 Conflicts with Other Peripherals and Catalog Items

25.4.1.1 Conflicts with SoC Peripherals

No conflicts.

25.4.1.2 Conflicts with Board Peripherals

No conflicts.

25.5 Software Operation

25.5.1 USB OTG Host Controller Driver

This driver enables the USB host functionality for the OTG, H1 and H2 port. It is part of the standard Windows USB software architecture as shown in Figure 25-1.

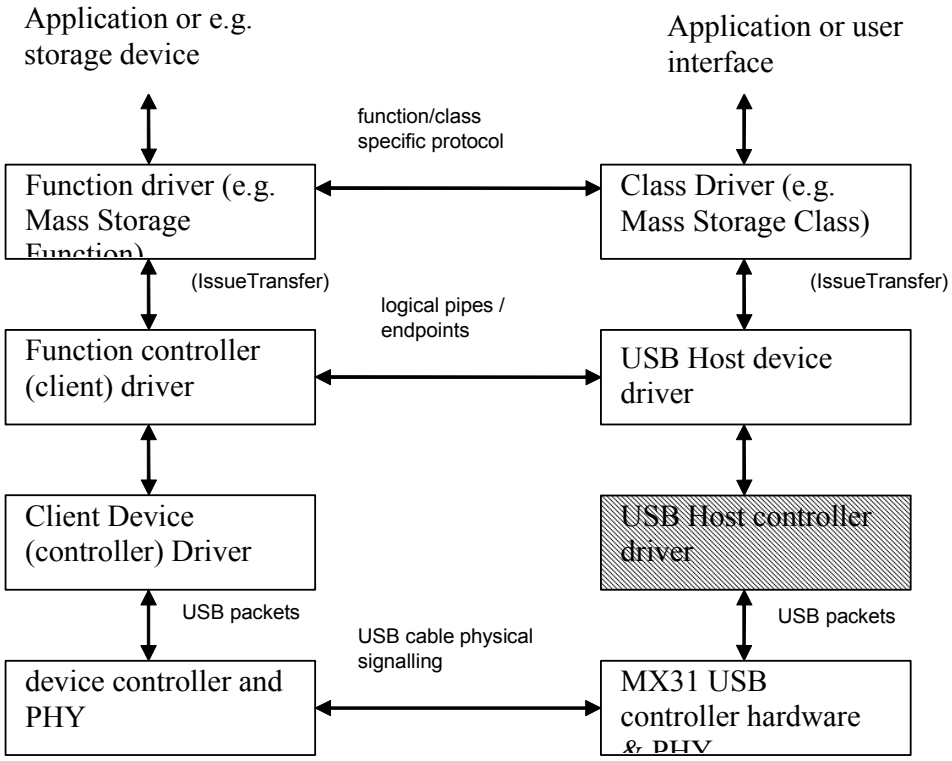


Figure 25-1. Windows USB Driver Architecture

For further details of the Windows CE USB driver architecture and usage, see the Platform Builder Windows Embedded Compact 7 help topic:

Windows Embedded Compact 7 > Device Drivers > USB Host Drivers

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.

25.5.1.1 User Interface

User access to the USB host driver is by class drivers. For further details on these Host Client Drivers refer to the Platform Builder Windows Embedded Compact 7 help topic:

Windows Embedded Compact 7 > Device Drivers > USB Host Drivers > USB Host Controller Drivers > Host Client Driver Reference.

25.5.1.2 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 25.5.1.5, "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.MX53 Applications Processor Reference Manual*.

25.5.1.3 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.

25.5.1.4 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 25.5.1.7.1, "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.

25.5.1.5 Registry Settings

25.5.1.5.1 OTG Registry Settings

Refer to the Section 25.5.5, "USB OTG Registry Settings," for information about OTG registry settings.

25.5.1.5.2 HSH1 Registry Settings

The USB OTG host controller settings are values located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\HCD_HSH1]
```

Table 25-6 lists the USB registry settings.

Table 25-6. hsh1.reg Default Values

Value	Type	Content	Description
Dll	sz	hcd_hsh1.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	02	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.
IClass	multi_sz	"{A32942B7-920C-486b-B0E6-92A702A99B35}"	This indicate the USB Host is a Generic power-manageable devices

25.5.1.5.3 HSH2 Registry Settings

The USB OTG host controller settings are values located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\HCD_HSH2]
```

Table 25-7 lists the USB registry settings.

Table 25-7. hsh2.reg Default Values

Value	Type	Content	Description
Dll	sz	hcd_hsh2.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	02	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.

Table 25-7. hsh2.reg Default Values (continued)

Value	Type	Content	Description
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.
IClass	multi_sz	"{A32942B7-920C-486b-B0E6-92A702A99B35}"	This indicate the USB Host is a Generic power-manageable devices

25.5.1.6 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).

WCE700 also provides Windows Embeded Compact Test Kit to do the related driver test. And it includes two parts of test, one part is included in Windows Embedded Compact Test Kit, and another one is included in Windows CEPC test

25.5.1.6.1 Windows Embedded CTK test

Documentation for the WCE700 CTK test refer to :

Windows Embedded Compact 7 > Compact Test Kit(CTK)

25.5.1.6.1.1 Prepare for CTK test

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> :ARMV7-Device Drivers-USB Devices-USB High Speed Host1-High Speed Host 1
   - Freescale <Target Platform> :ARMV7-Device Drivers-USB Devices-USB High Speed OTG Device
   - Core OS > Windows Embedded Compact > Device Drivers > USB > USB HOST > USB HOST Support
   - Core OS > Windows Embedded Compact > Device Drivers > USB > USB HOST > USB Class drivers, and all the sub-components of this catalog item (Sub-Components like USB Storage Class Driver.)
   - Core OS > Windows Embedded Compact > Device Drivers > USB > USB Function > USB Function Support.

— Core OS > Windows Embedded Compact > Device Drivers > USB > USB Function > USB Function Clients, and all the sub-components except Composite Function driver.
— Core OS > Windows Embedded Compact > File Systems And Data store > Storage Manager; (Sub-Components: FAT File System, Partition Driver, Storage Manager Control Panel Applet)
— Core OS > Windows Embedded Compact > ActiveSync > File Sync

3. Sysgen and build the image

After image generated, it should be downloaded to target device, for more information about image download, please refer to BSP reference guide.

25.5.1.6.1.2 Run CTK test

If Windows Embedded Compact Test Kit is included when you install Windows Embedded Compact 7, then you can find it at:

Start > All Programs > WindowsEmbededCompact7TestKit

If you want to run CTK with graph tool,before you can run test suites, you need to connect to device successfully. Or you can copy the required softwares to device and run it manually on device without graph tool.

The following test case suite need to be test for USB driver.

1. USB Function Driver verification tests

lTable 25-3 lists the software required for the USB Funtion Driver Verification Test.

Table 25-3. Software Requirements

Requirement	Description
Tux.exe	Test harness, required for executing the test .
Kato.dll	Logging engine, required for logging the test data .
ktux.dll	Test harness, required to execute kernel-mode tests.
usbfnbvt.dll	Library that contains the test code, loaded by the Tux test harness

execute "tux -o -n -d usbfnbvt" to run test case.

for more information about this test suite, please refer to:

Windows Embedded Compact 7 > Compact Test Kit(CTK) > USB - Function Tests

2. USB Host Driver verification tests

Table 25-4 lists the software required for the USB Host Driver Verification Test.

Table 25-4. Software Requirements

Requirement	Description
Tux.exe	Test harness, required for executing the test .
Kato.dll	Logging engine, required for logging the test data .

Table 25-4. Software Requirements (continued)

ktux.dll	Test harness, required to execute kernel-mode tests.
usbhostbvt.dll	Library that contains the test code, loaded by the Tux test harness

execute "tux -o -n -d usbhostbvt" to run test case.

for more information about this test suite, please refer to:

Windows Embedded Compact 7 > Compact Test Kit(CTK) > USB - Host Tests > USB Host Driver Verification Tests

3. USB OTG BUS IOCTL test

Table 25-5 lists the software required for the USB Host Driver Verification Test.

Table 25-5. Software Requirements

Requirement	Description
Tux.exe	Test harness, required for executing the test .
Kato.dll	Logging engine, required for logging the test data .
ktux.dll	Test harness, required to execute kernel-mode tests.
USB OTG Driver	Driver Library
otgtest.dll	Test library

execute "tux -o -n -d otgtest.dll" to run test case.

for more information about this test suite, please refer to:

Windows Embedded Compact 7 > Compact Test Kit(CTK) > USB - OTG Tests

4. USB Host High speed EHCI (2.0) interface tests

All these test cases require a CEPC Hardware, please refer to Table 25.5.1.6.2 to get more information

25.5.1.6.2 WCE700 CEPC test

25.5.1.6.2.1 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 25.5.1.1, “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 25-2.

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.

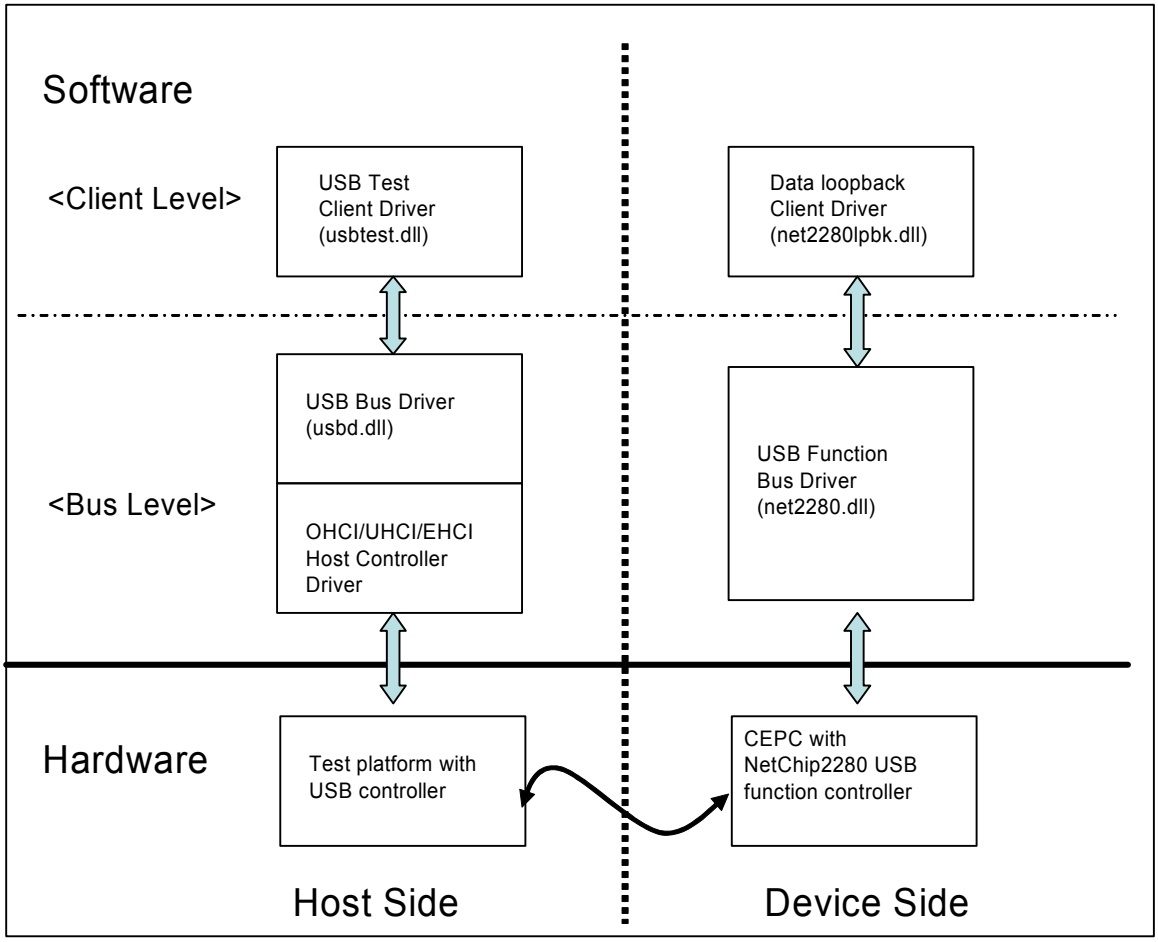


Figure 25-2. Test Setup

25.5.1.6.3 Unit Test Hardware

- Test platform
- Host Controller Card (if not onboard logic)
- CEPC
- Netchip2280 Card
- USB cable

25.5.1.6.4 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
  - Regular test: usbtest.dll, usbd.dll
  - Peformance test: ceperf.dll, perfscenario.dll, guidgenerator.dll, usbperf.dll
  - Stress test: usbstress.dll, xxx_clicker.dll

Device side requirements:

- Lufldrv.exe
- Net2280lpbk.dll
- NetChip2280 USB function support (net2280.dll) must be included in the CEPC run time image
- LpBkCfg1.dll
- LpBkCfg2.dll
- LpBkPerfCfg.dll

25.5.1.6.5 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

25.5.1.6.6 Test Cases

Table 25-2 shows the test cases contained in the test suite.

Table 25-2. 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 25-2 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

25.5.1.7 Platform-Specific API

This section describes the platform-specific API functions.

25.5.1.7.1 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

25.5.1.7.2 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

25.5.1.7.3 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

25.5.1.7.4 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

25.5.2 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

25.5.2.1 User Interface

The USB client driver provides a standard Windows CE USB driver implementation. For an overview see:

Windows Embedded Compact 7 > Device 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 Embedded Compact 7 Platform Builder help topic:

Windows Embedded Compact 7 > Device 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:

Windows Embedded Compact 7 > Device Drivers > USB Function Controller Drivers > USB Function Controller Driver Reference.

25.5.2.2 Client Driver Configuration

Refer to the Section 25.5.4, "USB OTG Catalog Settings," for information about the client driver configuration.

25.5.2.3 Registry Settings

Refer to the Section 25.5.5, "USB OTG Registry Settings," for information about the registry settings.

25.5.2.4 Unit Test

Beside the CTK tests for USB Funtion driver, there still has some test case for it. 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. These tests verifie 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.

25.5.2.4.1 Unit Test Hardware

Table 25-3 lists the required hardware to run the unit tests.

Table 25-3. 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
Storage medium such as Nand Flash, eSDHC, U-Disk	Required as a storage device when the board is configured as mass storage class

25.5.2.4.2 Unit Test Software

Table 25-4 shows the software requirements for the USB Function controller driver test.

Table 25-4. Software Requirements

Requirement	Description
ActiveSync 4.1 and above	Host side software that is required to be available for testing the Serial class functionality

25.5.2.4.3 Running the USB Function Controller Driver Tests

Table 25-5 lists USB function controller driver tests.

Table 25-5. 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

25.5.2.5 Platform-Specific API

This section describes the platform-specific API functions.

25.5.2.5.1 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

25.5.2.5.2 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

25.5.3 USB OTG Driver (Pin-Detection Driver)

This driver is responsible for detecting the type of USB connector plugged into the USB OTG socket of the i.MX System. It loads the USB host driver or USB peripheral driver and let it in charge.

25.5.3.1 User Interface

There is no user interface to the transceiver driver. This driver merely manages the USB host or peripheral drivers, which provide the appropriate programming API.

25.5.3.2 OTG Driver Configuration

See the Section 25.5.4, “USB OTG Catalog Settings” for information on the OTG driver configuration.

25.5.3.3 Registry Settings

See the Section 25.5.5, “USB OTG Registry Settings” for information on the registry settings.

25.5.3.4 Unit Test

It is tested using the mini or micro USB OTG plug A and mini or micro USB OTG plug B. The test is done by manually plugging in different cables to the OTG socket on the System and verifies if the appropriate driver is activated.

25.5.3.4.1 Unit Test Hardware

Table 25-6 lists the required hardware to run the unit tests.

Table 25-6. Hardware Requirements

Requirement	Description
Full OTG configuration selected in BSP	Make sure the OTG driver is running

Table 25-6. Hardware Requirements

PC (with appropriate driver and software installed) Peripherals such as thumb disk, USB keyboard, and hub	To test if control reaches the Host controller driver
mini or micro A to A receptacle cable mini or micro B to A cable	For connecting system with PC and peripherals. System acts as peripheral and host accordingly

25.5.3.4.2 Running the OTG Test

Table 25-7 lists OTG tests.

Table 25-7. OTG Tests

Test Cases	Entry Criteria, Procedure and Expected Results
Idle case when the cable is not plugged in	Entry Criteria: Ensure there is no cable connected and the board is turned ON, wait until the board boots-up completely Procedure: When the board is powered and completely booted-up, the board should be idle. Expected Result: Device boots up and is stable
Switch to peripheral	Entry Criteria: Ensure there is no mini USB OTG plug connected and the board is turned ON and wait until the board boots-up completely Procedure: When the board is powered and completely booted-up, connect the system to PC with the mini or micro B to A cable. Verify if PC recognizes it correctly. Expected Result: PC recognize the board (as peripheral) correctly (Activesync is active, or removable disk is visible, or network adaptor is recognized).
Switch to host	Entry Criteria: Unplug board from PC (in previous step) Procedure: 1. Disconnect the system with PC and connect a mini or micro A to A receptacle to the OTG socket. 2. Connect the USB peripheral device (such as a thumb disk) to the A receptacle. 3. The connected peripheral gets enumerated and starts functioning. For example, if an USB thumb disk is connected, a new disk is accessible on the CE system. Expected Result: Peripheral should start functioning on the CE system.
Switch between host and peripheral	Repeat the last 2 steps Expected Result: System always functions OK as both host and peripheral.

25.5.3.5 Platform-Specific API

NA.

25.5.4 USB OTG Catalog Settings

The driver is selected into the BSP build by dragging and dropping the appropriate catalog item for USB HS OTG.. There are 3 catalog items in **Freescale i.MX53 ARD: ARMV7> Device Drivers > USB Devices > USB High Speed OTG** related to USBOTG functionality:

(a) **High Speed OTG Port Full OTG Function**

(b) **High Speed OTG Port Pure Client Function**

(c) **High Speed OTG Port Pure Host Function**

The selection of (a) implicitly selects (b) and (c), without selecting (a), (b) and (c) separately. So there are 3 possible configurations available for BSP users:

(1) All 3 catalogs are explicitly or implicitly selected, corresponding to both host and peripheral support plus OTG pin detection.

(2) Only **High Speed OTG Port Pure Client Function** is selected, corresponding to peripheral-only support.

(3) Only **High Speed OTG Port Pure Host Function** is selected, corresponding to host-only support.

25.5.5 USB OTG Registry Settings

3 possible configurations available in Section 25.5.4, "USB OTG Catalog Settings," forms 3 corresponding registry structure.

25.5.5.1 Registry Structure

- With configuration 1, for full OTG configuration, the generated registry has the following structure:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\UsbOtg]
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\UsbOtg\USBFN]
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\UsbOtg\Hcd]
```

- With configuration 2, for full peripheral-only configuration, the generated registry has the following structure:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\UFN]
```

- With configuration 3, for full host-only configuration, the generated registry has the following structure:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\HCD_HSOTG]
```

The contents in **BuiltIn\USBOtg\UsbFN** are similar to those in **BuiltIn\UFN** and the contents in **BuiltIn\UsbOtg\Hcd** are similar to those in **BuiltIn\HCD_HSOTG**. Most of the settings are common between the both. The differences are as follows:

In configuration 1, only **UsbOtg** key is located under **BuiltIn** key, which means the OTG driver is automatically loaded by the OS. In this case, the OTG driver decides to load the peripheral driver and the host driver.

In configuration 2 and 3, **UFN** or **HCD_HSOTG** is put directly under **BuiltIn** key. So the peripheral driver or host driver is loaded automatically by the OS.

25.5.5.2 Registry Key Settings

This section explains about the registry key settings.

25.5.5.2.1 OTG Driver Settings

Table 25-8 lists the USB OTG transceiver registry settings.

Table 25-8. USB OTG Transceiver Registry Settings

Value	Type	Content	Description
Dll	sz	fsl_usbotg.dll	Driver dynamic link library
IsrDll	sz		
DynamicClientLoad	dword	3	The value is set to 0x3, indicating both host driver and peripheral driver are loaded dynamically by the OTG driver.

25.5.5.2.2 Peripheral Driver Settings

Table 25-9 lists the USB OTG client registry settings.

Table 25-9. USB OTG Client Registry Settings

Value	Type	Content	Description
Dll	sz	usbfn.dll	Driver dynamic link library
OTGSupport	dword	0	obsolete setting, must be set as 0
Priority256	dword	64	The reference peripheral driver IST priority
OTGGroup	sz	1	This unique string (for example, 00 to 99) is used to combine or correlate instances of the host, function, and transceiver driver within one USB OTG instance

25.5.5.2.3 Host Driver Settings

Table 25-10 lists the default values for the host driver settings.

Table 25-10. OTG Host Default Values

Value	Type	Content	Description
Dll	sz	hcd_hsotg.dll	Driver dynamic link library
OTGSupport	dword	0	obsolete setting, must be set as 0
OTGGroup	sz	01	This unique string (for example, 00 to 99) is used to combine or 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	NA	This value represents the number of bytes allocated for the physical memory pool of the OTG host driver, and defaults to 128 Kbyte. From this buffer, 75% is 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()`. The BSP does not provide this setting and the driver uses the default 128 Kbyte size.

25.5.6 Power Management

The USB OTG driver enters the low power mode in the following cases:

- No bus activity for a specified period of time
- System enters the suspend state

Similar procedures are followed to let the USB module to enter or exit low power mode in either of the 2 cases. The following section explains about the description on the general power management procedures.

25.5.6.1 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).

25.5.6.2 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.

25.5.6.3 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

25.5.7 Function Drivers

The function drivers can be configured into the image using the Windows Embedded Compact 7 Platform Builder catalog, and are located at:

Core OS > **Windows Embedded Compact** > **Device Drivers** > **USB > 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.

25.5.7.1 Mass Storage Function

Table 25-11. 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
```

25.5.7.2 Serial Function

The primary use for the serial function is ActiveSync.

Table 25-12. 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 www.Microsoft.com to download the latest ActiveSync software for the PC. In some cases, DEBUGCHK may be triggered during attachment to ActiveSync in DEBUG builds.

When SYSGEN_USBFN_SERIAL is defined, the default registry entry is automatically included from:

```
PUBLIC\Common\OAK\FILES\common.reg
```

For commercial products, this registry entry must be copied into platform.reg and modified to over-ride the defaults. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Serial_Class]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
   "idVendor"=dword:045E
   "Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
   "idProduct"=dword:00ce
   "Product"="Generic Serial (PROTOTYPE--Remember to change idVendor)"
   "bcdDevice"=dword:0
```

25.5.7.3 RNDIS Function

The RNDIS function allows communication over USB to be supplied to ethernet NDIS interface of protocol stack.

Table 25-13. RNDIS Function

Driver Attribute	Definition
CSP Driver Path	N/A
CSP Static Library	N/A

Table 25-13. RNDIS Function

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

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
```

25.5.8 Class Drivers

All host ports support the same class drivers, and this configuration is common to all host ports. Class drivers must also be configured for the USB host ports. Class driver configuration is common to all host ports—there is no port-specific configuration to be completed on any class driver.

Table 25-14 shows the standard Microsoft-supplied drivers that are available by drag and drop from the catalog.

Table 25-14. 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 25.6.2, "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`

25.5.8.1 HID Mouse

For mouse support, the cursor is required to test and use the mouse as shown in Table 25-15.

Table 25-15. HID Mouse Class Driver

Catalog Item	Configuration Flag	Catalog Item
HID	SYSGEN_CURSOR	Core OS > Shell and User Interface > User Interface > Mouse

25.5.8.2 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 25-16) and include the appropriate stub keyboard and keyboard .dll (Table 25-17)

Table 25-16. 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 25-17. 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
```

25.6 Basic Elements for Driver Development

This section provides details of the basic elements for driver development in the Platform System.

25.6.1 BSP Environment Variables

Table 25-18 shows the system environment variables.

Table 25-18. System Environment Variables Summary

Name	Definition
BSP_USB	Set to configure USB in BSP
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.

25.6.2 Dependencies of Drivers

Table 25-19 summarizes the Microsoft-defined environment variables used in the BSP.

Table 25-19. 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_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

25.7 Application Tools for USB

An application tool is provided for USB device class selection.

25.7.1 Application Tool for USB Device Class Select

There are three types of USB device classes: ActiveSync, MSC and RNDIS. An application with a GUI is provided to switch between the three classes.

Figure 25-3 shows the tool to switch the USB device class. Make sure the OTG port is operating under the USB device mode (by connecting the mini-B connector of the USB OTG cable to the OTG port in the board) before pressing the Apply button to switch USB device class.

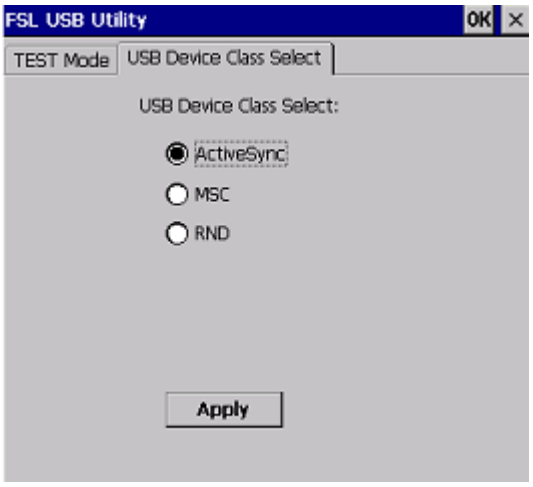


Figure 25-3. USB Device Class Switch User Interface

Chapter 26
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.

26.1 USB Boot and KITL Summary

Table 26-1 identifies the source code location, library dependencies, and other BSP information.

Table 26-1. USB Boot and KITL Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\MS\USBDBGRNDISMDD ..\PLATFORM\COMMON\SRC\COMMON\KITLDRV\USBDBG\USBDBGSERMDD
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\\<*Target SOC*>\USBDBGRNDISPDD
Platform Specific Path	..\PLATFORM\\<*Target Platform*>\SRC\COMMON\USBFN ..\PLATFORM\\<*Target Platform*>\SRC\KITL
Driver DLL	fsl_usbfn_rndiskitl.lib
SDK Library	N/A
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variable	N/A

26.2 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

26.3 Hardware Operation

For detailed operation and programming information of the USB OTG, see the chapter on the High-Speed USBOTG_UTMI in the corresponding platform User's Guide.

26.3.1 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.

26.4 Software Operation

This section explains about the software requirements for USB OTG.

26.4.1 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 26-1 illustrates the USB Boot and KITL software architecture.

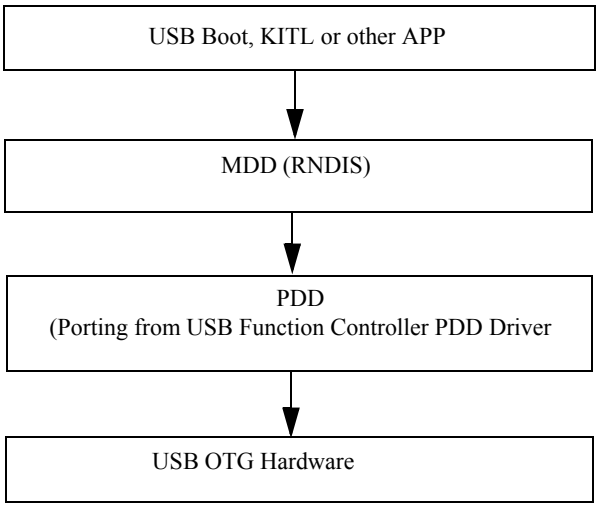


Figure 26-1. USB Boot and KITL Software Architecture Block Diagram

A RNDIS client MDD driver is implemented based on RNDIS client MDD driver provided by MSFT in Windows Embedded Compact 7. The original MSFT's code is in following location:

`%_WINCEROOT%\PLATFORM\COMMON\SRC\COMMON\KITLDRV\USBDBG\USBDBGRNDISMDD`

Also, a serial client MDD driver can be found here

`%_WINCEROOT%\PLATFORM\COMMON\SRC\COMMON\KITLDRV\USBDBG\USBDBGSERMDD`

26.4.2 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\<Target SOC>\USBDBGRNDISPDD`
- 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`
- Add USB Boot and KITL options into EBOOT menu

  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Eboot\menu.c`
- Add `oal_usbdbgsermdd.lib, fsl_usbdbgrndismdd_$(_COMMONSOCDIR).lib, fsl_usbdbgrndispdd_$(_SOCDIR).lib, usb_usbfn_eboot_$(_SOCDIR).lib` to `%_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`

26.4.3 Power Management

Power management is not implemented in USB Boot and KITL.

26.4.4 Registry Settings

There are no related register settings for the USB Boot and KITL.

26.5 Unit Test

The following section explains how to perform unit tests.

26.5.1 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.

26.5.2 Testing USB Boot and KITL

The steps to test USB Boot and KITL are as follows:

1. Connect the target board to a PC with a USB cable and power on the board.
2. At the EBOOT menu, change the boot configuration to match the following:

```
0) IP address: 192.168.0.2
1) Subnet Mask: 255.255.255.0
3) DHCP: Disabled
6) Set MAC Address : 0-12-34-56-78-12
I) KITL Work mode: Polling
K) KITL Enable Mode: Enable
```

```
P) KITL Passive Mode: Disable
E) Select Ether 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 pops up a Found New Hardware Wizard dialog box and prompts the user to install the driver for Microsoft Windows CE RNDIS virtual adapter on the Windows PC. The driver can be found at:WINCE700\platform\common\src\common\kitldrv\usbdbg\usbdbgrndismdd\host. If USB Serial is selected in option E, then no addtional driver need.
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. If USB Serial is selected in option E, then this step can be skip.
5. Check **Platform Builder Target** > **Connectivity** options to make sure the target device is selected.

   The image should be able to be download EBOOT. If USB Serial is selected in option E, please make sure the (auto)UsbSer is selected in Connectivity.
6. To test USB KITL, press **r** in the EBOOT menu to enable USB KITL. After the NK starts up, the KITL operates over the USB.

Chapter 27
UUT(Universal Updater Tool) Driver

The universal updater tool Driver provides the functionality of manufacturing tool (also called universal updater tool) client driver.

27.1 Universal Updater Tool Driver Summary

The UUT driver is a part of manufacturing tool to provide image burning functionality in mass production stage. Greatly different with typical BSP driver, UUT has no any dedicated hardware to drive. UUT is more likely an application which uses a number of drivers like USB, SD or NAND driver. We call UUT a driver just because it is packaged as a driver format and developed by BSP team.

UUT is so complicated and unique that we create an independent package to contain it. All the docs related to UUT usage, structure and mechanism are listed in Mfgtool package which is released along with BSP package.We only describe BSP related feature and resource here.

Table 27-1 provides a summary of source code location, library dependencies and other BSP information.

Table 27-1. Flash Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\UUT
SOC Specific Path	N\A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\UUT
Driver DLL	uut.dll, UUTApp.exe
SDK Library	N\A
Catalog Item	N\A
SYSGEN Dependency	N\A
BSP Environment Variables	BSP_UUT=1

Note: Different with typical BSP driver, UUT need a unique project to do building work. One can find the project in OSDesings\iMX53_ARD_UUT.

27.2 Supported Functionality

The UUT driver provides the following functionalities:

1. Supports imaging burning, including EBOOT and NK image.

2. Supports both NAND flash and SD/MMC meida, Regarding the type of NAND flash UUT can support, please refer to the dedicated driver part. In fact, UUT is an applicaiton which can invoke NAND flash driver functions. That is to say, UUT can supports the NAND flash type which is supported by NAND flash driver.
3. Supports file writing to specified media.
4. Supports OTP programming.

27.3 Hardware Operation

Please refer to <Manufacturing Tool Factory Operation manual.docx> in Mfgtool\Document.

27.4 Software Operation

Please refer to <MFG client driver guide.doc> in Mfgtool\Document.

27.5 Test operation

Please refer to <Manufacturing Tool User's Manual.doc> in Mfgtool\Document.

Chapter 28
Video Processing Unit (VPU)

The Video Processing Unit (VPU) is a multi-media video processing module. The multi-instance use case is supported by VPU API. This chapter describes the following topics:

- Brief information of VPU DLL
- API provided by Freescale which allow complete access to the full functionality of the VPU
- VPU control scheme based on the API with some practical programming issues

This document is intended for application developers who use the VPU to implement a high performance video codec and need to understand and gain access to the functionality provided by the VPU.

28.1 VPU Driver Summary

Table 28-1 provides a summary of source code location, library dependencies and other BSP information.

Table 28-1. VPU Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	MX53_FSL_V3
SOC Common Path	N/A
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\VPU
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\VPU
Driver DLL	vpu.dll
SDK Library	vpusdk_<*Target SOC*>.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*> > Device Drivers > VPU > Video Processing Unit Support
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_VPU=1

28.2 Supported Functionality

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

1. All APIs defined by Freescale
2. Interrupt mode
3. Multi-task function provided by the hardware
4. Power management using chip stop mode to power down the VPU while system suspends

5. Gates off VPU clock at any time when VPU is idle
6. Uses on chip RAM for performance-sensitive buffers, such as encode search RAM
7. Support decoding for:
   - H.264 BP/MP/HP
   - VC-1 SP/MP/AP
   - MPEG-4 SP/ASP except GMC
   - H.263 Base Profile
   - MPEG-1/2 MP@HL
   - MJPEG standards up to HD (1920×1080 or 2048×1024) resolution
   - JPG up to 8192×8192
8. Support encoding for:
   - H.264 up to BP@L3.0
   - H.263 Version 2 Interactive and Streaming Wireless Profile Level 60
   - MPEG4 up to SP@L5.0
   - MJPEG Baseline profile

For detailed VPU features, refer to the *i.MX5x VPU Application Programming Interface Windows Embedded Compact 7 Reference Manual*.

28.3 Hardware Operation

Refer to the chapter on Video Processing Unit (VPU) Chapter in the *i.MX53 Applications Processor Reference Manual* for detailed hardware operation and programming information.

28.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

28.4 Software Operation

28.4.1 Communicating with the VPU

The VPU software is divided into two parts: the driver and the API static library. The VPU driver is implemented as a stream interface driver and is thus accessed through the file system APIs. The static library, `VPUSDK_<Target SOC>.lib`, that wraps the file system APIs to access the VPU driver, opens the VPU driver to get a handle and calls the IOCTL codes to the driver to control the VPU hardware. Applications can easily use the APIs from the static library to control the VPU hardware regardless of the VPU stream interface driver.

28.4.2 Power Management

The VPU driver consumes power primarily through the VPU decode and encode operations. Even when the VPU is idle, the internal BIT processor consumes power. When the system enters the suspend state,

the VPU module is powered off to save power. To facilitate power management of the VPU module, the VPU 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.

28.4.3 Codecs Registry Settings

The following registry keys are required to properly load the decoder drivers:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\VPU]
   "Prefix"="VPU"
   "Dll"="vpu.dll"
   "Order"=dword:5

[HKEY_LOCAL_MACHINE\Drivers\VPU\Decoder]
   "UserDataBufferSize"=dword:10000
```

28.5 Unit Test

The VPU can be tested using a custom VPU test application.

28.5.1 Unit Test Hardware

Table 28-2 lists the required hardware to run the VPU application test.

Table 28-2. Hardware Requirements

Requirement	Description
Display plane	Need a display plane to show the video

28.5.2 Unit Test Software

Table 28-3 lists the required software to run the VPU application test.

Table 28-3. Software Requirements

Requirement	Description
vpu.dll	VPU stream interface driver
decdemo.exe	Decoding the bitstream date file and displaying the decoded images on the LCD
encdemo.exe	Encoding the YUV(4:2:0) file and saving the encoded stream to a file

28.5.3 Running the VPU Application Test

28.5.3.1 Decoding Test

The following items are needed to run the decoding test:

- Windows Embedded Compact 7 OS image with display plane support
- Windows Embedded Compact 7 Target Control and KITL support

- Bitstream data file

The procedure for the decoding test is as follows:

1. Change the `dec.cfg` configuration file according to the bitsteam format, image size to display, frame rate and other parameters. Detailed information is in the `readme.txt` and `dec.cfg` files.
2. Run Windows Embedded Compact 7 Target Control Debugging command `s decdemo.exe [path]\dec.cfg`.

The decoded image should be displayed on the display panel.

28.5.3.2 Encoding Test

The following items are needed to run the encoding test:

- Windows Embedded Compact 7 OS image with LCD support
- Windows Embedded Compact 7 Target Control and KITL support
- YUV(4:2:0) image to be encoded

The procedure for the encoding test is as follows:

1. Change the `enc.cfg` configuration file according to the bitsteam format, size of the YUV image, frame rate and other parameters. Detailed information is in the `readme.txt` and `enc.cfg` files.
2. Run Windows Embedded Compact 7 Target Control Debugging command `s encdemo.exe [path]\nec.cfg.`

The encoded stream should be saved to a file.

28.6 VPU Driver API Reference

The API functions are defined by Freescale and a third party IP vender. For details, refer to the *i.MX5x VPU Application Programming Interface Windows Embedded Compact 7 Reference Manual*.

28.7 Sample Demo Application

This section describes how to build and run the custom VPU test application. The VPU decoding demo application can be found in the following locations:

`\WINCE700\SUPPORT\APP\VPU\DECTEST`

The encoding demo application can be found under

`\WINCE700\SUPPORT\APP\VPU\ENCTEST`

The demo application provides an example of how to implement a video decoder or encoderusing the VPU video acceleration hardware by calling the predefined API.

28.7.1 System Requirements

In order to build and run the VPU demo application, the following requirements must be met:

- The OS image must be built with the VPU driver from the Catalog

- The OS image must include SD Host Controller drivers or storage drivers, such as ATA, NAND, SD from the Catalog to enable fast loading of test data

28.7.2 Building the OS Image and VPU Test Application

28.7.2.1 Building the OS Image

To build the image:

1. Include **Third Party** > **BSPs** > **Freescale <*Target Platform*>** > **Device Drivers** > **Video Processing Unit** on Windows Embedded Compact 7
2. Optionally include **Third Party** > **BSP** > **Freescale <*Target Platform*>** > **Device Drivers** > **SD Host Controller (**or **Storage Drivers)** on Windows Embedded Compact 7

28.7.2.2 Building and Running the Decoding Demo Application

To build and run the decoding demo application:

1. Click **Build** > **Open Release Directory in Build Window** on Windows Embedded Compact 7 to open the command prompt.
2. Run command **set wincerel=1** in command prompt window.
3. Change the current path to `\WINCE700\SUPPORT\APP\VPU\DECTEST`
4. Build the application with **build -c** command.
5. Run the VPU application from the Windows Embedded Compact 7 Target Control with the command `s dectest \release\dec.cfg` (if the `dec.cfg` file is copied to `\release` directory). Make sure the parameters set in the `dec.cfg` file are correct for the bitstream and hardware display. For detailed information, refer to the `readme.txt` and `dec.cfg` files in `\WINCE700\SUPPORT\APP\VPU\DECTEST`.

28.7.2.3 Building and Running the Encoding Demo Application

To build and run the encoding demo application:

1. Click **Build** > **Open Release Directory in Build Window** on Windows Embedded Compact 7 to open the command prompt.
2. Run command **set wincerel=1** in command prompt window.
3. Change the current path to `\WINCE700\SUPPORT\APP\VPU\ENCTEST`
4. Build the application with **build -c** command.
5. Run the VPU application from the Windows Embedded Compact 7 Target Control with the command `s enctest \release\enc.cfg` (if `enc.cfg` file is copied to `\release` directory). Make sure the parameters set in the enc.cfg file are correct for the bitstream and hardware display. For detailed information, refer to the `readme.txt` and `enc.cfg` files in `\WINCE700\SUPPORT\APP\VPU\ENCTEST`.

Chapter 29
WLAN Driver

The WLAN driver is used to drive the AR6102 and AR6003 modules to implement Wi-Fi functionality. The WLAN module exchanges data with the i.MX device.

NOTE

for AR6003, there are two kind of wifi adapter, a SDIO card and a miniPCIe card.

29.1 WLAN Driver Summary

WLAN driver is provided in binary form instead of source codes. Table 29-1 provides a summary of the source code location, library dependencies, and other BSP information.

Table 29-1. WLAN Client Driver Summary

Driver Attribute	Definition
Target Platform	iMX53_ARD
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\SDIO\WIFI\AR6102 ..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V3\SDIO\WIFI\AR6003
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\SDIO\WiFi\AR6102 ..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\SDIO\WiFi\AR6003
Import Library	N/A
Driver DLL	For AR6003: AR6003_NDIS_SDIO.dll For AR6102: AR6102_NDIS_SDIO.dll, athsrvc.dll
Catalog Item	Third Party > BSP -> Freescale <*Target Platform*>: ARMV7 > Device Drivers > WiFi > Atheros AR6003 (SDIO) Driver Third Party > BSP -> Freescale <*Target Platform*>: ARMV7 > Device Drivers > WiFi > Atheros AR6102 (SDIO) Driver
SYSGEN Dependency	SYSGEN_ETH_80211_NWIFI SYSGEN_EAP SYSGEN_AUTH_NTLM SYSGEN_AUTH_SCHANNEL
BSP Environment Variable	BSP_AR6003_SDIO = 1 BSP_AR6102_SDIO = 1

The Recommended Catalog Items listed in Table 29-1 should be included in the OS design in order to provide Wi-Fi functionality.

29.2 Supported Functionality

The Wi-Fi driver provides the following software and hardware support:

Drives wifi module in AR6102:

1. Supports scanning and connection to 802.11b/g AP
2. Supports WPA, WPA2, WEP, WAPI

Drives wifi module in AR6003:

1. Supports scanning and connection to 802.11a/b/g/n AP
2. Supports WPA, WPA2, WEP, WAPI

On Windows Embedded Compact 7, it supports:

1. 802.11 authentication
2. 802.1x authentication
3. Automatic configuration
4. Native 802.11
5. Wi-Fi Protected Access
6. Extensible Authentication Protocol
7. Wired Equivalent Privacy
8. Roaming

29.3 Hardware Operation

The Wi-Fi client driver exchanges data and commands between the SD stack and the Wi-Fi hardware through SDIO port, so does the miniPCIe card.

29.3.1 Conflicts with Other Peripherals

No Conflicts with other peripherals. But these wifi adapters will conflict with each other, which is caused by the client driver provided by vendor. So make sure only one type of supported wifi adapters is enabled on the system.

29.4 Software Operation

29.4.1 Wi-Fi Registry Setting

The following registry keys are required to properly load and configure WLAN driver

For AR6003:

```
[HKEY_LOCAL_MACHINE\Comm\AR6K_SD]
```

```
    "DisplayName"="AR6003 WLAN Adapter SD"
    "Group"="NDIS"
    "ImagePath"="AR6003_NDIS_SDIO.dll"

[HKEY_LOCAL_MACHINE\Comm\AR6K_SD\Linkage]
    "Route"=multi_sz:"AR6K_SD1"

[HKEY_LOCAL_MACHINE\Comm\AR6K_SD1]
    "DisplayName"="AR6003 WLAN Adapter SD"
    "Group"="NDIS"
    "ImagePath"="AR6003_NDIS_SDIO.dll"
    "Wireless"=dword:1

[HKEY_LOCAL_MACHINE\Comm\AR6K_SD1\Parms]
    "*PhysicalMediaType"=dword:00000009
    "*MediaType"=dword:00000010
    "*IfType"=dword:00000047
    "binRoot"="\\Release"
    "clkFreq"=dword:18CBA80
    "nodeAge"=dword:1D4C0
    "enableUARTprint"=dword:1
    "tcmd"=dword:0
    "bkScanEnable"=dword:1
    "powerSaveMode"=dword:2

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-0271-CARDID-0200-FUNC-1]
    "Dll"="AR6003_NDIS_SDIO.dll"
    "Prefix"="DRG"

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-0271-CARDID-0201-FUNC-1]
    "Dll"="AR6003_NDIS_SDIO.dll"
    "Prefix"="DRG"

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-0271-CARDID-0300-FUNC-1]
    "Dll"="AR6003_NDIS_SDIO.dll"
    "Prefix"="DRG"

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-0271-CARDID-0301-FUNC-1]
    "Dll"="AR6003_NDIS_SDIO.dll"
    "Prefix"="DRG"
```

For AR6102:

```
[HKEY_LOCAL_MACHINE\Comm\AR6K_SD]
    "DisplayName"="AR6000 WLAN Adapter SD"
    "Group"="NDIS"
    "ImagePath"="ar6k_ndis_sdio.dll"
    "Wireless"=dword:1

[HKEY_LOCAL_MACHINE\Comm\AR6K_SD\Linkage]
    "Route"=multi_sz:"AR6K_SD1"

[HKEY_LOCAL_MACHINE\Comm\AR6K_SD1]
    "DisplayName"="AR6000 WLAN Adapter SD"
    "Group"="NDIS"
    "ImagePath"="ar6k_ndis_sdio.dll"
    "Wireless"=dword:1
```

```
[HKEY_LOCAL_MACHINE\Comm\AR6K_SD1\Parms]
    "BtCoexAntConfig"=dword:0
    "eepromFile"="calData_15dBm.bin"

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-0271-CARDID-0108-FUNC-1]
    "Dll"="ar6k_ndis_sdio.dll"
    "Prefix"="DRG"

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-0271-CARDID-0109-FUNC-1]
    "Dll"="ar6k_ndis_sdio.dll"
    "Prefix"="DRG"

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-0271-CARDID-010a-FUNC-1]
    "Dll"="ar6k_ndis_sdio.dll"
    "Prefix"="DRG"

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-0271-CARDID-010b-FUNC-1]
    "Dll"="ar6k_ndis_sdio.dll"
    "Prefix"="DRG"

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-0271-CARDID-0201-FUNC-1]
    "Dll"="ar6k_ndis_sdio.dll"
    "Prefix"="DRG"

[HKEY_LOCAL_MACHINE\Drivers\SDCARD\ClientDrivers\Custom\MANF-0271-CARDID-0200-FUNC-1]
    "Dll"="ar6k_ndis_sdio.dll"
    "Prefix"="DRG"

[HKEY_LOCAL_MACHINE\Services\ATHSRVC]
   "FriendlyName"="AthSrvc"
   "Dll"="ATHSRVC.Dll"
   "Order"=dword:10
   "Keep"=dword:1
   "Prefix"="ATH"
   "Index"=dword:0
   "appId"=dword:1
```

29.5 Unit Test

WLAN test includes manual WLAN connection without protection.

29.5.1 Running CTK Test: WiFi Authentication Tests

The Wi-Fi Authentication tests run a variety of authentication and encryption methods to validate Wi-Fi functionality for a device, as shown in the table that follows.

Table 29-2. Authentication methods

Authentication method	Description
Opened	All associations are accepted.
Shared	All associations are accepted, but the clinet must use WEP encryption.
WPA	Wi-Fi Protected Access. Requires EAP authentication.
WPA-PSK	WPA with a pre-shared key (PSK).
WPA2	Wi-Fi Protected Access 2. Requires EAP authentication.
WPA2-PSK	WPA2 with PSK.

29.5.2 Test the WLAN Communication without Protection

This test covers the practical functionality of the Wireless LAN driver to connect to any public wireless network for internet access. For this test, it is required to have a test board and any wireless access point (maybe a wireless router) without any protection to the Internet access. The test is considered passed if the user can access http://www.google.com and view its contents. To run the test:

1. Turn on the board
2. If Wireless driver can be successfully loaded, a dialog window listing the available wireless networks will be shown.
3. Select the wireless network without protection that you can use to navigate through the Internet
4. Open Internet Explorer from the desktop icon
5. Navigate through the Internet to http://www.google.com