i.MX25 PDK Windows Embedded CE 6.0

Reference Manual

Document Number: 924-76356
Rev. 2009.12
01/20010

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

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

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

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

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

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

Contents

About This Book

Chapter 1
Introduction

Chapter 2
Analog-Digital Converter (ADC) Driver

Chapter 3

Audio Driver

Chapter 4
Backlight Driver

Chapter 5
Battery Driver

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

Chapter 7
Camera Driver

Chapter 8
Chip Support Package Driver Development Kit (CSPDDK)

Chapter 9
Configurable Serial Peripheral Interface (CSPI) Driver

Chapter 10
Controller Area Network (CAN) Driver

Chapter 11
Dynamic Voltage and Frequency Control (DVFC) Driver

Chapter 12
Enhanced Secure Digital Host Controller (eSDHC) Driver

Chapter 13
Enhanced Serial Audio Interface (ESAI) Driver

Chapter 14
General Purpose Timer (GPT) Driver

Chapter 15
Fast Ethernet Controller (FEC) Driver

Chapter 16
Inter-Integrated Circuit (I^2C) Driver

Chapter 17
Keypad Driver

Chapter 18
Liquid Crystal Display Controller (LCDC) Driver

Chapter 19
NAND Flash Driver

Chapter 20
Pulse Width Modulator (PWM) Driver

Chapter 21
Serial Driver

Chapter 22
Subscriber Identification Module (SIM) Driver

Chapter 23
Touch Panel Driver

Chapter 24
Universal Serial Bus (USB) OTG Driver

Chapter 25
USB Boot and KITL

About This Book

This reference manual describes the requirements, implementation details, and testing for each module included in the Freescale Board Support Package (BSP) for Windows Embedded CE 6.0.

Audience

This document is intended for device driver developers, application developers, and software test engineers who are planning to use the product and for anyone who wants to understand more about the product.

Suggested Reading

The Freescale manuals can be found at the Freescale Semiconductor, Inc. World Wide Web site at http://www.freescale.com. These manuals may 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.MX25 Applications Processor IC Reference Manual
- i.MX25 3-Stack Release Notes for Windows Embedded CE 6.0
- i.MX25 3-Stack User’s Guide for Windows Embedded CE 6.0
- Microsoft Platform Builder for Windows Embedded CE 6.0 Help

Conventions

This document uses the following notational conventions:

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

Definitions, Acronyms, and Abbreviations

The following list defines the abbreviations used in this document.

ADS	application development system
API	application programming interface
BSP	board support package
CSP	chip support package
CSPI	configurable serial peripheral interface
D3DM	Direct 3D Mobile

DHCP	dynamic host configuration protocol
DPTC	dynamic power and temperature control
DVFC	dynamic voltage and frequency control
DVFS	dynamic voltage and frequency scaling
EBOOT	Ethernet bootloader
EVB	platform evaluation board
FAL	flash abstraction layer
FIR	fast infrared
FMD	flash media driver
GDI	graphics display interface
GPT	general purpose timer
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
NLED	Notification Light Emitting Diode
OAL	OEM adaptation layer
OEM	original equipment manufacturer
OS	operating system
OTG	on-the-go
PMIC	power management IC
PQOAL	production quality OEM adaptation layer
PWM	pulse-width modulator
SD	secure digital cards
SDC	synchronous display controller
SDHC	secure digital host controller
SDIO	secure digital I/O and combo cards
SDRAM	synchronous dynamic random access memory
SDK	software development kit
SIM	subscriber identification module
SIR	slow infrared

SOC	system on a chip
UART	universal asynchronous receiver transmitter
USB	universal serial bus

Chapter 1
Introduction

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

- i.MX25 3-Stack Development System

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

NOTE

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

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

1.1 Getting Started

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

1.2 Windows Embedded CE 6.0 Architecture

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

Welcome to Windows Embedded CE 6.0 > Windows Embedded CE Architecture

Chapter 2
Analog-Digital Converter (ADC) Driver

The Analog Digital Converter is a multipurpose module used to measure the voltage applied to dedicated input pins. Some of the input pins can be used to interface a resistive touchscreen, while other pins can be used for general purpose inputs. The ADC controller is not used directly by the software. It is controlled by the Touchscreen Controller (TSC).

2.1 ADC Driver Summary

The ADC driver can be used to measure the voltage of the General Purpose ADC pins and to interface with a touchscreen. Thus, only one of the driver interfaces is used by the touchscreen driver. The ADC driver interacts with the TSC to drive the ADC. Table 2-1 provides a summary of source code location, library dependencies and other BSP information.

Table 2-1. ADC Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	.PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\ADC
SOC Specific Path	..PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ADC
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\ADC
Driver DLL	adc.dll
SDK Library	adcsdk.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > ADC > Analog Digital Converter
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_ADC=1

2.2 Supported Functionality

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

1. Configures the Touchscreen conversion queue and items
2. Retrieves of the Touchscreen samples
3. Configures the general conversion queue and items
4. Retrieves the general purpose samples

2.3 Hardware Operation

Refer to the chapter on the TSC in the *i.MX25 Multimedia Applications Processor Reference Manual* for detailed hardware operation and programming information.

2.3.1 Conflicts with Other Peripherals and Catalog Items

Because the ADC inputs are not multiplexed with other functions, the ADC/TSC module does non conflict with other peripherals.

2.4 Software Operation

If the touchscreen component is used, the ADC driver should not use the Touchscreen conversion queue and items for general purpose operations.

2.4.1 ADC Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ADC]
    "Dll" = "adc.dll"
    "Prefix" = "ADC"
    "Index" = dword:1
    "Order" = dword:0
```

2.4.2 Interfacing with the ADC Driver

2.4.2.1 Stream Interface

The ADC driver is a stream interface driver, and is thus accessed through the file system APIs. The functional part of the driver is implemented by IOCTLs as shown in Table 2-2.

Table 2-2. Stream Interface

IOCTL Code	Description
IOCTL_CFG_ITEM	Configuration of the conversion item. The item configuration is described in the TSC section of the i.MX25Reference Manual.
IOCTL_CFG_QUEUE	Configuration of the conversion queue. The item configuration is described in the TSC section of the i.MX25Reference Manual.
IOCTL_START_ACQUIRE_SNGL	Starts the conversion
IOCTL_STOP_ACQUIRE_SNGL	Stops the conversion, is not necessary if the conversion is a single shot conversion
IOCTL_GET_DATA_AND_CLEAR_QUEUE_STATUS	Used to retrieve the data resulting of the conversion and clear the status of the queue
IOCTL_WAKEUP_SOURCE	Used to tell the driver if the Touchscreen is a wakeup source or not. This is necessary because if the Touchscreen is a wakeup source, then the TSC should not be gated off.

2.4.2.2 Using the SDK

The ADC driver includes a wrapper library that simplifies its use. This library is the ADC SDK and is described in Section 2.7, “ADC SDK API Reference”.

2.4.2.3 DMA Support

The ADC driver currently does not support DMA.

2.5 Power Management

2.5.1 ADC_PowerUp

This function is not implemented for the ADC driver.

2.5.2 ADC_PowerDown

This function is not implemented for the ADC driver.

2.5.3 IOCTL_POWER_CAPABILITES

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

2.5.4 IOCTL_POWER_SET

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

2.5.5 IOCTL_POWER_GET

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

2.6 Unit Test

The ADC CETK provided with the BSP verifies the ADC driver functionality.

2.6.1 Unit Test Hardware

Table 2-3 lists the required hardware to run the unit tests.

Table 2-3. Hardware Requirements

Requirement	Description
3-Stack Platform System	On the Personality board, R1288 is a variable resistor that can be used to adjust the voltage input of the pin INAUX0

2.6.2 Unit Test Software

Table 2-4 lists the required software to run the unit tests.

Table 2-4. 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
ADCTEST.dll	Test .dll file

2.6.3 Building the Unit Tests

To build the ADC 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 ADC Tests directory: `\WINCE600\SUPPORT\TEST\ADC`
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** command to build ADC test.

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

2.6.4 Running the Unit Tests

The command line for running the ADC tests is `tux -o -d adctest`. The ADC tests do not contain any test specific command line options. Table 2-5 describes the test cases contained in the ADC tests.

Table 2-5. ADC Test Cases

Test Case	Description
30001: TST_ADCConfigTest	Attempt to configure and make ten conversion/acquisition using the ADC driver (without using the SDK)
30002:TST_ADCSDKTest	Attempt to configure and make one conversion/acquisition using the ADC driver (using the SDK)

2.7 ADC SDK API Reference

2.7.1 ADC SDK Functions

2.7.1.1 AdcInit

This function initializes the SDK internal data (mutex for example) and opens the ADC driver.

```
BOOL AdcInit(
        void
);
```

Parameters	This function accepts no parameters
Return Values	TRUE indicates success
	FALSE indicates failure

2.7.1.2 AdcDeinit

This function is used to de-initializes the SDK: it closes the ADC driver and the mutex.

```
void AdcDeinit(
         void
);
```

Parameters	This function accepts no parameters
Return Values	This function returns no value

2.7.1.3 AdcConfigureChannel

This function configures a channel with the given settings.

```
BOOL AdcConfigureChannel(
        CHAN_ID id,
        DWORD settlingTime,
        POS_REF pRef,
        DWORD numSamples
);
```

Parameters

id	[in] Identifier of the channel to configure: 0 is for INAUX0, 1 is for INAUX 1, and so on
settlingTime	[in] Settling time used for the conversion (see the chapter on the TSC in the *i.MX25 Multimedia Applications Processor Reference Manual* for more details about the settling time)
pRef	[in] Specifies the reference that must be used for the conversion: could be either EXTREF (for the external reference) or INTREF (for the internal reference)
numSamples	[in] Number of samples that must be converted, between 1 and 16
Return Values	TRUE on success and FALSE indicates a failure

2.7.1.4 AdcGetSamples

The function starts a conversion, gets the data and stops the conversion.

```
BOOL AdcGetSamples(
        CHAN_ID id,
        UINT16* pBuf
        DWORD nbSamples
);
```

Parameters

id	[in] Identifier of the channel to configure: 0 is for INAUX0, 1 is for INAUX 1, and so on
pBuf	[Out] Points to a buffer where the SDK stores the acquired data
nbsamples	[in] Number of samples excepted after the conversion is complete
Return Values	TRUE on success and FALSE indicates a failure
Remarks	This function starts a single-shot conversion and waits for its completion. nbSamples must be same value as the one used in AdcConfigureChannel

Chapter 3
Audio Driver

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

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

3.1 Audio Driver Summary

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

Table 3-1. Audio Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\WAVEDEV2
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\WAVEDEV2
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\WAVEDEV2\SGTL5000
Driver DLL	wavedev2_sgtl5000
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX25-3DS:ARMV4I > Device Drivers > Audio > SGTL5000 Stereo Audio
SYSGEN Dependency	SYSGEN_AUDIO
BSP Environment Variables	BSP_NOAUDIO=

NOTE

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

3.2 Supported Functionality

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

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

3.3 Hardware Operation

3.3.1 Audio Hardware Design

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

3.3.1.1 i.MX25 3-Stack Audio Hardware Design

Since the Synchronous Serial Interface (SSI) is a full-duplex serial port, the i.MX25 uses instance two (SSI2) for both audio playback and recording. The external stereo codec SGTL5000 is connected to AUDMUX port 4 (external) and SSI2 is internally connected to AUDMUX port 2 (internal). Both ports are configured to operate in synchronous 4-wire mode.

The i.MX25 uses the I^2C bus interface to access the SGTL5000 control registers to configure the SGTL5000. The stereo codec SGTL5000 on the i.MX25 3-Stack System supports output to Headphone/Line Out/Speaker and input from Microphone/Line In.

For operation and programming, see the chapters in the *i.MX25 Multimedia Applications Processor Reference Manual* for the SSI, SDMA, AUDMUX, and IOMUX components, and see the *SGTL5000 Datasheet* for the external audio codec.

3.3.2 Audio Playback

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

- The audio driver is configured to use SSI2 for I^2S mode and a sampling rate of 44.1 KHz
  - The first two time slots transmit the left and right audio channel data words, respectively
  - Each audio data word is 16 bits long

  - SSI2 is also configured to operate in slave mode
  - The SSI2 transmitter watermark level is set to support SDMA transfers during audio playback
- The stereo codec is also configured for I^2S mode using a 44.1 KHz sample rate in master mode
- The Digital Audio MUX is configured to connect internal port 2 (which is assigned to SSI2) with one external port, which is used to communicate with the Stereo DAC. At the same time, the appropriate IOMUX pins are configured so that the Audio MUX external port signals can be routed off-chip to the Stereo Codec. The external port 4 is used to connect the Stereo Codec on the i.MX25 System.
- The SDMA channel supports 16-bit data transfers between the application memory buffers and the SSI2 TX FIFO0. The SSI2 TX FIFO0 is pre-filled with audio data at this point along with the DMA buffers.
- Finally, the SSI2 transmitter is enabled, which begins the transmission of the audio data stream.

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

- The SSI2 issues a new DMA request when the transmitter FIFO0 level reaches the empty watermark level. The SDMA controller then refills FIFO0 using data from the DMA buffers, until the DMA buffer is empty.
- An interrupt is generated when a DMA buffer is empty and this interrupt is handled by the audio driver. The audio driver refills the DMA buffer and returns it to the SDMA controller for processing.
- Due to the double-buffering scheme, the SDMA controller simply uses the other DMA buffer to continue refilling the SS12 transmitter FIFO0 while the previous DMA buffer is being refilled.

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

- When the entire audio stream is transmitted, there is no more data available to refill the empty DMA buffers. Therefore, the output DMA channel is disabled when both output DMA buffers are empty and there is no additional data available to refill them.
- The audio components that were used for playback are disabled to minimize power consumption. This step is done before disabling SSI2 to avoid any extraneous noise or “pop” that may be heard over the headphones.
- Finally, gate SSI2 is disabled and clocked if receiver is not working.

3.3.3 Audio Recording

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

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

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

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

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

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

3.3.4 Required SoC Peripherals

Table 3-2 shows the SoC hardware components required by the audio driver.

Table 3-2. Required SoC Peripherals

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

3.3.5 Conflicts with SoC Peripherals

No conflicts.

3.3.6 Conflicts with Board Peripherals

No conflicts.

3.3.7 Known Issues

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

3.4 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**

3.4.1 Audio Playback

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

3.4.2 Audio Recording

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

3.4.3 Audio Driver Compile-Time Configuration Options

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

NOTE

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

3.4.3.1 i.MX25 Audio Driver Configuration Options

Table 3-3 gives the compile-time configuration options of the i.MX25 stereo audio driver.

Table 3-3. i.MX25 Audio Driver Configuration Options (oemsettings.h)

Configuration Setting Name	Description
INCHANNELS	Defines the number of input/recording channels that are available. Can be set to either 1 or 2. Default is 2.

Table 3-3. i.MX25 Audio Driver Configuration Options (oemsettings.h)

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

3.4.4 DMA Support

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

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

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

The DMA data buffers can be allocated from either internal memory (which is provided by on-chip internal RAM) or external memory (which is provided by off-chip external DRAM). Table 3-4 describes the issues and considerations for the type of memory to use for the DMA data buffers.

Table 3-4. DMA Memory Allocation Issues and Considerations

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

Table 3-5 describes how to configure the build so that the audio driver allocates its DMA data buffers from either internal or external memory. The DMA buffer descriptors can also be allocated from either internal or external memory. However, the choice is made automatically through the use of the CSPDDK APIs, specifically DDKSdmaAllocChain(). See Chapter 8, “Chip Support Package Driver Development Kit (CSPDDK),” for additional information about the DDKSdmaAllocChain() API.

Table 3-5. Configuration Options for Internal/External Memory DMA Data Buffer Allocation

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

3.4.4.1 i.MX25 Audio DMA Buffer Use

The i.MX25 audio driver supports both playback and recording. The playback function uses internal memory for the DMA buffer, while the recording function allocates DMA buffers from external memory.

3.4.5 Power Management

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

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

3.4.5.1 PowerUp

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

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

3.4.5.2 PowerDown

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

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

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

3.4.5.3 IOCTL_POWER_SET

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

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

This registry entry is required for proper power management functionality.

3.4.6 Audio Driver Registry Settings

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

3.4.6.1 i.MX25 Audio Driver Registry Settings

The following registry keys are required for the Device Manager to properly load the i.MX25 audio device driver during the device normal boot process. These registry settings should typically not be modified. If these registry settings are missing or incorrectly defined, then the audio driver may not be loaded and all audio functions may be disabled.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio]
         "Prefix"="WAV"
         "Dll"="wavedev2_SGTL5000.dll"
         "Index"=dword:1
         "Order"=dword:10
         "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"    ; PMCLASS_GENERIC_DEVICE

; Override wave API load order to follow audio driver
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\WAPIMAN]
         "Order"=dword:11
```

3.5 Unit Test

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

3.5.1 Unit Test Hardware

Table 3-6 identifies the hardware needed to run the unit tests.

Table 3-6. Hardware Requirements

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

3.5.2 Unit Test Software

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

Table 3-7. Software Requirements

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

3.5.3 Building the Audio Driver CETK Tests

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

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

3.5.4 Running the Audio Driver CETK Tests

The command line for running the audio driver test is:

```
tux -o -d wavetest
```

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

```
tux -o -d wavetest -c “-e”
```

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

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

3.6 System Level Audio Driver Tests

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

3.6.1 Checking for a Boot-Time Musical Tune

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

3.6.2 Confirming Touchpanel Taps and Keypad Key Presses

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

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

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

3.6.4 Using the SDK Sample Audio Applications for Testing

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

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

3.7 Audio Driver API Reference

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

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

3.8 Audio Driver Troubleshooting Guide

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

3.8.1 Checking Build-Time Configuration Options

Compile- or link-time errors are probably due to incorrect or invalid configuration settings defined in `hwctxt.h` or `hwctxt.cpp`. See Section 3.4.3.1, “i.MX25 Audio Driver Configuration Options for information about the device driver build configuration options. Follow the build procedure documented in the Release Notes to compile and link the audio driver. Confirm that the required Platform Builder catalog items are included in the OS design. See Table 3-1 for a list of the required and recommended audio driver-related catalog items.

3.8.2 Media Player Application Not Found

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

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

3.8.3 Media Player Fails to Load and Play an Audio File

This problem is typically caused by failing to include the appropriate software codec that is required to handle the audio file format. See the list of recommended audio driver catalog items in Table 3-1 and make sure that support for the desired audio file format is included.

Chapter 4
Backlight Driver

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

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	iMX25-3DS -PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\BACKLIGHT
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\BACKLIKGHT
Driver DLL	backlight.dll
SDK Library	N\A
Catalog Item	Third Party > BSP > Freescale i.MX25 3DS PDK1_7: ARMV4I > Device Drivers > Backlight > Backlight LCDC
SYSGEN Dependency	SYSGEN_BATTERY=1
BSP Environment Variables	BSP_BACKLIGHT=1

4.2 Supported Functionality

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

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

4.3 Hardware Operation

The hardware consists of a PWM implemented by the LCDC. This PWM is usually dedicated to the contrast but on the i.MX25 3-Stack Platform System it is used to drive the backlight. It can be configured by writing the LPCCR register.

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

4.4.1.1 i.MX25 Backlight Driver Registry Setting

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

```
[HKEY_CURRENT_USER\ControlPanel\Backlight]
         "BattBacklightLevel"=dword:7F      ; Backlight level settings. 0xFF = Full On
         "ACBacklightLevel"=dword:7F        ; Backlight level settings. 0xFF = Full On
         "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
```

4.4.2 Power Management

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

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 tested by the application test.

4.5.1 Unit Test Hardware

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

Table 4-2. Hardware Requirements

Requirement	Description
CHUNGHWA CLAA057VA01CT VGA Panel	Display panel required for display of graphics data

4.5.2 Unit Test Software

Table 4-3 lists the required software to run the backlight application test.

Table 4-3. Software Requirements

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

4.5.3 Running the Backlight Application Test

Table 4-4 lists the backlight application test.

Table 4-4. Backlight Application Test

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

4.6 Backlight API Reference

The API for the backlight driver conforms to the stream interface and exposes the standard functions. Further information can be found at:

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

Chapter 5
Battery Driver

The battery driver module provides information about the battery level to the OS. As the platform is lacking hardware support for battery management (monitoring and charging), the battery driver is essentially a stub.

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	iMX25-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	N/A
SOC Specific 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 i.MX25-EVK PDK1_7:ARMV4I> Device Drivers >Battery > Fake Battery
SYSGEN Dependency	SYSGEN_BATTERY
BSP Environment Variables	BSP_NOBATTERY=

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 current i.MX25 3-Stack 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"="battdrvr.dll"
   "Flags"=dword:8                  ; DEVFLAGS_NAKEDENTRIES
   "IClass"="{DD176277-CD34-4980-91EE-67DBEF3D8913}"
   "BattFullLiftTime" = dword:8     ;Batt Spec defined: in unit of hr, here 8hr is assumed
   "BattFullCapacity"=dword:320     ;Batt Spec defined: in unit of mAh, here 800mAhr is assumed
   "BattMaxVoltage"=dword:1068      ;Batt Spec defined: in unit of mV, here 4200mV is assumed
   "BattMinVoltage"=dword:BB8       ;Batt Spec defined: in unit of mV, here 3000mV is assumed
   "BattPeukertNumber"=dword:73     ;Batt Spec defined, here 1.15 is assumed
   "BattChargeEff"=dword:50         ;Batt Spec defined, here 0.80 is assumed

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

5.4.2 Power Management

There is no additional power management implementation for battery driver.

5.5 Unit Test

5.6 Battery API Reference

The API for the battery driver conforms to the stream interface and exposes the standard functions. Further information can be found at:

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

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

Boot support from SD/MMC includes the following components:

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

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

NOTE

> XLDR and EBOOT only support boot from ESDHC1. Boot ROM supports booting from all ESDHC ports; therefore, XLDR and EBOOT can be extended to boot from other ports. SD/MMC boot requires a card that is at least 36 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)	iMX25-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	N/A
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\BOOTLOADER ..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\BOOT\FMD\SDMMC
Driver DLL	N/A
SDK Library	N/A
Catalog Item(s)	N/A
SYSGEN Dependency	N/A
BSP Environment Variable(s)	N/A

6.2 Supported Functionality

The boot support from SD/MMC includes:

1. Boot from low or high capacity SD/MMC card at least 36 Mbytes in size on ESDHC1
2. Storing bootloader and SD/MMC Xloader images to SD/MMC flash
3. Storing OS images to SD/MMC flash
4. Loading OS image from SD/MMC flash to RAM
5. File system on bootable SD/MMC card
6. Internal boot (BMOD = 00) from SD/MMC
7. eSD2.1 and eMMC 4.3 boot from boot partition if boot partition can be configured to be at least 36 Mbytes in size; otherwise, boot from user partition on these devices is supported

6.3 Hardware Operation

6.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts for eSDHC1 with other on-chip peripherals.

6.4 Software Operation

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

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

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

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

For preparing and downloading the SD/MMC bootloader and for usage of the SD/MMC bootloader, refer to the *BSP User'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 36 Mbytes for boot images (XLDR, EBOOT, NK) and the remaining portion of the card for the file system. The card must then be inserted into a PC to format the file system partition. Subsequently, it can be used as a boot device as well as to store and load user files once the OS has loaded. Refer to the *BSP User's Guide* for details.

eSD v2.1 and eMMC v4.3 both provide the capability of having more than one physical partition, thus eliminating the need to put an MBR on the device. Reading, writing, and erasing one partition has no effect on the other partitions. Starting with TO1.0, the ROM is able to boot from the boot partition on these devices. During boot, the ROM code selects the boot partition #1 on the eSD v2.1 device and either boot partition #1 or #2 on the eMMC v4.3 device (depending on which partition is enabled in the EXT_CSD register), and subsequently reads out the data that is flashed to the boot partition and executes it. EBOOT provides menu options to create and enable/disable boot partitions on both devices using the MMC and SD Utilities sub-menu. Refer to the *BSP User's Guide* for details.

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

6.4.1.1 i.MX25 Card Memory Layout

Figure 6-1 shows the card memory layout for the i.MX25.

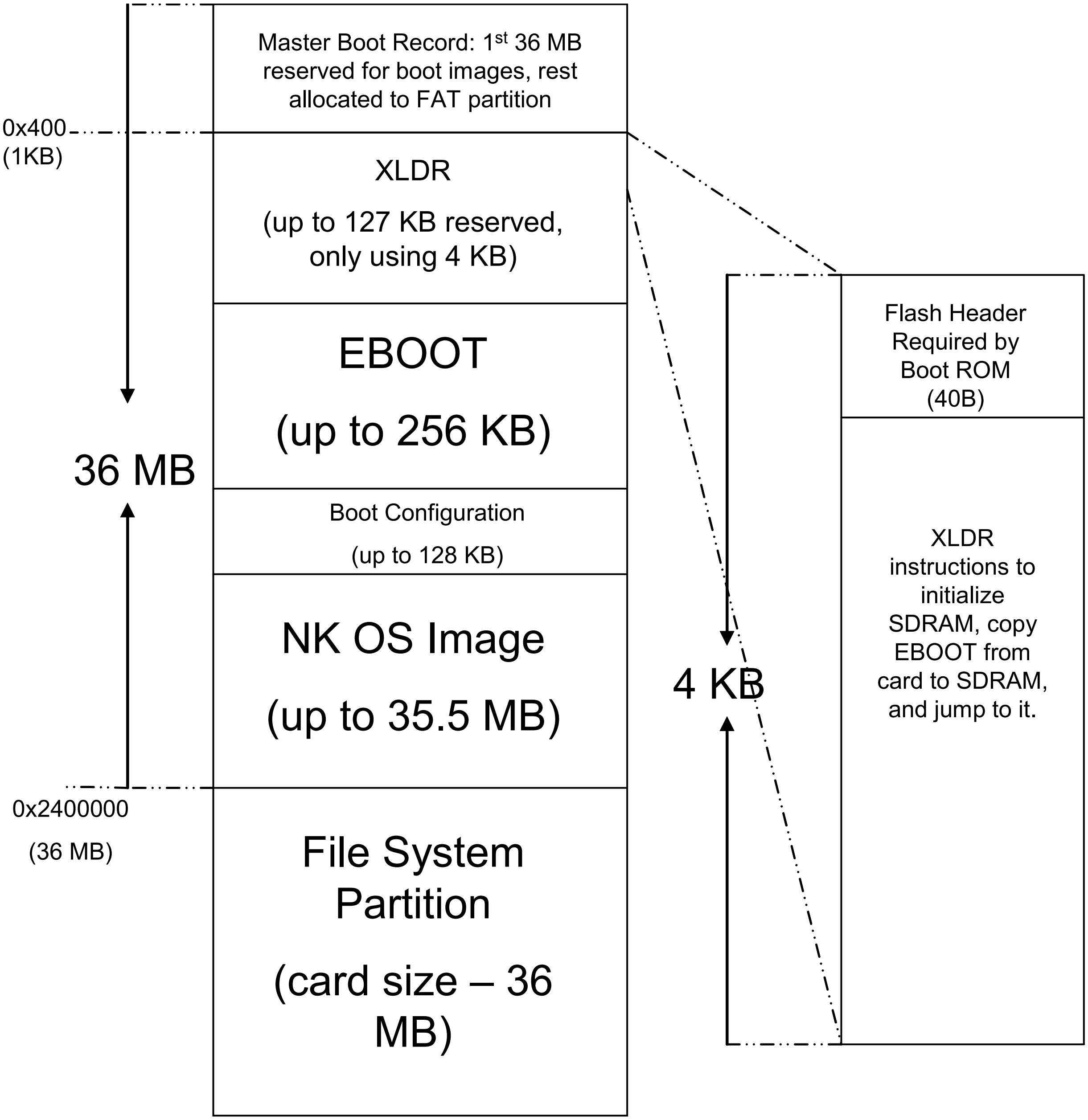
Master Boot Record: 1st 36 MB reserved for boot images, rest allocated to FAT partition
0x400
(1KB)
XLDR
(up to 127 KB reserved, only using 4 KB)
EBOOT
(up to 256 KB)
36 MB
Boot Configuration
(up to 128 KB)
NK OS Image
(up to 35.5 MB)
0x2400000
(36 MB)
File System Partition
(card size – 36 MB)
4 KB
Flash Header Required by Boot ROM (40B)
XLDR instructions to initialize SDRAM, copy EBOOT from card to SDRAM, and jump to it.

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 36 Mbytes of the card for boot images, and allocate the remaining portion to the file system. The XLDR is saved at 0x400 (1 Kbyte) offset, which is sector 2 in the card. The Boot ROM calculates the entry point of the XLDR from the flash header structure found in the XLDR.

The MBR is only required on cards that are older than eSD v2.1 and eMMC v4.3 because these newer devices can have multiple physical partitions. On these devices, the first 36 Mbytes shown above are flashed on a separate boot partition (without an MBR at sector 0), and the file system partition referenced above is another separate physical partition, which should only be active while OS is running.

Chapter 7
Camera Driver

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

7.1 Camera Driver Summary

The camera driver is based on the Windows CE Camera Device Driver Interface. This interface provides basic support for video and still image capture devices. The camera driver conforms to the architecture for Windows CE stream interface drivers. At the lower layer, the camera driver performs several tasks:

- Communicating with and configuring the camera device or other input device through the I^{2}C interface
- Interfacing with the Camera Sensor Interface (CSI) to perform the tasks on captured images or preview data

For i.MX25 BSP, this module is compatible with the Omnivision OV2640 camera sensor. 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	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	N/A
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\CAMERA
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\CAMERA
Driver DLL	camera.dll
SDK Library	N/A
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*>*:ARMV4I* > Device Drivers > Camera
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_CAMERA=1

7.2 Supported Functionality

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

1. Supports the Windows CE Camera Device Driver Interface
2. Supports Preview, Capture, and Still pins for camera application
3. Supports the OV2640 camera sensor for camera used for i.MX25.
4. Supports these resolutions “QQCIF, QQVGA, QCIF, QVGA, CIF, and VGA“ for 30fps Preview, Caputer and Still pin in i.MX25.
5. Supports these resolutions “QQCIF, QQVGA, QCIF, QVGA, CIF, VGA, 1024*800 “ for 15fps Preview, Caputer and Still pin in i.MX25.
6. Supports the resolution “1280*960“ for 7fps Preview, Capture and Still pin in i.MX25.
7. Supports these formats “RGB565“ for Preview pin in i.MX25.
8. Supports these formats “RGB565, UYVY, YUY2 and YV12“ for Capture and Still pin in i.MX25.
9. Only support “CSPROPERTY_BUFFER_DRIVER” mode.

7.3 Hardware Operation

The CSI is the only controller involved in the operation of the camera driver. The input device (camera sensor) captures external image data. The CSI controller retrieves image data from the sensor and transfers the data to a memory destination using two DMA channels embedded in the CSI controller. The i.MX25 processor does not include a preprocessing controller. For detailed operation and programming information, refer to the chapter on the CSI in the *i.MX25 Multimedia Applications Processor Reference Manual*.

7.3.1 Conflicts with Other Peripherals and Catalog Items

7.3.1.1 Conflicts with SoC Peripherals

All the pins of the CSI controller can be configured for alternate functionality (ESAI, SIM) using the i.MX25 IOMUX. The configuration is specified by BSP serial driver. Changing this configuration results in a conflict and prevents proper operation of the ESAI driver.

7.3.2 Conflicts with 3-Stack Peripherals

On i.MX25 3-Stack System, the pins used for communication between CSI and the external camera sensor are shared with the ESAI and the SIM1 controllers. Do not include the ESAI or SIM1 drivers while including the camera driver in the image. Also, the i.MX25 3-Stack System has an extra connector (J28) for connecting an Omnivision OV2640 sensor. Do not connect sensors on J28 and CN14 at the same time, which could lead to unexpected results and hardware damage.

7.4 Software Operation

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

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

7.4.1 Communicating with the Camera

Communication with the camera driver is accomplished through Camera APIs defined by Microsoft for Windows Embedded CE 6.0.

7.4.1.1 Using the Windows CE Video Camera Device Driver Interface

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

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

7.4.1.2 Using DirectShow for Video Capture

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

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

7.4.2 Registry Settings

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

7.4.2.1 i.MX25 Registry Settings

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

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Camera]
   "Prefix"="CAM"
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:1
   "CameraId"=dword:0     ; Default sensor is Omnivision OV2640
   "IClass"=multi_sz:
"{CB998A05-122C-4166-846A-933E4D7E3C86}","{A32942B7-920C-486b-B0E6-92A702A99B35}"

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

The CameraId registry key identifies the available camera sensor modules. The only value is 0 for the Omnivision OV2640 camera sensor.

7.5 Power Management

The camera driver consumes power primarily through the operation of the CSI module, which synchronizes and receives image data from the camera sensor. The CSI module is enabled when the camera is set to a running state. Support for transitioning to the Suspend and Resume states if provided through the IOCTL_POWER_SET IOCTL.

7.5.1 Power Up

This function is not implemented for the camera driver.

7.5.2 Power Down

This function is not implemented for the camera driver.

7.5.3 IOCTL_POWER_SET

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

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

7.6 Unit Test

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

The Camera CETK tests include:

- The Camera Driver Data Structure Verification Test - queries the driver for the various properties and formats, and verifies that the data structures returned are valid.
- The Camera Driver I/O Test - verifies the functionality of the preview and capture streams on the camera driver.

Additionally, for Windows Embedded CE 6.0, 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	OV2640 CMOS camera sensor

For i.MX25 BSP, the OV2640 camera sensor is used to run the Windows Embedded CE 6.0 Camera CETK test and the camera application.

7.6.2 Unit Test Software

7.6.2.1 CETK Test

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

Table 7-3. Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
CameraGraphTests.dll	Library containing the camera and directshow integration test cases
CamTestProperties.dll	Library containing the camera driver data structure verification test cases
CamIOTests.dll	Library containing the camera driver I/O test cases
CameraPerfTests.dll	Library containing the camera performance test cases
Cameragrabber.dll	Filter required by many command-line options to track and output information about media samples
camera.dll	Driver.dll file

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

7.6.2.2 Custom Camera Application Test

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

7.6.2.3 Camera Application Test

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

7.6.3 Building the Unit Tests

7.6.3.1 CETK Test

The camera tests come pre-built as part of the CETK so no steps are required to build these tests. These test files can be found with the other required CETK files in the following location:

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

7.6.3.2 Custom Camera Application Test

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

Build an OS image for the desired Camera configuration:

1. Add a new folder named `APP` under the folder ..\PLATFORM\<Target Platform>\SRC
2. Create an empty dirs file under the folder ..\PLATFORM\<Target Platform>\SRC\APP
3. Copy the folder of `CAMAPP` under the folder `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. The camapp.exe file is copied to the workspace release directory.

Complete the following steps to build the custom Camera application:

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

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

7.6.4 Running the Unit Tests

7.6.4.1 Running the Camera Unit Tests

7.6.4.1.1 Running the Camera CETK Test

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

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

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

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

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

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

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

```
tux –o –d CamIOTests.dll
```

Use this command line to run the Camera Performance test:

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

NOTE

The Camera Performance test requires the configuration file `capconfig.ini` which specifies what to test, by copying the file under the corresponding folder such as \release before testing from the following location:

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

For i.MX25 camera and directshow integration test, there will be some cases fail:(1)405 and 407 try to allocate 22 buffers the size of an image (10 CAPTURE + 10 PREVIEW + 2 STILL). The hardware is not able to provide enough memory when hitting high resolutions (@1280x960x16 => 1280*960*2*22 = 51,5 Mbytes);(2)508 start a PREVIEW with the first resolution available, then run a CAPTURE with all the resolutions available. This causes the driver to return an error as PREVIEW and CAPTURE should not run at different resolutions (not supported by hardware).

For i.MX25 camIOtest, 1003 will not pass with resolution 1280x960 at YV12 format because of lower frame rate. Resolution 1280x960 is only 7fps, and at YV12 format, there is a software arithmatic to change non-standard YV12 to standard YV12 format, so the frame rate is lower. Please note that all the other resolutions will pass 1003 successfully, but as there is no way to ban a specific resolution from a single test case, 1003 will end up failing.

7.6.4.1.2 Running the Custom Camera Application Test

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

7.7 Camera Driver API Reference

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

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

Chapter 8
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)

8.1 CSPDDK Driver Summary

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

Table 8-1. CSPDDK Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFROM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\CSPDDK
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\CSPDDK
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\CSPDDK
Driver DLL	cspddk.dll
SDK Library	N/A
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOCSPDDK=

8.2 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)

8.3 Hardware Operation

Refer to the *i.MX25 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

Refer to the *i.MX25 Multimedia Applications Processor Reference Manual* for possible conflicts.

8.3.1.2 Conflicts with Board Peripherals

No conflicts.

8.4 Software Operation

8.4.1 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.

8.4.2 Compile-Time Configuration Options

The CSPDDK exposes compile-time options for configuring the SDMA support. In some cases, these compilation variables are also leveraged by driver code to expose a central point of controlling SDMA functionality. Table 8-2 describes the available CSPDDK compile options.

Table 8-2. CSPDDK Compile Options

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

Table 8-2. CSPDDK Compile Options (continued)

IMAGE_WINCE_CSPDDK_RAM_OFFSET	image_cfg.h	Offset in bytes from the base of external RAM for the shared CSPDDK data structures.
IMAGE_WINCE_CSPDDK_RAM_SIZE	image_cfg.h	Size in bytes of the external RAM reserved for CSPDDK data structures. This size must correspond to the region reserved in config.bib.
IMAGE_WINCE_DDKSDMA_RAM_PA_START	image_cfg.h	Physical starting address in external RAM where the shared DDK_SDMA data structures are located. This starting address must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions.
IMAGE_WINCE_DDKSDMA_RAM_SIZE	image_cfg.h	Size in bytes of the external RAM reserved for DDK_SDMA data structures. This size must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions.
IMAGE_WINCE_DDKCLK_RAM_PA_START	image_cfg.h	Physical starting address in external RAM where the shared DDK_CLK data structures are located. This starting address must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions.
IMAGE_WINCE_DDKCLK_RAM_SIZE	image_cfg.h	Size in bytes of the external RAM reserved for DDK_CLK data structures. This size must fall within the region reserved by the IMAGE_WINCE_CSPDDK definitions.
BSP_SDMA_MC0PTR	bsp_cfg.h	Starting address for the shared SDMA data structures. Set to IMAGE_WINCE_SDMA_PA_START to use external RAM.
BSP_SDMA_CHNPRI_xxx	bsp_cfg.h	Assigns a SDMA channel priority to the respective peripheral. Refer to the individual driver chapters for more information on the specific priorities.
BSP_SDMA_SUPPORT_xxx	bsp_cfg.h	Boolean to specifies if SDMA-based transfers are enabled for each respective peripheral. Refer to the individual driver chapters for more information on the DMA support provided.

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

8.4.3 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.

8.4.4 Power Management

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

8.5 Unit Test

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

8.5.1 Unit Test Hardware

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

Table 8-3. Hardware Requirements

Requirement	Description
No additional hardware required	

8.5.2 Unit Test Software

Table 8-4 lists the required software to run the unit tests.

Table 8-4. 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

8.5.3 Building the Unit Tests

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

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

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

8.5.4 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 8-5 describes the test cases contained in the DDK_SDMA tests.

Table 8-5. 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.

8.6 CSPDDK DLL Reference

8.6.1 CSPDDK DLL System Clocking (DDK_CLK) Reference

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

8.6.1.1 DDK_CLK Enumerations

Table 8-6. 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_DVFC_SETPOINT	Frequency/voltage setpoints supported by the DVFC driver

8.6.1.2 DDK_CLK Functions

8.6.1.2.1 DDKClockSetGatingMode

This function sets the clock gating mode of the peripheral.

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

Parameters

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

8.6.1.2.2 DDKClockGetGatingMode

This function retrieves the clock gating mode of the peripheral.

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

Parameters

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

8.6.1.2.3 DDKClockGetFreq

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

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

Parameters

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

8.6.1.2.4 DDKClockConfigBaud

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

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

Parameters

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

8.6.1.2.5 DDKClockSetCKO

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

```
BOOL DDKClockSetCKO(
        BOOL bEnable,
        DDK_CLOCK_CKO_SRC src,
        UINT32 divider)
```

Parameters

bEnable	[in] Set to TRUE to enable CKO output. Set to FALSE to disable CKO output
src	[in] Selects the CKO source signal

divider	[in] Specifies the CKO, value of divider - 1
Return Values	Returns TRUE if successful, otherwise returns FALSE

8.6.1.2.6 DDKClockSetpointRequest

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

```
BOOL DDKClockSetpointRequest(
        DDK_DVFC_SETPOINT setpoint,
        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

8.6.1.2.7 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

8.6.1.3 DDK_CLK Examples

Example 8-1. 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 8-2. CSPDDK Clock Rate Query

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

UINT32 freq;
```

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

8.6.2 CSPDDK DLL GPIO (DDK_GPIO) Reference

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

8.6.2.1 DDK_GPIO Enumerations

Table 8-7. 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

8.6.2.2 DDK_GPIO Functions

8.6.2.2.1 DDKGpioSetConfig

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

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

Parameters

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

8.6.2.2.2 DDKGpioWriteData

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

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

Parameters

port	[in] GPIO module instance

portMask	[in] Bit mask for data port pins to be written
data	[in] Data to be written
Return Values	Returns TRUE if successful, otherwise returns FALSE

8.6.2.2.3 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

8.6.2.2.4 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

8.6.2.2.5 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

8.6.2.2.6 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

8.6.2.2.7 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

8.6.2.2.8 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

8.6.2.3 DDK_GPIO Example

Example 8-3. 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);
```

8.6.3 CSPDDK DLL IOMUX (DDK_IOMUX) Reference

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

8.6.3.1 DDK_IOMUX Enumerations

Table 8-8. 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.
DDK_IOMUX_PAD_OPENDRAIN	Open drain for a pad; if no ODE bit for a PAD, the DDK_IOMUX_PAD_OPENDRAIN_NULL should be set
DDK_IOMUX_PAD_HYSTERESIS	Hysteresis mode for a pad; if no HYS bit for a PAD, the DDK_IOMUX_PAD_HYSTERESIS_NULL should be set
DDK_IOMUX_PAD_VOLTAGE	Driver voltage for a pad, either 1.8 V or 3.3 V
DDK_IOMUX_PAD_PULL	Pull-up/pull-down/keeper configuration for a pad
DDK_IOMUX_SELECT_INPUT	Functional pad name to be selected and involved in Daisy Chain

8.6.3.2 DDK_IOMUX Functions

8.6.3.2.1 DDKIomuxSetPinMux

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

```
BOOL DDKIomuxSetPinMux(
         DDK_IOMUX_PIN pin,
         DDK_IOMUX_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

8.6.3.2.2 DDKIomuxGetPinMux

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

```
BOOL DDKIomuxGetPinMux(
         DDK_IOMUX_PIN pin,
         DDK_IOMUX_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

8.6.3.2.3 DDKIomuxSetPadConfig

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

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

Parameters

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

8.6.3.2.4 DDKIomuxGetPadConfig

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

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

Parameters

pad	[in] Functional pad name used to select the pad that is configured
pSlew	[out] Slew rate configuration
pDrive	[out] Drive strength configuration
pOpenDrain	[out] Open drain configuration
pPull	[out] Pull-up/pull-down/keeper configuration
pHysteresis	[out] Hysteresis configuration
pVoltage	[out] Drive voltage configuration
Return Values	Returns TRUE if successful, otherwise returns FALSE.

8.6.3.2.5 DDKIomuxSelectInput

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

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

Parameters

port	[in] Port to select input
daisy	[in] Data to be written
Return Values	Returns TRUE if successful, otherwise returns FALSE

8.6.3.3 DDK_IOMUX Examples

Example 8-4. 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 8-5. CSPDDK IOMUX Pad Configuration

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

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

8.6.4 CSPDDK DLL SDMA (DDK_SDMA) Reference

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

8.6.4.1 DDK_SDMA Enumerations

Table 8-9. 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

8.6.4.2 DDK_SDMA Functions

8.6.4.2.1 DDKSdmaOpenChan

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

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

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

8.6.4.2.2 DDKSdmaUpdateSharedChan

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

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

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan
dmaReq	[in] Specifies the DMA request that is bound to a virtual channel
Return Values	Returns TRUE if successful, otherwise returns FALSE

8.6.4.2.3 DDKSdmaCloseChan

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

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

Parameters

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

8.6.4.2.4 DDKSdmaAllocChain

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

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

Parameters

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

8.6.4.2.5 DDKSdmaFreeChain

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

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

Parameters

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

8.6.4.2.6 DDKSdmaSetBufDesc

This function configures a buffer descriptor for a DMA transfer.

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

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan.
index	[in] Index of buffer descriptor within the chain to be configured.
modeFlags	[in] Specifies the buffer descriptor mode word flags that control the continue, wrap, and interrupt settings
memAddr1PA	[in] For memory-to-memory transfers, this parameter specifies the physical memory source address for the transfer. For memory-to-peripheral transfers, this parameter specifies the physical memory source address for the transfer. For

	peripheral-to-memory transfers, this parameter specifies the physical memory destination address for the transfer
memAddr2PA	[in] Used only for memory-to-memory transfers to specify the physical memory destination address for the transfer. Ignored for memory-to-peripheral and peripheral-to-memory transfers
dataWidth	[in] Used only for memory-to-peripheral and peripheral-to-memory transfers to specify the width of the data for the peripheral transfer. Ignored for memory-to-memory transfers
numBytes	[in] Virtual channel returned by DDKSdmaOpenChan
Return Values	Returns TRUE if successful, otherwise returns FALSE

8.6.4.2.7 DDKSdmaGetBufDescStatus

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

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

Parameters

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

8.6.4.2.8 DDKSdmaGetChainStatus

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

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

Parameters

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

8.6.4.2.9 DDKSdmaClearBufDescStatus

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

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

Parameters

chan	[in] Virtual channel returned by DDKSdmaOpenChan
index	[in] Index of buffer descriptor within the chain

Return Values	Returns TRUE if successful, otherwise returns FALSE

8.6.4.2.10 DDKSdmaClearChainStatus

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

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

Parameters

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

8.6.4.2.11 DDKSdmaInitChain

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

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

Parameters

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

8.6.4.2.12 DDKSdmaStartChan

This function starts the specified channel.

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

Parameters

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

8.6.4.2.13 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 9
Configurable Serial Peripheral Interface (CSPI) Driver

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

9.1 CSPI Driver Summary

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

Table 9-1. CSPI Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\CSPIBUSV2
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\CSPIBUS
Platform Driver Path	..\PLATFORM\<*Target Platform*>\DRIVERS\CSPIBUS
Import Library	cspisdk.lib
Driver DLL	cspi.dll
Catalog Item	Third Party > BSP > Freescale <*TGTPLAT*> > Device Drivers > CSPI Bus
SYSGEN Dependency	N/A

BSP Environment Variables	BSP_CSPIBUS1=1
	BSP_CSPIBUS2=1
	BSP_CSPIBUS3=1

9.2 Supported Functionality

The CSPI driver supports the following features:

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

9.2.1 Conflicts with Other Peripherals and Catalog Items

9.2.1.1 Conflicts with SoC Peripherals

The i.MX25 platform contains three CSPI modules, but only the CSPI1 module is used on the i.MX25 3-Stack Platform System. No pad is provided to CSPI2 and CSPI3 modules.

9.2.2 Conflicts with 3-Stack Peripherals

The CSPI1 is also used by the OAL to communicate with an external debug device. The transfer is not handled by the driver itself. Instead the driver calls a dedicated Kernel IOCTL: `IOCTL_HAL_SHARED_CSPI_TRANSFER`. This IOCTL does not use DMA or interrupts; therefore, the values related to this in the CSPI_XCH_PKT_T structure may be set to NULL.

9.3 Software Operation

9.3.1 Registry Settings

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

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

IF BSP_CSPIBUS2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\CSPI1]
   "Prefix"="SPI"
   "Dll"="cspi.dll"
   "Index"=dword:2
ENDIF ; BSP_CSPIBUS2

IF BSP_CSPIBUS3
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\CSPI1]
   "Prefix"="SPI"
   "Dll"="cspi.dll"
   "Index"=dword:3
ENDIF ; BSP_CSPIBUS3
```

9.3.2 Communicating with the CSPI

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

9.3.3 Creating a Handle to the CSPI

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

To open a handle to the CSPI:

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

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

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

9.3.4 Data Transfer Operations

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

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

To perform an CSPI transfer:

1. Create a CSPI_XCH_PKT object and initialize the fields of the packet as follows:
   a) Initialize a CSPI_BUSCONFIG datum to specify the bus parameters as CHANNEL SELECT, DATA RATE, BURST LENGTH, SSPOL, SSCTL, POL, PHA, DRCTL, and specify the method parameters for use/not use DMA, use/not use POLLING.
   b) Set the *pTxBuf* field to the user buffer which the data is written.
   c) Set the *pRxBuf* field to the user buffer which receives data, if does not care of the Rx data, set the field to NULL.
   d) Set the *xchCnt* field, for the 1-8 bit XCH, the *xchCnt = bytes*, for the 9-16 bit XCH, the *xchCnt = words*, for the 17-32 bit XCH, the *xchCnt = dwords.*

   e) If give the *xchEvent* parameter, also give the *xchEventlength* and include the tail Zero character. Otherwise, set *xchEvent* to NULL, and *xchEventlength* to 0. When use *xchEvent* the XCH data is queued inside driver.

2. Set the *hDevice* parameter to the previously acquired CSPI port handle.
3. Set the *dwIoControlCode* to the SPI_IOCTL_EXCHANGE IOCTL code.
4. Set the *lpInBuffer* to point to the CSPI_XCH_PKT object created in step 1. Set *nInBufferSize* to the size of that packet object.
5. Set *lpOutBuffer*, *lpBytesReturned*, and *lpOverlapped* to NULL. Set *nOutBufferSize* to 0.

If the CSPI is also used by the OAL to communicate with an external debug device, then the transfer is not handled by the driver itself. Instead the driver calls a dedicated Kernel IOCTL: `IOCTL_HAL_SHARED_CSPI_TRANSFER`. This IOCTL does not use DMA or interrupts, therefore the values related to this may be set to NULL.

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

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

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

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

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

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

9.3.5 Closing the Handle to the CSPI

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

9.3.6 Power Management

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

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

9.3.6.1 PowerUp

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

9.3.6.2 PowerDown

This function is not implemented for the CSPI driver.

9.3.6.3 IOCTL_POWER_SET

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

9.4 Restrictions

The following restrictions apply to the DMA XCH:

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

9.5 Unit Test

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

9.5.1 Building the Unit Tests

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

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

   A DOS prompt is displayed.
2. Change to the CSPI Test directory: `\WINCE600\SUPPORT`_PDK1_7`\TEST\CSPI`
3. Enter **set WINCEREL=1** on the command prompt and press return.

   This copies the EXE to the flat release directory.
4. Input **build -c** to build CSPI test.

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

9.6 CSPI Driver API Reference

9.6.1 CSPI Driver IOCTLS

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

9.6.1.1 CSPI_IOCTL_EXCHANGE

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

Parameters

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

9.6.1.2 CSPI_IOCTL_ENABLE_LOOPBACK

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

9.6.1.3 CSPI_IOCTL_DISABLE_LOOPBACK

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

9.6.2 CSPI Driver SDK Wrapper

9.6.2.1 CSPIOpenHandle

This function retrieves the CSPI device handle.

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

Parameters

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

9.6.2.2 CSPICloseHandle

This function closes a handle of the CSPI stream driver.

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

Parameters

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

9.6.2.3 CSPIEnableLoopback

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

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

Parameters

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

9.6.2.4 CSPIExchange

This function performs XCH operations.

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

Parameters

hDev	The CSPI device handle retrieved from CreateFile()
pCspiXchPkt	[in] Pointer to XCH packet with bus configuration parameters

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

9.6.3 CSPI Driver Structures

9.6.3.1 CSPI_BUSCONFIG_T

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

```
// CSPI bus configuration
typedef struct
{
    UINT8         chipselect;
    UINT32        freq;
    UINT8         bitcount;
    BOOL          sspol;
    BOOL          ssctl;
    BOOL          pol;
    BOOL          pha;
    UINT8         drctl;
    BOOL          usedma;
    BOOL          usepolling;
} CSPI_BUSCONFIG_T, *PCSPI_BUSCONFIG_T;
```

Table 9-2. CSPI_BUSCONFIG_T Structure Members

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

9.6.3.2 CSPI_XCH_PKT_T

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

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

Table 9-3. CSPI_XCH_PKT_T Structure Members

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

Chapter 10
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.

10.1 CAN Driver Summary

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

Table 10-1. CAN Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\CANBUS
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\CANBUS
Driver DLL	can.dll
SDK Library	cansdk.lib
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*> > Device Drivers > CAN Bus
SYSGEN Dependency	N/A

BSP Environment Variables	BSP_CANBUS1=1
	BSP_CANBUS2=1

10.2 Supported Functionality

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

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

10.3 Hardware Operation

Refer to the chapter on CAN in the *i.MX25 Multimedia Applications Processor Reference Manual* for detailed operation and programming information.

10.3.1 Conflicts with Other Peripherals and Catalog Items

10.3.1.1 Conflicts with SoC Peripherals

The CAN1 and CAN2 controller shares the PIO with the FEC controller and conflicts with the GPIOA pin. The driver configures which controller (FEC or GPIO) the CAN conflicts with. The default conflict value is GPIO for the CAN1 and CAN2 controller.

10.3.1.2 Conflicts with 3-Stack Peripherals

No conflicts.

10.4 Software Operation

10.4.1 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.

10.4.2 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.

10.4.3 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);
```

10.4.4 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);
```

10.4.5 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.

10.4.6 Power Management

10.4.6.1 PowerUp

This function is not implemented for the CAN driver.

10.4.6.2 PowerDown

This function is not implemented for the CAN driver.

10.4.6.3 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.

10.4.6.4 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}"
```

10.4.6.5 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.

10.4.7 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
```

10.5 Unit Test

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

10.5.1 Unit Test Hardware

The CANBUS2 controller should be connected to the CANBUS2 controller of an other i.MX25 board and data exchange is tested between the two boards (two controllers). The CANBUSs are not connect directly. An external transceiver on each board is needed. The i.MX25 Personality board already contains this transceiver. The two boards must be connected by the CON1 CAN port (using an serial invert male-male).

10.5.2 Unit Test Software

Table 10-2 lists the required software to run the unit tests.

Table 10-2. Software Requirements

Requirement	Description
CANApp.exe	Test file

10.5.3 Building the Unit Tests

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

1. Within the Platform Builder, choose **Build OS** > **Open Release Directory**.
   A DOS prompt is displayed.
2. Change to the CAN Tests directory: `\WINCE600\SUPPORT_PDK1_7\TEST\CANBUS\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.

10.5.4 Running the Unit Tests

On the receiver board run the application with this command: `CANApp.exe -r`

On the sender board run the application with this command: `CANApp.exe -s`

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	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\DVFC
SOC Specific Path	
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\DVFC
Driver DLL	dvfc_mc34704.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > DVFC driver support using the MC13892
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_DVFC = 1

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/voltage setpoint based on Power Manager device power states
4. Supports peripheral setpoint requests initiated by CSPDDK clock management code

11.2.1 i.MX25 Supported Functionality

1. Exposes API to application to change setpoint
2. Not support CPU load tracking to control setpoint

11.3 Hardware Operation

11.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

11.3.2 i.MX25 3-Stack Configuration

The DVFC driver is dependent upon the MC34704 PMIC interface for dynamic voltage control via I2C1 port. The I^{2}C driver SDK is needed in the DVFC driver.

11.4 Software Operation

11.4.1 i.MX25 Registry Settings

The following registry keys are required to properly load the i.MX25 DVFC module.

```
; DVFC Driver
;
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\DVFC]
   "Prefix" = "DVF"
   "Index" = dword:1
   "Dll"="dvfc_MC34704.dll"
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"    ; PMCLASS_GENERIC_DEVICE
   "order"=dword:1
```

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.MX25 Voltage/Frequency Setpoints

The i.MX25 DVFC driver supports three voltage frequency setpoints. Table 11-2 provides the voltage/frequency characteristics for these setpoints.

Table 11-2. i.MX25 DVFC Setpoints

Setpoint Name	CPU/BUS/PER Clock (MHz)	Core Voltage (V)
DDK_DVFC_SETPOINT_HIGH	399/133/66.5	1.450
DDK_DVFC_SETPOINT_MEDIUM	266/133/66.5	1.196
DDK_DVFC_SETPOINT_LOW	133/133/66.5	1.196

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.

11.4.4 DDK Interface

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

- DDKClockSetpointRequest, Section 8.6.1.2.6, “DDKClockSetpointRequest.”
- DDKClockSetpointRelease, Section 8.6.1.2.7, “DDKClockSetpointRelease.”

11.4.5 Power Management

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

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

11.5.1 i.MX25 Unit Testing

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

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

   NOTE

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

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

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

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

Chapter 12
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.

12.1 eSDHC Driver Summary

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

Table 12-1. eSDHC Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\ESDHC
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ESDHC
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\ESDHC
Driver DLL	esdhc.dll
SDK Library	esdhcbase_common_fsl_v2PDK1_7.lib, esdhcbase_<*Target SOC*>.lib, sdcardlib.lib, sdhclib.lib, sdbus.lib
Catalog Item	Third Party > BSP > Freescale i.MX25 3DS PDK1_7: ARMV4I > Device Drivers > SD Host Controller > Enhanced SD Host Controller 1 (ESDHC1) Support
SYSGEN Dependency	SYSGEN_SD_MEMORY=1
BSP Environment Variables	BSP_NOESDHC= BSP_ESDHC1=1 IMGSDBUS2

12.2 Supported Functionality

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

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

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

12.3 Hardware Operation

Refer to the chapter on the eSDHC in the *i.MX25 Applications Processor Reference Manual* for detailed operation and programming information.

12.3.1 Conflicts with Other Peripherals and Catalog Options

12.3.1.1 Conflicts with SoC Peripherals

All eSDHC1 pads can be configured for their primary function in Alternate Mode 0. However, some eSDHC1 pads are shared with CSPI2 in ALT Mode1, FEC in ALT Mode 2, GPIO2 pins in ALT 5 and SLCDC in ALTMode6. eSDHC2 pads can be configured in ATL Mode 1 over FEC pin. Some eSDHC2 pads are conflicts with FEC/ATA/GPIO43/LCDC. Some UART2 pads can be configured as eSDHC1 DATA5~DATA8 for 8-bit mode.

12.3.1.2 Conflicts with Other Board Peripherals

No conflicts.

12.4 Software Operation

The eSDHC driver follows the Microsoft-recommended architecture (standard host controller driver) for Secure Digital Host Controller drivers, whenever possible. The details of this architecture and its operation can be found in the Platform Builder Help under the heading **Secure Digital Card Driver Development**

Concepts, or in the online documentation at the following URL:
http://msdn2.microsoft.com/en-us/library/aa925967.aspx

12.4.1 Required Catalog Items

12.4.1.1 SD and MMC Memory Card Support

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

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

12.4.2 eSDHC Registry Settings

12.4.2.1 i.MX25 SDHC Registry Settings

```
      ; @CESYSGEN IF CE_MODULES_SDBUS

      IF BSP_ESDHC1
      [HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ESDHC1_MX25]
          "Order"=dword:21
          "Dll"="esdhc.dll"
          "Prefix"="SHC"
          "Index"=dword:1
         "DisableDMA"=dword:0                    ; Use this reg setting to disable internal 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

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

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

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

      [HKEY_LOCAL_MACHINE\System\StorageManager\Profiles\SDMemory]
          "Name"="SD Memory Card"
          "Folder"="SDMemory"
      ENDIF BSP_ESDHC1

; @CESYSGEN ENDIF CE_MODULES_SDBUS
```

12.4.3 DMA Support

12.4.3.1 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.

12.4.3.1.1 i.MX25 DMA Support

For i.MX25, both DMA mode and non-DMA modes are supported by the driver. DMA mode is used by default, and user can change the *DisableDMA* value in registry file esdhc_mx25.reg to enable non-DMA mode. Internal DMA on eSDHC is used. Two internal DMA modes are supported by the eSDHC hardware: Simple DMA and Advanced DMA. The driver supports only ADMA mode.

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

- Start of the buffer should be a page aligned address (4096 bytes)

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

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

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

12.4.4 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.

12.4.4.1 i.MX25 Power Management

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

12.5 Unit Test

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

- File System Driver Test

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

12.5.1 Unit Test Hardware

Table 12-2 lists the required hardware to run the unit tests.

Table 12-2. 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)

12.5.2 Unit Test Software

Table 12-3 lists the required software to run the unit tests.

Table 12-3. Software Requirements

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

12.5.3 Building the Unit Tests

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

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

12.5.4 Running the Unit Tests

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

12.5.4.1 File System Driver Test

Use command line

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

to run the tests on an SD card. For MMC cards, use

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

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

12.5.4.2 Storage Device Block Driver Read/Write Tests

Use the command line

```
tux -o -d rwtest -c “-z”
```

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

12.5.4.3 Storage Device Block Driver API Tests

Use the command line

```
tux -o -d disktest -c “-z”
```

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

12.5.4.4 Storage Device Block Driver Performance Tests

Use the command line

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

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

12.5.4.5 Partition Driver Test

Use command line

```
tux -o -d msparttest -c “-z”
```

to run the tests.

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

12.5.5 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.

12.6 Secure Digital Card Driver API Reference

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

Chapter 13
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.

13.1 ESAI Driver Summary

The ESAI consists of independent transmitter and receiver sections, each section with its own clock generator. It is called synchronous because all serial transfers are synchronized to a clock. Up to six transmitters and four receivers are supported. Table 13-1 provides a summary of source code location, library dependencies and other BSP information.

Table 13-1. ESAI Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\ESAI
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ESAI
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\ESAI
Driver DLL	esai_wm8580.dll
SDK Library	esai_common_fsl_v2_PDK1_7.lib, esai_<*Target SOC*>.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > ESAI
SYSGEN Dependency	SYSGEN_AUDIO
BSP Environment Variables	BSP_ESAI=1

13.2 Supported Functionality

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

1. Conforms to the audio driver architecture as defined for Windows Embedded CE 6.0 and all related operating systems
2. Uses double-buffered DMA operations to transfer audio data
3. Supports multi-channel PCM wave data playback function
4. Supports multi-channel PCM wave data record function
5. Supports 16-bit and 24-bit PCM data. (24-bit PCM data is packed in bits 23–0 of the 32 bits)

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

13.3 Hardware Operation

Refer to the chapter on the ESAI in the *i.MX25 Multimedia Applications Processor Reference Manual* for detailed operation and programming information.

13.3.1 Conflicts with Other Peripherals and Catalog Items

All the pins of ESAI can be configured for alternate functionality (CSI, SIM) using the i.MX25 IOMUX. The configuration is specified by the BSP serial driver. Changing this configuration results in a conflict and prevents proper operation of the ESAI driver.

13.3.1.1 Conflicts with 3-Stack Peripherals

On i.MX25 3-Stack System, the pins used for communication between ESAI and the external codecs are shared with the CSI and the SIM1/SIM2 controllers. Do not include the camera, SIM1 or SIM2 drivers while including ESAI in the image.

13.3.2 Hardware Limitation

13.3.2.1 Channel Order

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

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

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

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

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

Channel Number = 2, CH0,CH1->TX0

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

13.3.2.2 Full Duplex Support

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

13.4 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.**

13.4.1 Required Catalog Items

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

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

13.4.2 ESAI Registry Settings

13.4.2.1 i.MX25 Registry Settings

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

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

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

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

13.4.3 Supported Wave Data Format

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

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

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

13.4.4 DMA Support

13.4.4.1 DMA Support

Double-buffer is used for audio data transfer.

13.4.5 Power Management

This function is not implemented for the ESAI driver.

13.4.5.1 PowerUp

This function is not implemented for the ESAI driver.

13.4.5.2 PowerDown

This function is not implemented for the ESAI driver.

13.4.5.3 IOCTL_POWER_CAPABILITIES

This function is not implemented for the ESAI driver.

13.4.5.4 IOCTL_POWER_GET

This function is not implemented for the ESAI driver.

13.4.5.5 IOCTL_POWER_SET

This function is not implemented for the ESAI driver.

13.5 Unit Test

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

13.5.1 Building the Unit Test

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

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

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

13.5.2 Hardware Setup

The audio extension card is necessary for the ESAI test. The multi-channel audio codec for playback and record function is located on the audio extension card. Connect the audio card with J6 on the i.MX25 Personality board (refer to the *User's Manual* for details).

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

13.5.3 Running the Unit Test

13.5.3.1 Playback Function Test

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

Usage: `m_player wave_file_name channel_number channel_mask`

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

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

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

13.5.3.2 Record Function Test

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

Usage: `m_recorder wave_file_name seconds_length sample_rate bit_depth channel_number channel_mask`

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

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

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

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

NOTE

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

13.5.4 Known Issues

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

Chapter 14
General Purpose Timer (GPT) Driver

The GPT is a multipurpose module used to measure intervals or generate periodic output. The GPT counter value can be captured in a register using an event on an external pin. The GPT can also generate an event on a chip boundary signal and an interrupt when the timer reaches a programmed value.

14.1 GPT Driver Summary

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

Table 14-1. GPT Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\GPT
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\GPT
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\GPT
Driver DLL	gpt.dll
SDK Library	gptsdk.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > GPT > GPTn
SYSGEN Dependency	N/A

BSP Environment Variables	BSP_GPT=1
	BSP_GPT=2
	BSP_GPT=3
	BSP_GPT=4

14.2 Supported Functionality

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

1. Clock source selection including IPG_CLK (microsecond level precision) and GPT_32KCLK (microsecond level precision)
2. Both reset and free-run mode count operation
3. Two power management modes: power on and power off
4. Exposes the SDK API interface which is used by application

NOTE

GPT_IPGCLK is adapted for short time period (GPT_IPGCLK is 66.5 MHz, the maximum time period is 64.599 seconds), while the maximum time period of GPT_32KCLK is approximately 37 hours, 16 minutes, 57 seconds.

14.3 Hardware Operation

Refer to the chapter on GPT in the *i.MX25 Applications Processor Reference Manual* for detailed hardware operation and programming information.

14.3.1 Conflicts with Other Peripherals and Catalog Items

Because the external GPT clock source is not used, GPT module does not conflict with other peripherals.

14.4 Software Operation

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

14.4.1 GPT Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\GPTn]
        "Prefix"="GPT"
        "Dll"="gpt.dll"
        "Index"=dword:n           ;n is 1,2,3,4
```

14.4.2 Communicating with the GPT

The GPT driver controls the General Purpose Timer. This timer is used to provide high resolution (microsecond) timing functionality to other platform modules. The GPT is a stream interface driver and is thus accessed through the file system APIs. To communicate using the GPT, a handle to the device must first be obtained using the **GptOpenHandle** function. Subsequent commands to the device are issued using various APIs supported by this driver. For more information about the API refer to Section 14.7, "GPT SDK API Reference." To use this API, it is necessary to include the `gptsdk.lib` library.

14.4.2.1 Creating a Handle to the GPT

To communicate with the GPT, a handle to the device must first be created using the **GptOpenHandle** API. The default GPT port is 1.

The following code shows how to open a handle to the GPT:

```
// Global data
// Handle to the GPT device
HANDLE g_hGpt = NULL;

// opening the GPT1 port.
g_hGpt = GptOpenHandle(L"GPT1:");
```

14.4.2.2 Create Event for GPT

```
HANDLE GptCreateTimerEvent(HANDLE hGpt, LPTSTR eventName)
// Function: GptCreateTimerEvent
//
// This method returns a handle triggered
// when the GPT timer period has elapsed.
//
// Parameters:
//      hGpt
//          [in] Handle to GPT driver.
//
//      eventName
//          [in] String identifying timer event.
//
// Returns:
//      Timer event handle created. Handle is NULL if failure.
```

The following is an example:

```
// Name to create the named event for Timer
#define GPT_EVENT_NAME L"GptTest1"

// create an event for the timer interrupt
hGptIntr = GptCreateTimerEvent(hGpt, GPT_EVENT_NAME);
```

14.4.2.3 Configuring the GPT

Calling the **GptStart**(`g_hGpt, pTimerConfig`) function starts the GPT module and enables the timer event trigger. g_hGpt is valid and opened handle for GPT, and pTimerConfig struct is as follows:

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

and timerSrc may select *GPT_IPGCLK* or *GPT_32KCLK.*

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

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

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

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

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

After the GPT starts to time and the timer event handle is created, call the following command to wait the coming of the predefined time:

```
// waiting for event triggering
if(WaitForSingleObject(g_hGptIntr, INFINITE) == WAIT_OBJECT_0)
{
}
```

14.4.2.4 Closing the Handle to the GPT

To close the GPT handle, call the **GptCloseHandle** API. Before performing the close operation, stop the timer using **GptStop** API. It is always advised to call **GptReleaseTimerEvent** to release any pending timer events before closing the handle.

The following code shows how to close the GPT Handle:

```
// Name to create the named event for Timer
#define GPT_EVENT_NAME L"GptTest1"

// releasing the Timer Event.
GptReleaseTimerEvent(g_hGpt, eventString);
GptStop(g_hGpt);
GptCloseHandle(g_hGpt);
```

To pause the timer and then restart for a moment, use the **GptStop** function, as follows:

```
GptStop(g_hGpt);
Sleep(sometime);
GptResume(g_hGpt);

BOOL GptResume(HANDLE hGpt)
// Function: GptResume
//
// This method reactivates the GPT(Usually called after a Stop))
//
// Parameters:
//      hGpt
//          [in] Handle to GPT driver.
//
// Returns:
//      TRUE if success.
//      FALSE if failure.
```

14.4.3 DMA Support

The GPT driver does not use the DMA.

14.5 Power Management

The primary method for limiting power consumption in the GPT module is to gate off all clocks to the module when the GPT is not used. The clock is enabled when an application calls **GPT_Open()**. This clock then remains enabled as long device is kept open. The GPT clock is turned off when the application closes the device using **GPT_Close()**.

14.5.1 PowerUp

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

14.5.2 PowerDown

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

14.5.3 IOCTL_POWER_SET

This function is not implemented for the GPT driver.

14.6 Unit Test

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

14.6.1 Unit Test Hardware

Table 14-2 lists the required hardware to run the unit tests.

Table 14-2. Hardware Requirements

Requirement	Description
No additional hardware required	

14.6.2 Unit Test Software

Table 14-3 lists the required software to run the unit tests.

Table 14-3. Software Requirements

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

14.6.3 Building the Unit Tests

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

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

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

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

14.6.4 Running the Unit Tests

To run this test the `tux.exe` and `kato.dll` files must be present in the release directory. These files are not present by default and need to be copied from this location:

```
\Program Files\Microsoft Platform Builder\6.00\cepb\wcetk\ddtk\armv4i
```

to the release directory.

To run the test using the Target Control window use the following steps:

1. Within the Platform Builder, go to the Target menu option and select the Target Control menu option. This opens a Windows CE Command Prompt window
2. Run on the Command Prompt windows this command: `s tux -o -d gpttest.dll`

The test starts and the results can be viewed in the Output panel in the Visual Studio.

Table 14-4 describes the test cases contained in the GPT tests.

Table 14-4. GPT Test Cases

Test Case	Description
1: TST_StartBeforeCfg	Attempt to start the GPT timer without setting the timer period (expected failure)
2: TST_OpenMultipleHandle	Attempt to open multiple GPT Handles (expected failure)
3: TST_ComparewithSysTick	Check timer accuracy with system clock
4:TST_PeriodicMode	Periodic mode test
5: TST_FreerunMode	Free run mode test
6: TST_StopAndResume	Stop and resume test

14.7 GPT SDK API Reference

14.7.1 GPT SDK Functions

14.7.1.1 GptOpenHandle

This API creates a handle to the GPT stream driver.

```
HANDLE GptOpenHandle(
        LPCWSTR lpDevName);
```

Parameters

lpDevName	[in] Device name to open
Return Values	Open handle to the specified file indicates success INVALID_HANDLE_VALUE indicates failure

Remarks	Use the GptCloseHandle function to close the handle returned by GptOpenHandle()

14.7.1.2 GptCreateTimerEvent

This API is used to create the GPT Timer event.

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

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
eventName	[in] Pointer to a null-terminated string that specifies the name of the object
Return Values	Non-null handle to the specified event indicates success. NULL indicates failure
Remarks	Use the **GptReleaseTimerEvent** function to close the event. The system closes the handle automatically when the process terminates. The event object is destroyed when its last handle has been closed.

14.7.1.3 GptStart

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

```
BOOL GptStart(
          HANDLE hGpt,
          pGPT_Config pTimerConfig);
```

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
pTimerConfig	[in] Object of the **pGPT_Config** structure
Return Values	TRUE on success and FALSE indicates a failure
Remarks	Set desired event trigger time and start GPT

14.7.1.4 GptGetCounterValue

This API gets the current counter register value.

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

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
pTimerCount	[in] Pointer to the variable which receives current counter value
Remarks	None

14.7.1.5 GptResume

This API reactivates the GPT.

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

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
Remarks	Often called after a stop

14.7.1.6 GptStop

This API disables the GPT interrupt and stops the GPT timer.

```
BOOL GptStop(
        HANDLE hGpt);
```

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
Return Values	TRUE on success and FALSE indicates a failure
Remarks	None

14.7.1.7 GptReleaseTimerEvent

This API closes the currently open GPT Timer Event.

```
BOOL GptReleaseTimerEvent(
        HANDLE hGpt,
        LPTSTR eventName);
```

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
eventName	[in] Pointer to a null-terminated string that specifies the name of the object
Return Values	Nonzero indicates success; Zero indicates failure To get extended error information, call GetLastError()
Remarks	None

14.7.1.8 GptCloseHandle

This API closes a handle to the GPT driver.

```
BOOL GptCloseHandle(
        HANDLE hGpt);
```

Parameters

hGpt	[in] Handle to the GPT driver returned by **GptOpenHandle** API
Return Values	Nonzero indicates success; Zero indicates failure To get extended error information, call GetLastError()
Remarks	None

14.7.2 GPT Driver Structures

14.7.2.1 GPT_Config

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

Members

timerMode	Selects between two supported modes: reset or periodic mode (timerModePeriodic) and free-running mode (timerModeFreeRunning)
period	Counter period (in microsecond)
timerSrc	Selects GPT clock source: GPT_IPGCLK or GPT_32KCLK

14.7.2.2 GPT_TIMER_SRC_PKT

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

Members

timerSrc	Select clock source between two supported timer clock sources: GPT_IPGCLK or GPT_32KCLK

Chapter 15
Fast Ethernet Controller (FEC) Driver

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

15.1 Fast Ethernet Driver Summary

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

Table 15-1. FEC Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\FEC
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\FEC
Driver DLL	fec.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>:ARMV4I > Device Drivers > FEC

SYSGEN Dependency	SYSGEN_NDIS=1
	SYSGEN_TCPIP=1
	SYSGEN_WINSOCK=1

BSP Environment Variables	BSP_NOFEC=

15.2 Supported Functionality

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

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

15.3 Hardware Operations

The Fast Ethernet Controller connects with the external transceiver using standard RMII (Reduced Media Independent Interface) connection. All the registers in the external transceiver (DP83640) can be accessed by the RMII compatible management frames. The attached transceiver for the Fast Ethernet Controller can detect the speed of the ethernet network automatically by the auto-negotiation process. The software accesses the status register of attached transceiver to determine the speed of the ethernet network (10 Mbps or 100 Mbps)

15.3.1 Conflicts with Other Peripherals and Catalog Items

15.3.1.1 Conflicts with SoC Peripherals

No conflicts.

15.3.1.2 Conflicts with 3-Stack Peripherals

On i.MX25 3-Stack System, the pins used for communication with the FEC external transceiver are shared with the SD host controller, port 2. This port is used for the communication with the onboard APM6628 WIFI chip. Do not include the FEC and onboard WIFI in the same image.

15.4 Software Operations

The Fast Ethernet driver follows the Microsoft-recommended architecture for NDIS miniport drivers. The details can be found in the Platform Builder Help at the following location:

Developing a Device Driver > **Windows Embedded CE Drivers** > **Network Drivers** > **Network Driver Development Concepts** > **Miniports, Intermediate Drivers, and Protocol Drivers**.

15.4.1 FEC Driver Registry Settings

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

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

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

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

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

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

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

15.5 Unit Tests

The Fast Ethernet driver is tested using the following:

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

15.5.1 Unit Test Hardware

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

Table 15-2. Hardware Requirements

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

15.5.2 Unit Test Software

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

Table 15-3. Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
Ws2bvt.dll	Test .dll file for Winsock 2.0 Test (v4/v6)
Perflog.dll	Module that contains functions that monitor and log performance for Winsock Performance Test
Perf_winsock2.dll	Test .dll file for Winsock Performance Test
Perf_winsockd2.exe	Test .exe file (server program) for Winsock Performance Test
Ndt.dll	Protocol driver for One-card network card miniport driver test
Ndt_1c.dll	Test .dll for One-card network card miniport driver test
Ndp.dll	MS_NDP protocol driver for NDIS performance test
Perf_ndis.dll	Test .dll file NDIS performance test

15.5.3 Building the Unit Tests

15.5.3.1 Network Utilities Related Tests

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

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

15.5.3.2 Winsock 2.0 Test (v4/v6)

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

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

15.5.3.3 Winsock Performance Test

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

`Perf_winsock2.dll` in:

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

`Perf_winsockd2.exe` in:

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

15.5.3.4 One-Card Network Card Miniport Driver Test

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

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

15.5.3.5 NDIS Performance Test

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

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

15.5.4 Running the Unit Tests

15.5.4.1 Network Utilities Related Tests

15.5.4.1.1 Ping Tests

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

15.5.4.1.2 Browsing

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

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

15.5.4.1.3 FTP Tests

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

```
services start FTP0:
```

15.5.4.2 Winsock 2.0 Test (v4/v6)

The test can be executed by using

```
tux -o -d Ws2bvt.dl
```

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

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

15.5.4.3 Winsock Performance Test

Start the server on the PC by typing

```
Perf_winsockd2 - install
```

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

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

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

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

NOTE

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

15.5.4.4 One-Card Network Card Miniport Driver Test

This test can be done by including `ndt.dll` and `ndt_1c.dll` in the image, and starting the test by entering

```
tux -o -d ndt_1c.dll –c “-t FEC1”
```

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

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

15.5.4.5 NDIS Performance Test

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

```
tux -o -d perf_ndis.dll –c “FEC1”
```

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

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

15.6 Fast Ethernet Driver API Reference

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

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

Chapter 16
Inter-Integrated Circuit (I^2C) Driver

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

16.1 I^2C Driver Summary

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

Table 16-1. I^2C Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\I2C
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\I2C
Platform Driver Path	..\PLATFORM*Target Platform*>\SRC\DRIVERS\I2C
Import Library	N/A
Driver DLL	i2csdk.dll i2c.dll
Catalog Item	Third Party > BSP > Freescale <TGTPLAT> > Device Drivers > I2CBus
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_I2CBUS1=1

16.2 Supported Functionality

The I^2C driver supports the following features:

1. I^2C communication protocol
2. Multiple I^2C controllers
3. I^2C master mode of operation
4. I^2C slave mode of operation
5. Stream interface
6. Two power management modes: full on and full off

16.3 Hardware Operation

16.3.1 Conflicts with Other Peripherals and Catalog Items

16.3.1.1 Conflicts with SoC Peripherals

The i.MX25 platform contains three I^2C modules, but only one of these modules may be used on the i.MX25 3-Stack Platform System. The I2C1 module I2C2 and I2C3 pins are not available in the i.MX25 3-Stack Platform System and the pads are used for LCDC, CSPI, GPIO and FEC.

16.3.1.2 Conflicts with Board Peripherals

No conflicts.

16.4 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.

16.4.1 Registry Settings

16.4.1.1 i.MX25 Registry Settings

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

```
IF BSP_I2CBUS1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\I2C1]
   "Prefix"="I2C"
   "Dll"="i2c.dll"
   "Index"=dword:1
   "Order"=dword:0
ENDIF ; BSP_I2CBUS1

IF BSP_I2CBUS2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\I2C2]
   "Prefix"="I2C"
   "Dll"="i2c.dll"
   "Index"=dword:2
   "Order"=dword:0
ENDIF ; BSP_I2CBUS2

IF BSP_I2CBUS3
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\I2C3]
   "Prefix"="I2C"
   "Dll"="i2c.dll"
   "Index"=dword:3
   "Order"=dword:0

ENDIF ; BSP_I2CBUS3
```

16.4.2 Communicating with the I^2C

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

16.4.3 Creating a Handle

Call the **CreateFile** function to open a connection to the I^2C device. An I^2C port must be specified in this call. The format is `I2CX:`, with `X` being the number indicating the I^2C port. This number should not exceed the number of I^2C instances on the platform. If an I^2C port does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the I^2C:

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

The following code example shows how to open an I^2C port.

```
// Open the I2C port.
hI2C = CreateFile (CAM_I2C_PORT,                       // name of device
                  GENERIC_READ | GENERIC_WRITE,         // access (read-write) mode
                  FILE_SHARE_READ | FILE_SHARE_WRITE,   // sharing mode
                  NULL,                                 // security attributes (ignored)
                  OPEN_EXISTING,                        // creation disposition
                  FILE_FLAG_RANDOM_ACCESS,              // flags/attributes
                  NULL);                                // template file (ignored)
```

Before writing to or reading from an I^2C port, configure the port. When an application opens an I^2C port, it uses the default configuration settings, which might not be suitable for the device at the other end of the connection.

16.4.4 Configuring the I^2C

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

- Setting the master or slave mode
- Setting the I^2C clock rate

Before these actions can be taken, a handle to the I^2C port must already be opened. Each of these steps requires a call to the **DeviceIoControl** function. As parameters, the I^2C port handle, appropriate IOCTL code, and other input and output parameters are required. Use the helper APIs to correctly configure the port.

As an example, the code below configures an I^2C port:

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

if (hI2C == INVALID_HANDLE_VALUE)
{
        ERRORMSG(1, (L"Unable to open handle to I2C block\r\n"));
        retVal = -1;
        goto exit;
}

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

if (!I2CSetFrequency(hI2C, EEPROM_CLOCK_RATE))
{
        ERRORMSG(1, (L"Unable to set frequency\r\n"));
        retVal = -1;
        goto exit;
}
```

16.4.5 Data Transfer Operations

The I^2C driver provides one command, Transfer, that facilitates performing both reads and writes through the I^2C. The basic unit of data transfer in the I^2C driver is the I2C_PACKET, which contains a buffer for reading or writing data and a flag that specifies whether the desired operation is a read or a write. An array of these packets makes up an I2C_TRANSFER_BLOCK object, which is needed to perform a Transfer operation. The steps below detail the process of performing write and read operations through the I^2C.

Before these actions can be taken, a handle to the I^2C port must already be opened, and it should already be configured in the correct mode with the correct frequency.

To perform an I^2C transfer:

1. Create an array of I2C_PACKET objects and initialize the fields of each packet as follows:
   a) Set the *byRW* field to I2C_RW_WRITE to specify that the I^2C operation is a write, or I2C_RW_READ to specify that the I^2C operation is a read.
   b) Set the *byAddr* field to the 7-bit I^2C slave address of the device to which the data is written.

NOTE

The byAddr field requires the 7-bit I2C slave address, aligned to the least significant 7 bits. This address is shifted left one bit and OR-ed with the read/write bit to compose the 8-bit value sent out during the I^2C slave address cycle. In older versions of this driver, the slave address was entered as the most significant 7 bits of the 8-bit value.

   c) If *byRW* is set to I2C_RW_WRITE, create a buffer of bytes and fill it with the data to write to the slave device. Set the *pbyBuf* field to point to this buffer. If *byRW* is set to I2C_RW_READ, create a buffer of bytes to hold the data which is read from the slave device.
   d) Set the *wLen* field to the size, in bytes, of the read or write buffer. This indicates the number of bytes to write or read.
   e) Set the *lpiResult* field to point to an integer that holds the return value from the write operation.
2. Call the I2CTransfer SDK API to start the I^2C transfer.
3. After calling the I2CTransfer function, check the *lpiResult* field if the function returned FALSE, to narrow down the type of error that occurred.

The following code example demonstrates how to perform a transfer that contains one write and one read packet. The write is performed before the read operation.

```
I2C_TRANSFER_BLOCK I2CXferBlock;
I2C_PACKET I2CPacket[2];
BYTE byAddr = 0x2D;                          // Slave Address
BYTE byOutData = 0x39;                       // Data to write
BYTE byInData;                               // Read buffer

         // Packet 0 contains write operation
I2CPacket[0].pbyBuf = (PBYTE) &byOutData;
I2CPacket[0].wLen = sizeof(byOutData);

I2CPacket[0].byRW = I2C_RW_WRITE;
I2CPacket[0].byAddr = byAddr;
I2CPacket[0].lpiResult = lpiResult;

         // Packet 1 contains read operation
I2CPacket[1].pbyBuf = (PBYTE) &byInData;
I2CPacket[1].wLen = sizeof(byInData);

I2CPacket[1].byRW = I2C_RW_READ;
I2CPacket[1].byAddr = byAddr;
I2CPacket[1].lpiResult = lpiResult;

I2CXferBlock.pI2CPackets = I2CPacket;
I2CXferBlock.iNumPackets = 2;

         // Transfer data via I2C
if (!I2CTransfer(hI2C,&I2CXferBlock))
         {
                  ERRORMSG(1, (_T("Data transfer failed!\r\n")));
                  retVal = -1;
                  goto exit; // examine value in lpiResult
         }
```

16.4.5.1 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.

16.4.6 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.

16.4.7 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.

16.4.7.1 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.

16.4.7.2 PowerDown

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

16.4.7.3 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.

16.5 Unit Test

16.5.1 Unit Test Hardware

The unit tests are not supported for this release.

16.5.2 Unit Test Software

The unit tests are not supported for this release.

16.5.3 Building the Unit Tests

The unit tests are not supported for this release.

16.5.4 Running the Unit Tests

The unit tests are not supported for this release.

16.6 Hardware Limitations

For the slave function, the hardware does not distinguish between a START and REPEATED START signal from the I^{2}C bus. Hence the driver checks the IAAS address cycle start flag to detect a new I^{2}C transmission.

16.7 I^{2}C Driver API Reference

16.7.1 I^{2}C Driver IOCTLS

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

16.7.1.1 I2C_IOCTL_GET_CLOCK_RATE

This DeviceIoControl request retrieves the clock rate divisor. The value is not the absolute peripheral clock frequency. The value retrieved should be compared against the I^{2}C specifications to obtain the true frequency.

Parameters

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

16.7.1.2 I2C_IOCTL_GET_SELF_ADDR

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

Parameters

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

16.7.1.3 I2C_IOCTL_IS_MASTER

This DeviceIoControl request determines whether the I^{2}C 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

16.7.1.4 I2C_IOCTL_IS_SLAVE

This DeviceIoControl request determines whether the I^{2}C 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

16.7.1.5 I2C_IOCTL_RESET

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

16.7.1.6 I2C_IOCTL_SET_CLOCK_RATE

This DeviceIoControl request initializes the I^{2}C 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^{2}C specification. If absolute clock frequency must be used, use the macro I2C_MACRO_SET_FREQUENCY.

Parameters

lpInBuffer	Pointer to the divisor index. Refer to the I^{2}C specification to obtain the true clock frequency
nInBufferSize	Size in bytes of the divisor index

16.7.1.7 I2C_IOCTL_SET_FREQUENCY

This DeviceIoControl request estimates the nearest clock rate acceptable for I^2C device and initialize the I^2C device to use the estimated clock rate divisor. If the estimated clock rate divisor index is required, refer to the macro I2C_MACRO_GET_CLOCK_RATE to determine the estimated index.

Parameters

lpInBuffer	Pointer to the desired I^2C frequency
nInBufferSize	Size in bytes of the I^2C frequency requested

16.7.1.8 I2C_IOCTL_SET_MASTER_MODE

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

16.7.1.9 I2C_IOCTL_SET_SELF_ADDR

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

Parameters

lpInBuffer	Pointer to the expected I^2C device address, valid range is [0x00–0x7F]
nInBufferSize	Size in bytes of the I^2C device address
Remarks	The device expects to respond when any master on the I^2C bus wishes to proceed with any transfer. This IOCTL has no effect if the I^2C device is in Master mode.

16.7.1.10 I2C_IOCTL_SET_SLAVE_MODE

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

16.7.1.11 I2C_IOCTL_TRANSFER

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

Parameters

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

16.7.1.12 I2C_IOCTL_ENABLE_SLAVE

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

16.7.1.13 I2C_IOCTL_DISABLE_SLAVE

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

16.7.1.14 I2C_IOCTL_GET_SLAVESIZE

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

16.7.1.15 I2C_IOCTL_SET_SLAVESIZE

This DeviceIoControl request sets the interface buffer size of the I^2C device for slave mode. The maximum size for the buffer is configured by I2CSLAVEBUFSIZE.

16.7.1.16 I2C_IOCTL_GET_SLAVE_TXT

This DeviceIoControl request gets the current data from interface buffer of the I^2C device for slave mode. Both slave device or external master can change this data.

16.7.1.17 I2C_IOCTL_SET_SLAVE_TXT

This DeviceIoControl request sets data to interface buffer of the I^2C device for slave mode. An external I^2C master can get this data immediately from driver after it connects the slave.

16.7.2 I^2C Driver SDK Encapsulation

16.7.2.1 I2COpenHandle

This function retrieves the I^2C device handle.

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

Parameters

lpDevName	The I^2C device name for retrieving handle from CreateFile()
Return Values	Returns the handle for I^2C driver, returns INVALID_HANDLE_VALUE if failure

16.7.2.2 I2CCloseHandle

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

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

Parameters

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

16.7.2.3 I2CSetSlaveMode

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

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

Parameters

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

16.7.2.4 I2CSetMasterMode

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

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

Parameters

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

16.7.2.5 I2CIsMaster

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

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

Parameters

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

16.7.2.6 I2CIsSlave

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

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

Parameters

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

16.7.2.7 I2CGetClockRate

This function retrieves the clock rate divisor. This value is not the absolute peripheral clock frequency. The value retrieved should be compared against the I^2C specifications to obtain the true frequency.

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

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pwClkRate	Pointer of WORD variable that retrieves divisor index. Refer to I^2C specification to obtain the true clock frequency
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

16.7.2.8 I2CSetClockRate

This function initializes the I^2C device with the given clock rate. This function does not expect to receive the absolute peripheral clock frequency. Rather, it expects the clock rate divisor index stated in the I^2C specification. If absolute clock frequency must be used, use the function I2CSetFrequency().

```
BOOL I2CSetClockRate(
        HANDLE hDev,
        WORD wClkRate);
```

Parameters

hDev	I^2C device handle retrieved from CreateFile()
wClkRate	Divisor index. Refer to I^2C specification to obtain the true clock frequency
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

16.7.2.9 I2CSetFrequency

This function estimates the nearest clock rate acceptable for I^2C device and initializes the I^2C device to use the estimated clock rate divisor. If the estimated clock rate divisor index is required, refer to the macro I2CGetClockRate to determine the estimated index.

```
BOOL I2CSetFrequency(
        HANDLE hDev,
        DWORD dwFreq);
```

Parameters

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

16.7.2.10 I2CSetSelfAddr

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

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

Parameters

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

16.7.2.11 I2CGetSelfAddr

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

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

Parameters

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

16.7.2.12 I2CTransfer

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

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

Parameters

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

16.7.2.13 I2CReset

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

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

Parameters

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

16.7.2.14 I2CEnableSlave

This function enables a I^{2}C slave access from the bus. After the I^{2}C slave interface is enabled, the I^{2}C slave driver waits for an external master access.

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

Parameters

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

16.7.2.15 I2CDisableSlave

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

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

Parameters

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

16.7.2.16 I2CGetSlaveSize

This function returns the I^{2}C slave interface buffer length. The I^{2}C 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^{2}C slave module has been turned off.

```
BOOL I2CGetSlaveSize(
        HANDLE hDev,
        PDWORD pdwSize);
```

Parameters

hDev	I^{2}C 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

16.7.2.17 I2CSetSlaveSize

This function sets the I^{2}C 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^{2}C 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^{2}C slave module has been turned off.

```
BOOL I2CSetSlaveSize(
        HANDLE hDev,
        DWORD dwSize);
```

Parameters

hDev	I^{2}C 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

16.7.2.18 I2CGetSlaveText

This function returns the I^{2}C slave interface buffer text. The I^{2}C 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^{2}C slave module has been turned off.

```
BOOL I2CGetSlaveText(
         HANDLE hDev,
         PBYTE pbyTextBuf,
         DWORD dwBufSize,
         PDWORD pdwTextLen );
```

Parameters

hDev	I^{2}C 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

16.7.2.19 I2CSetSlaveText

This function returns the I^{2}C slave interface buffer text. The I^{2}C 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^{2}C slave module has been turned off.

```
BOOL I2CSetSlaveText(
         HANDLE hDev,
         PBYTE pbyTextBuf,
         DWORD dwTextLen );
```

Parameters

hDev	I^{2}C 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

16.7.3 I^{2}C Driver Structures

16.7.3.1 I2C_PACKET

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

```
typedef struct {
         BYTE byAddr;
         BYTE byRW;
         PBYTE pbyBuf;
         WORD wLen;
```

```
            LPINT lpiResult;
    } I2C_PACKET, *PI2C_PACKET;
```

Members

byAddr	7-bit slave address that specifies the target I^{2}C 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

16.7.3.2 I2C_TRANSFER_BLOCK

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

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

Members

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

Chapter 17
Keypad Driver

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

17.1 Keypad Driver Summary

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

Table 17-1. Keypad Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\KEYBD
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\KEYBD
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\KEYBD
Driver DLL	KbdUs.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX25 3DS PDK1_7: ARMV4I > Device Drivers > Input Devices > Keyboard US/keypad
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOKEYPAD=

17.2 Supported Functionality

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

1. Conforms to the Microsoft Layout Manager Interface
2. Multiple simultaneous key presses
3. Two power management modes, full on and full off
4. Keypad Port (KPP) module, which is an internal module that can detect, debounce, and decode one key on the keypad, or two keys pressed simultaneously

17.3 Hardware Operation

17.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

17.3.2 Keypad

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

17.3.2.1 i.MX25 Keypad Mapping

The 24-key keypad is located on the Personality board and the mapping is shown in Table 17-2.

Table 17-2. Keypad Mapping

Label	Key
SW13	0
SW14	1
SW29	2
SW38	3
SW9	4
SW10	5
SW30	6
SW37	7
SW40	UP
SW36	DOWN
SW31	LEFT
SW39	RIGHT
SW18	ENTER
SW17	TAB
SW32	SPACE
SW34	BACKSPACE

17.4 Software Operation

The keypad driver follows the Microsoft-recommended architecture for keyboard drivers. The details of this architecture and its operation can be found in the CE help documentation at the following location: **Developing a Device Driver** > **Windows Embedded CE Drivers** > **Keyboard Drivers** > **Keyboard Driver Development Concepts**

17.4.1 Keypad Scan Codes and Virtual Keys

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

17.4.1.1 i.MX25 Scan Code Mapping table

Table 17-3 shows the scan code mapping.

Table 17-3. Scan Code Mapping

Key	Keypad Scan Code	Virtual Key
UP	0	VK_UP
RIGHT	1	VK_RIGHT
3	2	3
7	3	7
DOWN	4	VK_DOWN
LEFT	5	VK_LEFT
2	6	2
6	7	6
BACKSPACE	8	VK_BACK
ENTER	9	VK_RETURN
1	10	1
5	11	5
SPACE	12	VK_SPACE
TAB	13	VK_TAB
0	14	0
4	15	4

17.4.2 Power Management

17.4.2.1 BSPKppPowerOn

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

17.4.2.2 BSPKppPowerOff

This function powers down the keypad.

17.4.2.3 IOCTL_POWER_CAPABILITIES

This function is not implemented for the keypad driver.

17.4.2.4 IOCTL_POWER_SET

This function is not implemented for the keypad driver.

17.4.2.5 IOCTL_POWER_GET

This function is not implemented for the keypad driver.

17.4.3 Keypad Registry Settings

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

```
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\KEYBD]
    "CalVKey"=dword:0
    "ContLessVKey"=dword:0
    "ContMoreVKey"=dword:0
    "TaskManVKey"=dword:2E
    "Keyboard Type"=dword:4
    "Keyboard SubType"=dword:0
    "Keyboard Function Keys"=dword:0
    "Keyboard Layout"="00000409"
    "DriverName"="KbdUs.dll"

[HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Control\Layouts\00000409]
    "Layout File"="KbdUs.dll"
    "Layout Text"="US-Keypad"
    "KPPLayout"="KbdUs.dll"

[HKEY_CURRENT_USER\Keyboard Layout\Preload\4]
    @="00000409"
```

17.5 Unit Test

As keypad only has only 16 keys; it is not a full-key keypad. It cannot pass the Keyboard Test included in the Windows CE Test Kit (CETK). A manual test to verify the 16-key functionality is described in following sections.

.

17.5.1 Unit Test Hardware

- i.MX25 3DS board

17.5.2 Unit Test Software

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

17.5.3 Building the Unit Tests

No additional steps are required to build the keypad tests.

17.5.4 Running the Unit Tests

The procedure of keyboard tests is as follows:

1. Run Microsoft WordPad application
2. Input Number key
3. Input Tab
4. Input Space
5. Input Del
6. Run the Internet Explorer application
7. Selcet the File menu
8. Input Up Down Left and Right
9. Quit Microsoft WordPad, there is a pop up dialog box, click the Yes button

NOTE

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

17.6 Keypad Driver API Reference

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

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

17.6.1 Keypad PDD Functions

Table 17-4 shows a mapping of the keyboard PDD functions to the functions used in the keypad driver:

Table 17-4. Keypad PPD Functions

PDD Function Pointer	Keypad Driver Function
PFN_KEYBD_PDD_ENTRY	KPP_Entry
PFN_KEYBD_PDD_GET_KEYBD_EVENT	KeybdPdd_GetEventEx2
PFN_KEYBD_PDD_POWER_HANDLER	KPP_PowerHandler

Chapter 18
Liquid Crystal Display Controller (LCDC) Driver

The LCDC display driver is based on the Microsoft DirectDraw Graphics Primitive Engine (DDGPE) classes. This driver combines the functionality of a standard LCD display with DirectDraw support. The display driver interfaces with the i.MX25 Liquid Crystal Display Controller (LCDC). This module is designed to be compatible with the CHUNGHWA CLAA057VA01CT(VGA) LCD panel.

18.1 Display Driver Summary

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

Table 18-1. Display Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX25-3DS-PDK1_7
Target SOC (TGTSOC)	MX25_FSL_V2_PDK1_7
SOC Common Path	..PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\LCDC
SOC Specific Path	..PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\LCDC
Platform Specific Path	..\PLATFORM*Target Platform*>\SRC\DRIVERS\LCDC
Import Library	ddgpe.lib, gpe.lib
Driver DLL	lcdc.dll
Catalog Items	Third Party >BSP > Freescale <TGTPLAT>: ARMV4I > Device Drivers > Display > CHUNGHWA CLAA057VA01CT(VGA)
SYSGEN Dependency	SYSGEN_DDRAW=1
BSP Environment Variables	BSP_NODISPLAY=

18.2 Supported Functionality

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

1. Derives from the DirectDraw Graphics Primitive Engine (DDGPE) class
2. Supports the DirectDraw Hardware Abstraction Layer (DDHAL)
3. Supports the CHUNGHWA CLAA057VA01CT(VGA) LCD panels
4. Supports two power management modes, full on and full off

18.3 Hardware Operation

Refer to the chapter on the LCDC in the *i.MX25 Multimedia Applications Processor Reference Manual* for detailed operation and programming information.

18.3.1 Conflicts with Other SoC Peripherals

No conflicts.

18.4 Software Operation

18.4.1 Communicating with the Display

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

18.4.1.1 Using the GDI

The Graphics Device Interface provides basic controls for the display of text and graphics. For instructions on using the GDI, see the Help:

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

18.4.1.2 Using DirectDraw

The DirectDraw API provides support for hardware-accelerated 2-D graphics, offering fast access to display hardware while retaining compatibility with the GDI. For information on using the DirectDraw API, see the Help:

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

18.4.1.3 Using Display Driver Escape Codes

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

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

18.4.2 Configuring the Display

The display is configured based on the **PanelType** registry key, which is described in Section 18.4.4, “Display Registry Settings”. The **PanelType** registry key indicates the display panel that is being used. There is currently one supported display: Chunghwa VGA LCD panel.

18.4.3 Rotation Support

The DirectDraw display driver may be configured to allow screen rotation, through [HKEY_LOCAL_MACHINE\SYSTEM\GDI\ROTATION] in the platform.reg file.

NOTE

> Due to lack of support for DirectDraw and screen rotation (see the Windows Embedded CE 6.0 Help, stating that “GDI screen rotation cannot be used with DirectDraw”), a DirectDraw display driver with rotation support enabled may yield failures in the GDI CETK test suite.

18.4.4 Display Registry Settings

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

```
[HKEY_LOCAL_MACHINE\Drivers\Display\LCDC]
        "PanelType"=dword:0        ; 0: Chunghwa VGA
```

18.4.5 Power Management

The display driver consumes power primarily through the operation of the LCDC, which combines and displays video and graphics data, and through the operation of the display panel. To facilitate management of these modules, the display driver implements the power management I/O Control (IOCTL) code IOCTL_POWER_SET.

18.4.5.1 PowerUp

This function is not implemented for the display driver.

18.4.5.2 PowerDown

This function is not implemented for the display driver.

18.4.5.3 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 – Clock gating is enabled for all clocks to the LCDC. The display panel is enabled. The LCDC modules are enabled.
- D4 – The LCDC are disabled. The display panel is disabled. Clock gating is disabled for all clocks to the LCDC.

18.5 Unit Test

The display driver is subject to two test suites provided with the Windows Embedded CE 6.0 Test Kit (CETK): the Graphics Device Interface (GDI) Test and the DirectDraw Test. Additionally, video playback may be verified by using the Windows Media Player application.

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

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

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

18.5.1 Unit Test Hardware

Table 18-2 lists the required hardware to run the GDI and DirectDraw tests.

Table 18-2. Hardware Requirements

Requirement	Description
Chunghwa VGA Panel	Display panel required for display of graphics data

18.5.2 Unit Test Software

18.5.2.1 GDI Tests

Table 18-3 lists the required software to run the GDI tests.

Table 18-3. 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
Gdiapi.dll	Main test .dll file
Ddi_test.dll	Graphics Primitive Engine (GPE)–based display driver that the GDI API uses to verify the success of each test case. If Ddi_test.dll is unavailable, run the test with manual verification.

18.5.2.2 DirectDraw Tests

Table 18-4 lists the software required to run the DirectDraw tests.

Table 18-4. Direct Draw Software Requirements

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

18.5.2.3 Windows Media Player Tests

Table 18-5 lists the software required to perform WMV playback with Windows Media Player.

Table 18-5. Windows Media Player Software Requirements

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

18.5.3 Building the Display Tests

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

Windows Embedded CE Test Kit > **Running the CETK**

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

18.5.4 Running the Display Tests

18.5.4.1 Running the GDI Tests

The command line for running the GDI tests is `tux -o -d gdiapi.dll -c "NoResize"`.

For detailed information on the GDI tests and command line options for these tests, see the Platform Builder Help:

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

Table 18-6 describes the test cases contained in the GDI test suite.

Table 18-6. GUI Test Cases

Test Case	Description
100–104: Clip	Tests the functionality of clipping using different shapes and verifies the functionality of complex clip regions
200–231: Draw	Calls functions that draw and functions that apply complex effects to drawing. These test cases perform blitting, line drawing, filling, color table manipulation, bitmap type creation, and device attribute modification **NOTE**: Test case 231, AlphaBlend, is skipped, as it is not supported in the image
300–307: Palette	Verifies color matching and color conversion for palettes, and modifies associated palettes
500–512: Region	Tests region management by calling functions that modify region rectangles
600–608: Brush and pen	Assesses the functionality of brushes and brush alignments
700–710: Device attribute	Verifies device attributes and exercises functions that modify device attributes
800–808: Device context	Creates, retrieves, saves, and restores a device context
900–905: Device object	Calls functions that retrieve, modify, and delete GDI objects
1000–1011: Font	Verifies font enumeration, selection, and attributes
1100–1108: Text	Writes text to various locations on the display. If the font required by the test is not available, some test cases do not run
1200–1205: Print	Passes bad parameters to printing functions
1300–1303: Verify	Assesses the functionality of test verification functions such as **CheckScreenHalves** and **CheckAllWhite**
1400, 1401: Manual	Manually tests font drawing. These test cases can be used to exercise code paths. To step through these test cases, press the left SHIFT key

NOTE

The GDI test cases may be interfered by the mouse movement. So it is preferred to remove Shell and User Interface > User Interface > Mouse catalog from OS workspace before run GDI tests.

18.5.4.2 Running the DirectDraw Tests

The command line for running the DirectDraw tests is `tux -o -d ddrawtk`.

For detailed information on the DirectDraw tests and command line options for these tests, see the Platform Builder Help:

Windows EmbeddedCE Test Kit > **CETK Tests and Test Tools** > **CETK Tests** > **Display Tests** > **DirectDraw Test**

Table 18-7 describes the test cases contained in the DirectDraw test suite.

Table 18-7. Direct Draw Test Cases

Test Case	Description
100: Get Caps	Retrieves the capabilities of the hardware abstraction layer (HAL), verifies that the operation is successful, and then displays all capabilities retrieved. This test case fails if it cannot retrieve the capabilities of the HAL.
101: Enumerate Display Modes	Enumerates the DirectDraw display modes, verifies that the enumeration completes, and then displays the results of the enumeration.
200: Blt (Windowed Mode)	Executes blits to/from various surfaces. Verifies that the actual destination matches the expected destination for each blit. This test case fails if any blits are unsuccessful.
210: ColorKey Blt (Windowed Mode)	Executes a variety of color key blits to/from assorted surfaces. Verifies that the actual destination matches the expected destination for each test. This test case fails if any color key blits are unsuccessful.
220: Color Filling Blts (Windowed Mode)	Executes a variety of color fill blits to assorted surfaces. The test case verifies that the surface is filled with the specified color. This test case fails if any color fill blits are unsuccessful.
300: Blt (Exclusive Mode)	Executes a variety of blits to and from assorted surfaces. The test case verifies that the actual destination matches the expected destination for test. This test case fails if any blits are unsuccessful.
310: ColorKey Blt (Exclusive Mode)	Executes a variety of color key blits to and from assorted surfaces. The test case verifies that the actual destination matches the expected destination for each test. This test case fails if any color key blits are unsuccessful.
320: Color Filling Blts (Exclusive Mode)	Executes a variety of color fill blits to assorted surfaces. The test case verifies that the surface is filled with the specified color. This test case fails if any color fill blits are unsuccessful.
330: Flip (Exclusive Mode)	Executes a variety of blits to a flipping chain and verifies that the flips are successful and that all surfaces display correctly. This test case fails if any flips or surface verifications fail.
400: CreateVideoPort (Video Port Container Test)	Enumerates the available video ports and connections for the video ports. This test case then verifies that each enumerated connection can be created.
410: EnumVideoPorts (Video Port Container Test)	Enumerates the available video ports and then enumerates the video ports based on specific capabilities.
420: GetVideoPortConnectInfo (Video Port Container Test)	Verifies the **GetVideoPortConnectInfo** function appropriately handles a variety of input conditions.
430: QueryVideoPortStatus (Video Port Container Test)	Verifies that the **QueryVideoPortStatus** function appropriately handles each video port.
500: GetBandwidthInfo (Video Port Test)	Verifies that the **GetBandwidthInfo** function returns consistent information about each type of video port available.
502: GetSetColorControls (Video Port Test)	Verifies that the **GetColorControls** and **SetColorControls** functions set and return consistent color controls if color control is supported.
504: GetInputOutputFormats (Video Port Test)	Verifies that the **GetInputFormats** and **GetOutputFormats** functions return consistent information under a variety of calling conditions.
506: GetFieldPolarity (Video Port Test)	Verifies that the **GetFieldPolarity** function returns consistent information about the field polarity of a video port.
508: GetVideoLine (Video Port Test)	Verifies that the **GetVideoLine** function returns consistent information.
510: GetVideoSignalStatus (Video Port Test)	Verifies that the **GetVideoSignalStatus** function returns consistent information.

Table 18-7. Direct Draw Test Cases (continued)

Test Case	Description
512: SetTargetSurface (Video Port Test)	Verifies that the **SetTargetSurface** function appropriately handles improper input.
514: StartVideo (Video Port Test)	Verifies that calls to the **StartVideo** function succeed with a variety of flags set.
516: StopVideo (Video Port Test)	Verifies that a call to the **StopVideo** function succeeds.
518: UpdateVideo (Video Port Test)	Verifies that calls to the **UpdateVideo** function succeed with a variety of flags set.
520: WaitForSync (Video Port Test)	Verifies that the **WaitForSync** function behaves consistently with each possible flag.
1200: Blt (Interactive Windowed Mode)	Executes blits that include color fills and scaling. Each blit must be verified manually.
1240: Overlay Blt (Interactive Windowed Mode)	Executes blits to an overlay surface. The test cannot verify that the contents of the overlay surface display correctly, so the output must be verified manually. If the test succeeds, an overlay appears in the middle of the screen with a blue and yellow checkerboard pattern. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.
1250: ColorKeyOverlay Blt (Interactive Windowed Mode)	Executes color key blits to an overlay surface. The test cannot verify that the contents of the overlay surface display correctly, so the output must be verified manually. If the test succeeds, a blue checkerboard pattern appears in a variety of locations across the display of the target device. If the driver enables stretching, the test also stretches the checkerboard pattern. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.
1260: ColorFill Overlay Blt (Interactive Windowed Mode)	Executes color fill blits to an overlay surface. The test cannot verify that the contents of the overlay surface display correctly, so the output must be verified manually. This test case cycles through red, green and blue on the primary overlay surface. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.
1300: Blt (Interactive Exclusive Mode)	Executes blits that must be verified when prompted. The test does not contain an automated mechanism for verifying scaling.
1340: Overlay Blt (Interactive Exclusive Mode)	Executes blits to an overlay surface. The test cannot verify that the contents of the overlay surface display correctly, so the output must be verified manually. If the test succeeds, an overlay is located in the middle of the screen on the target device with blue and yellow horizontal lines. The primary surface behind the overlay surface should fill the entire display. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.
1350: ColorKeyOverlay Blt (Interactive Exclusive Mode)	Executes color key blits to an overlay surface. Cannot verify that the contents of the overlay surface display correctly, so the output must be verified manually. If the test succeeds, blue horizontal bars appear across the display of the target device. The primary surface behind the overlay surface should fill the entire display. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.
1360: ColorFill Overlay Blt (Interactive Exclusive Mode)	Executes color fill blits to an overlay surface. Cannot verify that the contents of the overlay surface display correctly, so the output must be verified manually. This test case cycles through red, green and blue on the primary surface and the overlay surface. The color fills on the primary surface should fill the entire display. This test may run as many as three times with the same output, testing RGB, YUYV, and VYUY pixel formats on the overlay surface.

NOTE

The following DirectDraw test cases are not supported by the DirectDraw driver, and are therefore skipped: 400, 410, 420, 430, 500, 502, 504, 506, 508, 510, 512, 514, 516, 518, 520.

18.5.4.3 Running the Windows Media Player tests

The command line for starting playback of a WMV test video clip in Windows Media Player is `ceplayer [wmv test file]` (for example, `ceplayer motocross_208x160_30fps.wmv`). If audio support is not included in the current BSP, a dialog box reading “Audio hardware is missing or disabled” pops up when the WMV file is being loaded. Select OK to continue to WMV playback.

Correct operation of this test is confirmed by observing the application and verifying that the video clip is playing at a smooth rate (it should not be dropping frames or otherwise appear jerky) with a clear image, normal coloring, and correct image sizing.

NOTE

Because overlay destination color keying is not enabled due to hardware limitations, Windows Media Player uses GDI memory copy for video playback. This means that video clip playback usually drops frames and is not smooth for test videos via WMV.

18.6 Display Driver API Reference

Documentation for the display driver APIs can be found within the Platform Builder Help. No additional custom API information is required for the features currently supported in the display driver. Reference information on basic display driver functions, methods, and structures can be found at the following location in the Platform Builder Help documentation:

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

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

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

Chapter 19
NAND Flash Driver

The NAND Flash Driver provides the functionality of NAND storage accessing. The flash driver follows Windows CE 6.0 R2 Flash driver MDD and PDD model.

19.1 Flash Driver Summary

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

The flash driver supports both SLC and MLC NAND Flash devices. As for page size, 512 byte (small page size) is not supported. Table 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	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\NAND
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\NAND
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\BLOCK\NANDFMD ..\PLATFORM\<*Target Platform*>\SRC\COMMON\NANDFMD
Driver DLL	flashpdd_nand.dll
SDK Library	N/A
Catalog Item	Device Drivers > Storage Devices > MSFlash Drivers > Flash MDD Third Party > BSP > Freescale i.MX25 3DS PDK1_7: ARMV4I > Storage Drivers > MSFlash Drivers > Samsung K9LBG08U0D NAND Flash
SYSGEN Dependency	N/A

BSP Environment Variables	BSP_NONANDFMD=
	BSP_NAND_K9LAG08U0M=1
	BSP_NAND_K9LBG08U0M=1
	BSP_NAND_K9LBG08U0D=1

19.2 Supported Functionality

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

1. Supports the Windows CE MDD/PDD interface
2. Supports both MLC and SLC NAND
3. Supports both 2 Kbyte and 4 Kbyte page size NAND
4. Supports MLC NAND Flash K9LBG08U0D as default

19.3 Hardware Operation

Refer to the chapter on the NAND Flash Controller (NFC)in the *i.MX25 Multimedia Applications Processor Reference Manual* for detailed operation and programming information.

19.3.1 Conflicts with Other Peripherals and Catalog Items

19.3.1.1 Conflicts with SoC Peripherals

No conflicts.

19.4 Software Operation

The development concepts for flash media drivers are described in the Windows CE 6.0 Help Documentation section under the topic

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

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

19.4.1 MDD/PDD Layer Overview

The Microsoft Windows Embedded CE 6.0 Flash Driver component contains two components: the module device driver (MDD), and the platform-dependent driver (PDD).

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

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

The block diagram below (Figure 38-1) describes the high level architecture and basic interactions of the i.MX NAND driver implementation. The i.MX flash driver PDD consists of three major components:

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

The i.MX flash driver currently supports a limited number of commercially available flash modules. However, the i.MX flash driver software architecture allows support of new flash modules. The i.MX flash driver must be modified to support new flash modules that the BSP does not currently support.

The i.MX flash driver is table driven. That is, by appropriately modifing definitions described in this chapter, the flash driver can be reconfigured to support a different flash module. No other source code changes are required.

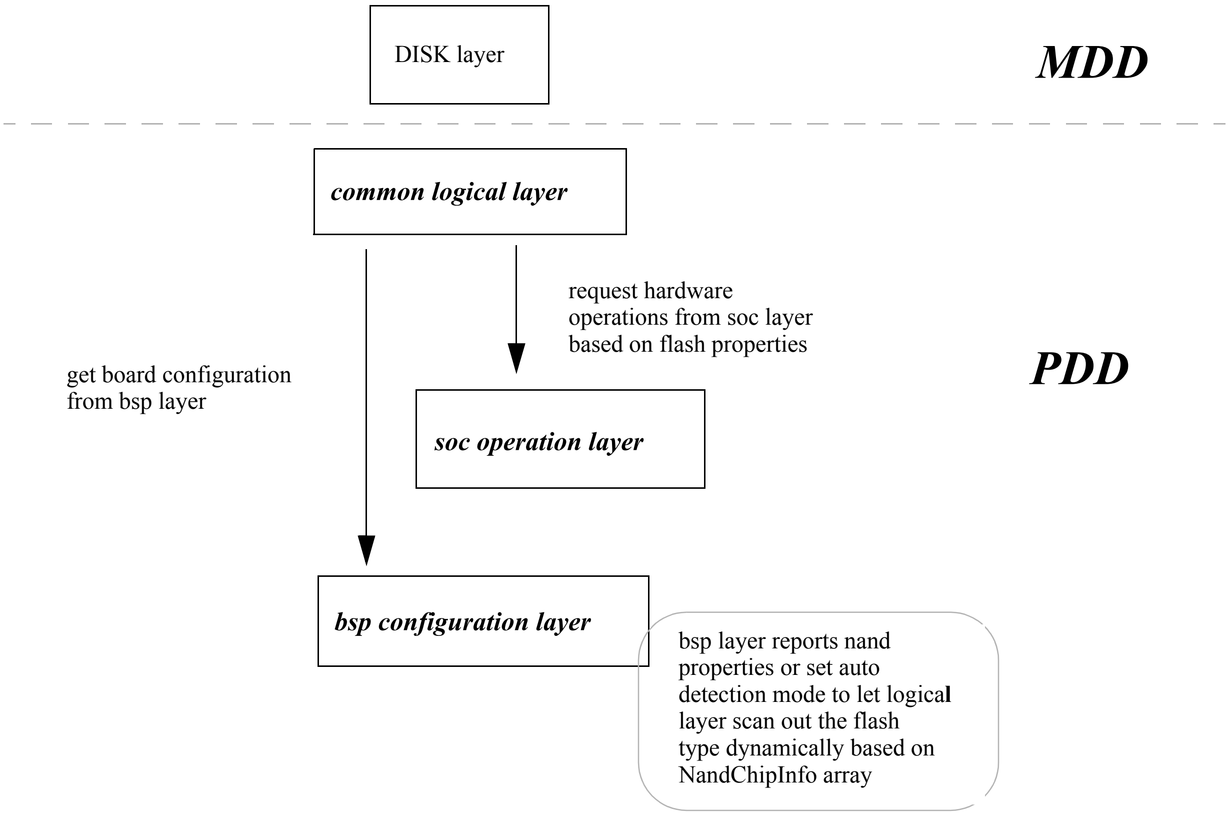
DISK layer
MDD
common logical layer
request hardware operations from soc layer based on flash properties
get board configuration from bsp layer
PDD
soc operation layer
bsp configuration layer
bsp layer reports nand properties or set auto detection mode to let logical layer scan out the flash type dynamically based on NandChipInfo array

Figure 19-1. PDD Layer Block Diagram

19.4.2 Definitions

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

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

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

The i.MX flash driver abstracts the characteristics of the NAND Flash memory device to a single header file. This header file is named in terms of the NAND device name, could be found in:

`\WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2`_PDK1_7**`\NAND\INC`**

For each NAND flash device, there are two header files, one is .h format used for C language code and another is .inc format used for assembly language code. The two files are highly similar.

Table 19-2. Nand Property Definitions

Member	Description
CMD_READID	Command used to read nand flash ID. Default value is 0x90.
CMD_READ	Command used as initial command for reading operation. Default value is 0x00.
CMD_READ2	Command used as start command for reading operation. Default value is 0x30.
CMD_RESET	Command used to reset nand flash. Default value is 0xFF.
CMD_WRITE	Command used as initial command for writing operation. Default value is 0x80.
CMD_WRITE2	Command used as start command for writing operation. Default value is 0x10.
CMD_ERASE	Command used as initial command for erasing operation. Default value is 0x60.
CMD_ERASE2	Command used as start command for erasing operation. Default value is 0xD0.
CMD_STATUS	Command used to read nand flash status. Default value is 0x70.
NAND_STATUS_BUSY_BIT	Bit number in status byte to indicate BUSY/IDLE status of nand flash status. Default value is 6.
NAND_STATUS_ERROR_BIT	Bit number in status byte to indicate PASS/FAIL status of nand flash operation. Default value is 0.
NAND_NUM_OF_CS	Number of dies per nand flash chip.
NAND_BLOCK_CNT	Number of blocks per nand flash die.
NAND_PAGE_CNT	Number of pages per block.
NAND_PAGE_SIZE	Number of bytes in main data area per page.
NAND_SPARE_SIZE	Number of bytes in spare area per page.

Table 19-2. Nand Property Definitions (continued)

Member	Description
NAND_BUS_WIDTH	Bit number of nand flash, it should be 8bits or 16bits.
NAND_MAKER_CODE	Nand ID defined in terms of manufacturer.
NAND_DEVICE_CODE	Nand ID defined in terms of nand type.
BBI_NUM	Number of pages, defined by manufacturer, that is used to indicate initial bad block during manufacturing.
BBMarkPage[BBI_NUM]	An array that indicates which pages are used to indicate initial bad block during manufacturing.
BBI_MAIN_ADDR	Byte address that is used to swap data with bad block mark to guarantee BBI position is untouched. For 2K+64B page size nand, define it as 464; For 4K+128B page size nand, define it as 400; For 4K+218B page size nand, define it as 330.
NAND_PAGE_CNT_LSH	The power exponent (log2) of NAND_PAGE_CNT. For exsample, it should be 7 if page number is 128 (2^7). The definition is defined in .inc file.
NAND_PAGE_SIZE_LSH	The power exponent (log2) of NAND_PAGE_SIZE. For exsample, it should be 12 if page size is 4096 (2^12). The definition is defined in .inc file.
NAND_BLOCK_SIZE_LSH	The power exponent (log2) of nand flash block size. The definition is defined in .inc file.
NUM_OF_NAND_DEVICES	Number of nand flash dies per nand flash chip. The definition is defined in .inc file.
NUM_OF_NAND_DEVICES_LSH	The power exponent (log2) of NUM_OF_NAND_DEVICES. The definition is defined in .inc file.

19.4.3 Adding New Flash Configurations

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

The flash driver currently supports the flash modules listed in section 38.2. The flash driver does not dynamically detect which flash module is installed. Instead, the flash driver must be modified and re-compiled to support a different flash module

To support a flash module that is not currently supported by the BSP, use the following steps:

1. create two new header files (.h and .inc format files) by using one of the existing NAND device headers as templates
2. update the newly created header file to include device-specific definitions described in 38.4.2
3. update the reference in the `nandbsp.h` device-specific header located in the following directory:
   `\WINCE600\PLATFORM\<Target Platform>\SRC\COMMON\NANDFMD`
4. recompile the flash driver for the new device

NOTE

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

19.4.4 Registry Settings

The registry keys implemented for the flash driver provide basic support for loading and configuring the NAND as a file system mount. Many more configuration options are available and are discussed in: Windows CE 6.0 Help Documentation section under the topic

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

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

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

The two items can be deleted to disable the functionality.

19.4.5 DMA Support

The flash driver does not support DMA.

19.4.6 Power Management

Flash driver handles power requests in MDD layer by default.

19.5 Unit Test

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

19.5.1 CETK Testing

NOTE

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

Table 19-3. CEKT Tests

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

19.5.2 System Testing

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

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

Chapter 20
Pulse Width Modulator (PWM) Driver

The PWM controller outputs a square wave with a variable duty cycle.

20.1 PWM Driver Summary

Table 20-1 provides a summary of source code location, library dependencies and other BSP information.

Table 20-1. PWM Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\PWM
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\PWM
Import Library	none
Driver DLL	pwm.dll
Catalog Item	Third Party > BSPs > Freescale <*TGTPLAT*>: ARMV4I > Device Drivers > PWM
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_PWM1=1 or BSP_PWM2=1 or BSP_PWM3=1 or BSP_PWM4=1

20.2 Supported Functionality

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

1. Supports duty cycle modulation
2. Supports varying output intervals
3. Supports two power management modes, full on and full off

20.3 Hardware Operation

The PWM driver outputs the toggling signal whose frequency and duty cycle can be modulated by programming the appropriate registers.

20.3.1 Conflicts with Other Peripherals and Catalog Items

20.3.1.1 Conflicts with SoC Peripherals

The i.MX25 platform contains four PWM modules. PWM1 is conflicted with the interrupt from the Debug board. Therefore, this PWM can be used if the Debug board is not attached. PWM2 is used for USB OTG. PWM3 is used for USB OTG. PWM4 is conflicted with the LCD and CAN.

20.3.2 Conflicts with 3-Stack Peripherals

PWM1 is used for Debug board interrupts.

20.4 Software Operation

20.4.1 Registry Settings

20.4.1.1 i.MX25 Registry Settings

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

```
IF BSP_PWM1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PWM1]
   "Prefix"="PWM"
   "Dll"="pwm.Dll"
   "Order"=dword:1
   "Index"=dword:1
ENDIF  ; BSP_PWM1

IF BSP_PWM2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PWM2]
   "Prefix"="PWM"
   "Dll"="pwm.Dll"
   "Order"=dword:1
   "Index"=dword:2
ENDIF  ; BSP_PWM2

IF BSP_PWM3
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PWM3]
   "Prefix"="PWM"
   "Dll"="pwm.Dll"
   "Order"=dword:1
   "Index"=dword:3
ENDIF  ; BSP_PWM3

IF BSP_PWM4
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PWM4]
   "Prefix"="PWM"
   "Dll"="pwm.Dll"
   "Order"=dword:1
   "Index"=dword:4
ENDIF  ; BSP_PWM4
```

20.4.2 Communicating with the PWM

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

20.4.3 Creating a Handle to the PWM

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

To open a handle to the PWM:

1. Insert a colon after the PWM port for the first parameter, *lpFileName*. For example, specify PWM1:.
2. Specify OPEN_EXISTING in the *dwCreationDisposition* parameter. This flag is required.

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

```
// Open the serial port.
hPWM = CreateFile (L”PWM1:”,                     // name of device
            GENERIC_READ | GENERIC_WRITE,   // desired access
            0,                              // sharing mode (ignored)
            NULL,                           // security attributes (ignored)
            OPEN_EXISTING,                  // creation disposition
            0,                              // flags/attributes
            NULL);                          // template file (ignored)
```

20.4.4 PWM Operations

The PWM uses the PWM_Write function to output a user defined waveform and the PWM_Read the registers of the PWM controller.

To read the registers of the PWM, use the ReadFile API as in the following example:

```
ReadFile (hPWM,                    // handle returned by CreateFile
        lpBuffer,                  // pointer to open buffer at least count bytes long
        NumberOfBytesToRead,       // must be at least 24
        lpNumberOfBytesRead,       // pointer. Will receive number of bytes written
        NULL);                     // lpOverlapped (ignored)
```

The PWM has two registers that are used to modify the output waveform. These registers are the period register and the sample register. When an output waveform is started, an internal counter starts counting from zero and the output is a low. When the internal counter is equal to the sample register, the output changes to high. When the internal counter is equal to the period register + 1, the counter and output reset to zero and the counter starts counting again. The duty cycle can be modified by changing the sample register. If the sample register is greater than the period register + 1, no waveform is generated. For more detailed information, consult the datasheet for the i.MX25.

To output a waveform use the WriteFile API as in the following example:

```
PwmSample_t PwmSample;
PwmSample.duration = 5000;          //waveform lasts for 5 secs
PwmSample.period = 0;               //period of 1
PwmSample.sample = 1;               //output will change when counter reaches 1
WriteFile(hOpen,                    //handle returned by CreateFile
        &PwmSample,                 //pointer to a PwmSample_t structure
        sizeof(PwmSample_t),        //size of the PwmSample_t structure
        &NumberOfBytesWritten,      //pointer to a DWORD
        NULL); //ignored
```

20.4.5 Closing the Handle to the PWM

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

20.4.6 Power Management

The primary method for limiting power consumption in the PWM module is to gate off the input clock to the module when the input PWM clock is not needed. This is accomplished through the **DDKClockSetGatingMode** function call.

20.4.6.1 PowerUp

This function resets the device.

20.4.6.2 PowerDown

This function is not implemented for the PWM driver.

20.4.6.3 IOCTL_POWER_SET

This function is implemented for the PWM driver. When D4 or D3 power mode is set, the driver turns off the clocks to the controller.

20.4.6.4 Restrictions

- The DMA XCH size is restrict to maximum DMA buffer size, up to 5 Kbytes bytes
- DMA exchange only supports a bit count of 32
- The maximum transfer size if using SPI1 is 64 Dwords
- Cannot use polling and DMA for the same transaction

20.5 Unit Test

The testing of the PWM requires a hardware modification and therefore is not easy for the user to test.

20.6 PWM Driver API Reference

20.6.1 PWM Driver IOCTLS

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

20.6.1.1 PWM_IOCTL_RESET

This **DeviceIoControl** request performs the transfer of data to a target device. An SPI_XCH_PKT object is required, which contains PWM 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	None

20.6.2 PWM Driver Structures

20.6.2.1 PWM_BUSCONFIG_T

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

```
/* struct of pwm play sample */
typedef struct {
   UINT32 sample;              // pwm sample value
   UINT32 period;              // pwm period
   UINT32 duration;            // duration of the sample (msec)
} PwmSample_t, *pPwmSample_t;
```

Members

sample	Count that the output negates
period	Period -1 of the waveform in clock cycles
duration	Length in milliseconds that the output waveform endures

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.

Table 21-1. Serial Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\SERIAL
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\SERIAL
Driver DLL	csp_serial.dll
SDK Library	N\A
Catalog Item	Third Party –> BSP –> Freescale <*Target Platform*>: ARMV4I – > Device Drivers –> Serial –> UART2
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_SERIAL_UART2 =1

21.2 Supported Functionality

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

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

21.3 Hardware Operation

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

21.3.1 Conflicts with Other Peripherals and Catalog Items

21.3.1.1 Conflicts with SoC Peripherals

All the pins of UART can be configured for alternate functionality (FEC, SDIO, GPT1, EPIT2) using the i.MX25 IOMUX. The configuration used by the serial driver does not have any conflict with other peripherals. Changing this configuration results in a conflict and prevents proper operation of the UART.

21.3.1.2 Conflicts with Board Peripherals

No conflicts.

21.4 Software Operation

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

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

21.4.1 Registry Settings

21.4.1.1 i.MX25 Registry Settings

```
IF BSP_SERIAL_UART2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2]
   "DeviceArrayIndex"=dword:0
   "IoBase"=dword:43F94000
   "IoLen"=dword:D4
   "Prefix"="COM"
   "Dll"="csp_serial.dll"
   "Index"=dword:2
   "Order"=dword:4
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\COM2\Unimodem]
   "Tsp"="Unimodem.dll"
   "DeviceType"=dword:0
   "FriendlyName"="i.MX25 COM2 UNIMODEM"
   "DevConfig"=hex: 10,00, 00,00, 05,00,00,00, 10,01,00,00, 00,4B,00,00, 00,00, 08, 00, 00,
00,00,00,00
ENDIF ;BSP_SERIAL_UART2
```

21.4.2 Power Management

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

21.5 Unit Test

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

21.5.1 Unit Test Hardware

- i.MX25 3DS board

21.5.2 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
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
SerDrvBvt.dll	Test .dll file for Serial Port Driver Test

21.5.3 Building the Unit Tests

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

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

21.5.4 Running the Unit Tests

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

For detailed information on the Serial Port tests, see

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

The serial port tests are designed to test that the serial port driver works properly and the API behaves correctly, and it should be pass all the test cases. Table 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.
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.

21.6 Serial Driver API Reference

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

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

21.6.1 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

Table 21-4. Serial PDD Functions

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

21.6.2 Serial Driver Structures

21.6.2.1 UART_INFO

This structure contains information about the UART Module.

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

Members

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

TransmitCritSec	This value is used as Critical Section for UART registers
RegCritSec	This value is used as Critical Section for UART
ChipID	This value contains Chip identifier (CHIP_ID_16550 or CHIP_ID_16450)

21.6.2.2 SER_INFO

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

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

Members

uart_info	This structure contains information about UART
fIRMode	This boolean value determines the module is FIR or serial
dwDevIndex	This static value contains the device index value which is read from registry
dwIOBase	This static value contains the I/O Base address of UART module which is read from registry
dwIOLen	This static value contains the I/O length of UART Module which is read from registry
pBaseAddress	Pointer to the start address of the UART registers mapped
cOpenCount	Contains count of the concurrent open
CommProp	Pointer to CommProp structure

pHWObj	Pointer to PDDs HWObj structure
useDMA	This boolean flag indicates if SDMA is to be used for transfers through this UART
SerialDmaReqTx	SDMA request line for Tx
SerialDmaReqRx	SDMA request line for Rx
SerialPhysTxDMABufferAddr	Physical address of Tx SDMA address
SerialPhysRxDMABufferAddr	Physical address of Rx SDMA address
pSerialVirtTxDMABufferAddr	Virtual address of Tx SDMA address
pSerialVirtRxDMABufferAddr	Virtual address of Rx SDMA address.
SerialDmaChanRx	SDMA virtual channel indices for Rx
SerialDmaChanTx	SDMA virtual channel indices for Tx
currRxDmaBufId	Index of the buffer descriptor next expected to complete its SDMA in the Rx SDMA buffer descriptor chains
currTxDmaBufId	Index of the buffer descriptor next expected to complete its SDMA in the Tx SDMA buffer descriptor chains
dmaRxStartIdx	Keeps the start index of byte to be delivered to MDD for Read
availRxByteCount	This variable keeps the remaining bytes in the Rx SDMA buffer
awaitingTxDMACompBmp	Indicates if an SDMA request is in progress on Tx SDMA buffer descriptor
dmaTxBufFirstUseBmp	Indicator for first time use of a Tx SDMA buffer descriptor
rxDMABufSize	Receive DMA buffer size
txDMABufSize	Transfer DMA buffer size

Chapter 22
Subscriber Identification Module (SIM) Driver

The SIM is a 32-bit peripheral used to communicate with the SIM cards. It has one port and can have direct control over one card. The SIM driver transfers the data between SIM module and the SIM card. These data transfers follow the smartcard transmission protocol as defined in the ISO-7816 specification.

22.1 SIM Driver Summary

Table 22-1 provides a summary of source code location, library dependencies and other BSP information.

Table 22-1. SIM Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\SIM
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\SIM
Platform Specific Path	..\PLATFORM\<*Target Platform*>\DRIVERS\SIM
Driver DLL	sim.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <*TGTPLAT*> > Device Drivers > SIM
SYSGEN Dependency	SYSGEN_SMARTCARD=1

BSP Environment Variables	BSP_NOSIM=
	BSP_SMARTCARD_SIM1=1
	BSP_SMARTCARD_SIM2=1

22.2 Supported Functionality

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

1. Implements ISO-7816 protocol specification
2. Supports the SIM module for GSM SIM Cards

22.3 Hardware Operation

Refer to the chapter on the SIM in the *i.MX25 Multimedia Applications Processor Reference Manual* for detailed operation and programming information.

22.3.1 Conflicts with Other Peripherals and Catalog Items

22.3.1.1 Conflicts with SoC Peripherals

All the pins of SIM1 can be configured for alternate functionality using the i.MX25 IOMUX. Hence SIM1 conflicts with both ESAI and CSI controllers. The configuration is specified by the BSP serial driver. Changing this configuration results in a conflict and prevents proper operation of the SIM.

22.3.1.2 Conflicts with 3-Stack Peripherals

On i.MX25 3-Stack System, the pins used for communication with the SIM1 external connector is shared with the ESAI and the CSI controllers. Do not include the camera and ESAI Audio drivers while including SIM1 in the image.

22.4 Software Operation

The SIM driver uses the APIs of smartcard driver and follows the Microsoft-recommended architecture for smart card drivers. The details of this architecture and its operation can be found in the Platform Builder Help at the following location:

Developing a Device Driver > **Windows Embedded CE Drivers** > **Smart Card Drivers** > **Smart Card Driver Development Concepts**.

22.4.1 Power Management

The power management is currently not implemented for the SIM driver.

22.4.2 SIM Registry Settings

The following registry keys are required to properly load the SIM driver into the image.

```
IF BSP_SMARTCARD_SIM1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SIM1]
    "Prefix"="SCR"
    "Dll"="sim.dll"
    "Index"=dword:1
    "Order"=dword:11
    "DeviceArrayIndex"=dword:1
    "Port"=dword:0
ENDIF   ;BSP_SMARTCARD_SIM1

IF BSP_SMARTCARD_SIM2
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SIM2]
    "Prefix"="SCR"
    "Dll"="sim.dll"
    "Index"=dword:2
    "Order"=dword:11
    "DeviceArrayIndex"=dword:2
    "Port"=dword:0
ENDIF   ;BSP_SMARTCARD_SIM2
```

SIM driver is a variance of the smartcard driver and uses the smartcard layer provided by Microsoft as its upper layer. Thus, to use the SIM driver, it is required to first include the smartcard layer into the OS design and then build the image.

22.5 Unit Test

There is no Windows CE 6.0 Test Kit (CETK) available to test the SIM driver. However, the driver can be tested by running the SIM Application present in the **WINCE600\SUPPORT_PDK1_7\APP\SIMApp folder**.

22.5.1 Unit Test Hardware

Table 22-2 lists the required hardware to run the unit tests.

Table 22-2. Hardware Requirements

Requirement	Description
GSM SIM card	Any GSM SIM card

22.5.2 Building the SIM Application

To build the SIM Application, complete the following steps: Build an OS image for the desired configuration:

1. Within VS2005, go to the **Build** menu option and select the **Open Release Directory in Build Window** menu option.This opens a DOS prompt.
2. Change to the SIM Application directory: \WINCE600\SUPPORT_PDK1_7\APP\SIMApp)
3. Enter **set WINCEREL=1** on the command prompt and press return. This copies the built EXE to the flat release directory.
4. Enter the build command (*build -c*) at the prompt and press return.

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

22.5.3 Running the SIM Application

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

Once the SIM application is loaded, a screen appears on the display containing three drop down menus in the tool bar: **Tools**, **About** and **File**. By now, the SIM application has sent three SELECT_FILE commands to the SIM card to open the Master File (MF), Dedicated File (DF) which contains telecom services features and an Elementary File (EF). Once the EF is opened, the application can issue commands to parse through the records in this file.

The user can choose the **Tools** option from the toolbar and then choose the **Phonebook** option. Here the user can parse through the phonebook entry, add a new entry into the phonebook or remove an existing entry.

The user can move to the previous record by clicking on the Previous button, the next record by clicking on the Next button. If there is no data in a selected record, the application prompts a message referring to an empty record.

To add/modify an entry into the SIM, it is first required to edit the entry and then click on the Modify button which saves the modifications. The number entry can only accept number characters, any other character input causes modification operation failed.

Table 22-3 shows the functions present in the application that are called when user performs any operation in the **Phonebook** option in the GUI.

Table 22-3. Phonebook Functions

Function	Description
SIMParsePBRecord	Used to parse the phone book data from the record
SIMAssemblePBRecord	Assembles the phone book record
SIMDataExchange	Used to exchange data (apdu) between the card reader and the SIM card using T0 protocol
SIMStatus	Gets currently selected file’s status, which specifies the allowed operation on that file
SIMSelectFile	Uses the SELECT_FILE command to select a file present in the SIM card
SIMReadRecord	Uses the READ_BINARY command to read a string of bytes from a selected file
SIMUpdateRecord	Uses the UPDATE_RECORD command to update one complete record in the selected file

22.6 SIM Driver API Reference

Detailed reference information for the SIM driver API may be found in VS2005 Help at the following location:

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

22.6.1 SIM PDD Functions

The SIM driver is developed from the sample PC Card smart card readers namely Bulltlp3.dll, Pscr.dll, and Stcusb.dll that are included in the Software Development Kit (SDK).

22.6.2 SIM Driver Structures

22.6.2.1 ClockRateFactor

This structure is used to store the Clock Rate Conversion Factor table defined in ISO-7816 specification.

```
typedef struct {
        UINT8 FI;
        INT16 Fi;
        INT16 f;
} ClockRateFactor;
```

Members

FI	Bit form representation of Clock Rate Conversion Factor as received from SIM Answer To Reset (ATR)
Fi	Clock Rate Conversion Factor
f	Maximum clock frequency in MHz

22.6.2.2 ClockRateFactor

```
typedef struct {
        UINT8   DI;
        UINT8   Di;
}BaudRateFactor;
```

Members

DI	Bit form representation of Baud Rate Adjustment Factor as received from SIM ATR (Answer To Reset)
Di	Baud Rate Adjustment Factor

Chapter 23
Touch Panel Driver

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

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	iMX25-3DS-PDK1_7
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\TOUCH
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\TOUCH
Driver DLL	touch.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX25 3DS PDK1_7:ARMV4I > Device Drivers > TOUCH > Touchscreen
SYSGEN Dependency	SYSGEN_TOUCH = 1
BSP Environment Variables	BSP_NOTOUCH=

23.2 Supported Functionality

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

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

23.3 Hardware Operations

The hardware consists in a 4-wire touchscreen, an internal ADC controller and an internal touch screen controller. The touch screen controller configures the ADC driver as required to do the measurement of the X and Y values of the touch screen.

For each acquisition, four samples are acquired for X and for Y. The first of each is discarded and the three remaining samples are filtered (if one of the samples is too far from the mean value of this acquisition, it is also discarded).

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

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

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

For every calibration point, perform the following steps:

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

23.4.1 Touch Driver Registry Settings

```
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]
    "DriverName"="touch.dll"
    "MaxCalError"=dword:10

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

;For TouchPannel calibration
[HKEY_LOCAL_MACHINE\init]
    "Launch80"="touchc.exe"
    "Depend80"=hex:14,00, 1e,00
```

23.5 Unit Tests

23.5.1 Unit Test Hardware

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

Table 23-2. Hardware Requirements

Requirement	Description
LCD panel	Display panel required for display of graphics data

23.5.2 Unit Test Software

Table 23-3 lists the required software 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
Touchtest.dll	The Test.dll File
Touch.dll	Touch Panel Driver

NOTE

The touch driver does work after the CETK Touch Panel Test. This is a known MSFT CETK issue. The MSFT online help notes that “When you complete the test, the operating system does not regain control of the touch panel. You must reset the touch panel to restore normal operation.” Refer to **CETK Tests and Test Tools** > **CETK Tests** > **Touch Panel Tests**

Cases 8011, 9001–9003 fail. The touch panel shows several lines when a circle or arc is drawn. This is also a known MSFT CETK issue. All these points are captured, but are not painted in time.

Case 8011 cannot draw in the right part of screen after a 90° rotation. ethca.exe works after rotation and the CETK works when the case runs again.

23.5.3 Running the Touch Panel Tests

The touch panel test cases can be run by typing:

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

The test case IDs are described in the documentation at:

Windows Embedded CE Test Kit > **CETK Tests and Test Tools** >**CETK Tests** > **Touch Panel Tests** > **Touch Panel Test**

23.6 Touch Panel API Reference

The complete API reference is given in the documentation at:

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

Chapter 24
Universal Serial Bus (USB) OTG Driver

The OTG USB driver provides High Speed USB 2.0 host and device support for the USB On The Go (OTG) port of the i.MX. The OTG driver automatically selects either host or device functionality at any given time, depending on the USB cable/mini-plug configuration. This is achieved by a set of three drivers: USB OTG host controller driver, USB client driver and/or USB transceiver controller (Full Function) driver, which performs the host/function client switching.

The USB host driver can be configured for class support for mass storage, HID, printer, and RNDIS peripherals. The device/client portion can be configured to provide mass storage, serial, or RNDIS function. The Full Function OTG transceiver driver automatically selects between the host or client driver. The host or client can also be configured as the only mode for the OTG port, using the Pure Host or Pure Client catalog item. All the OTG catalog items are exclusive. (See Section 24.1, "USB OTG Driver Summary.").

24.1 USB OTG Driver Summary

24.1.1 USB OTG Client Driver Summary

Table 24-1 provides a summary of source code location, library dependencies and other BSP information for the USB OTG client driver.

Table 24-1. OTG Client Driver Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
Common SOC	COMMON_FSL_V2
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<*Target S*OC>\USBD ..\PLATFORM\COMMON\SRC\SOC\<*Common Soc*>\ms\USBFN
CSP Static Library	usb_usbfn_<*Target SOC*>_PDK1_7.lib usb_usbfn_os_<*Target SOC*>_PDK1_7.lib usb_ufnmddbase_<*Common Soc*>_PDK1_7.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBD
Import Library	N/A
Driver DLL	usbfn.dll

Table 24-1. OTG Client Driver Summary (continued)

Driver Attribute	Definition
Catalog Item	High Speed OTG: Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device To support only client/device mode, choose .. > High Speed OTG Port Pure Client Function
SYSGEN Dependency	SYSGEN_USBFN=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSOTG_CLIENT=1

USB clients require a function driver to be loaded. A client can only present one function. Only one of the function drivers (described in Section 24.5.5, "Function Drivers,") should be configured through drag and drop. If more than one is configured, the serial function is the default unless the registry is manually modified.

24.1.2 OTG Host Driver Summary

Table 24-2 provides a summary of source code location, library dependencies and other BSP information for the USB OTG host driver.

Table 24-2. OTG Host Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX25-3DS-PDK1_7
Target SOC (TGTSOC)	MX25_FSL_V2_PDK1_7
Common SOC	COMMON_FSL_V2
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<*Common SOC*>\ms\USBH\EHCI ..\PLATFORM\COMMON\SRC\SOC\<*Common SOC*>\ms\USBH\EHCIPDD ..\PLATFORM\COMMON\SRC\SOC\<*Common SOC*>\ms\USBH\USB2COM
CSP Static Library	usbh_ehcdmdd_<*Common SOC*>_PDK1_7.lib usbh_ehcdpdd_<*Common SOC*>_PDK1_7.lib usbh_usb2com_<*Common SOC*>_PDK1_7.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBH\HSOTG
Import Library	N/A
Driver DLL	hcd_hsotg.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device To support only host mode, choose .. >High Speed OTG Port Pure Host Function.
SYSGEN Dependency	SYSGEN_USB=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSOTG_HOST=1

Host driver requires a set of class drivers to be loaded. See Section 24.5.6, "Class Drivers," for class driver information.

24.1.3 OTG Transceiver Driver Summary (For High-Speed Only)

Table 24-3 provides a summary of source code location, library dependencies and other BSP information for the USB OTG transceiver driver.

Table 24-3. OTG Transceiver Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX25-3DS-PDK1_7
Target SOC (TGTSOC)	MX25_FSL_V2_PDK1_7
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\USBXVR
CSP Static Library	usb_xvc_<*Target SOC*>_PDK1_7.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBXVR
Import Library	N/A
Driver DLL	imx_xvc.dll
Catalog Item	Third Party > BSPs > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed OTG Device > High Speed OTG Port Full OTG Function Support
SYSGEN Dependency	SYSGEN_USBFN=1

BSP Environment Variable	BSP_NOUSB=
	BSP_USB_HSOTG_CLIENT=1
	BSP_USB_HSOTG_HOST=1
	BSP_USB_HSOTG_XVC=1

24.2 USB Host Driver Summary

Table 52-4 provides a summary of source code location, library dependencies and other BSP information for the HS host driver.

24.2.1 FS Host2 Driver Summary

Table 24-4. HS Host2 Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	MX25-3DS-PDK1_7
Target SOC (TGTSOC)	MX25_FSL_V2_PDK1_7
CSP Driver Path	..\SOC\<*Common SOC*>\ms\USBH\EHCI ..\SOC\<*Common SOC*>\ms\USBH\EHCIPDD ..\SOC\<*Common SOC*>\ms\USBH\USB2COM
CSP Static Library	usbh_ehcdmdd_<*Common SOC*>.lib usbh_ehcdpdd_<*Common SOC*>.lib usbh_usb2com_<*Common SOC*>.lib
Platform Driver Path	\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBH\HSH2
Import Library	N/A
Driver DLL	hcd_hsh2.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed Host To support only host mode, choose .. >High Speed Host Function.
SYSGEN Dependency	SYSGEN_USB=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSH2=1

Host driver requires a set of class drivers to be loaded. See Section 24.5.6, “Class Drivers for class driver information.

Notes: The H2 host on MX25 is acturally a FULL SPEED Host controller.

24.3 Supported Functionality

The OTG driver provides the following software and hardware support:

1. High Speed OTG/Host driver supports USB specification 2.0.
2. Configured as client/peripheral by default, with one function driver defined as default. When nothing is connected to the OTG port, the port does not drive Vbus and awaits attachment to a host by raising its D+ signal. On attachment of a mini-A plug the driver switches to host mode.
3. When a mini-B plug is connected to the OTG port, and the cable opposite end is connected by a mini-A (or A-type) plug to a PC, then the OTG initiates operation as peripheral and responds to USB protocol from the host.

4. When a mini-A plug is connected to the OTG port and the cable opposite end is connected by a mini-B plug to another OTG device, then the OTG initializes/re-initializes itself into host mode and begin to act as a host. The OTG port remains in host mode whenever a mini-A plug is mated to the OTG socket connector.
5. OTG port as client/peripheral supports mass storage, RNDIS and serial clients
6. OTG port as host or HS Host supports mass storage, HID and RNDIS classes
7. When nothing is attached to the OTG port, the driver configures the controller and transceiver into a low power state
8. When the system is suspended with nothing attached to the OTG/Host port, the system does not create a wake condition upon attachment of the port to a host or attachment of a device with mini-A plug
9. When the system is suspended while the OTG/Host port is connected to a host or to a device with a mini-A plug, the system remains suspended when the OTG port connection is unplugged
10. When the system resumes after suspend, any attached devices are enumerated and their class drivers loaded appropriately
11. Data transfer rates on the client port exceeds 40 Mbits/sec in Mass Storage client

24.4 Hardware Operation

The USBOH module contains all of the functionality required to support one USB port, compatible with the USB 2.0 specification. In addition to the normal USB functionality, the module also provides support for direct connections to on-board USB peripherals with Serial, UTMI or ULPI protocol, and supports multiple interface types for ULPI and Serial Transceivers.

In addition to the USB cores, the module provides for Full-Speed Transceiverless Link (TLL) operation on the host port and the OTG core also supplies the UTMI interface for the internal UTMI PHY.

24.4.1 Conflicts with Other Peripherals and Catalog Items

24.4.1.1 Conflicts with SoC Peripherals

No conflicts.

24.4.1.2 Conflicts with Board Peripherals

The USB OTG Host conflicts with the USB Host only port. The polarity of the USB_VBUS_EN signal is inverted between the two and there is only one polarity selection bit in the USB controller.

24.5 Software Operation

24.5.1 USB OTG Host Controller Driver

This driver enables the USB host functionality for the OTG port. It is part of the standard Windows USB software architecture as shown in Figure 24-1.

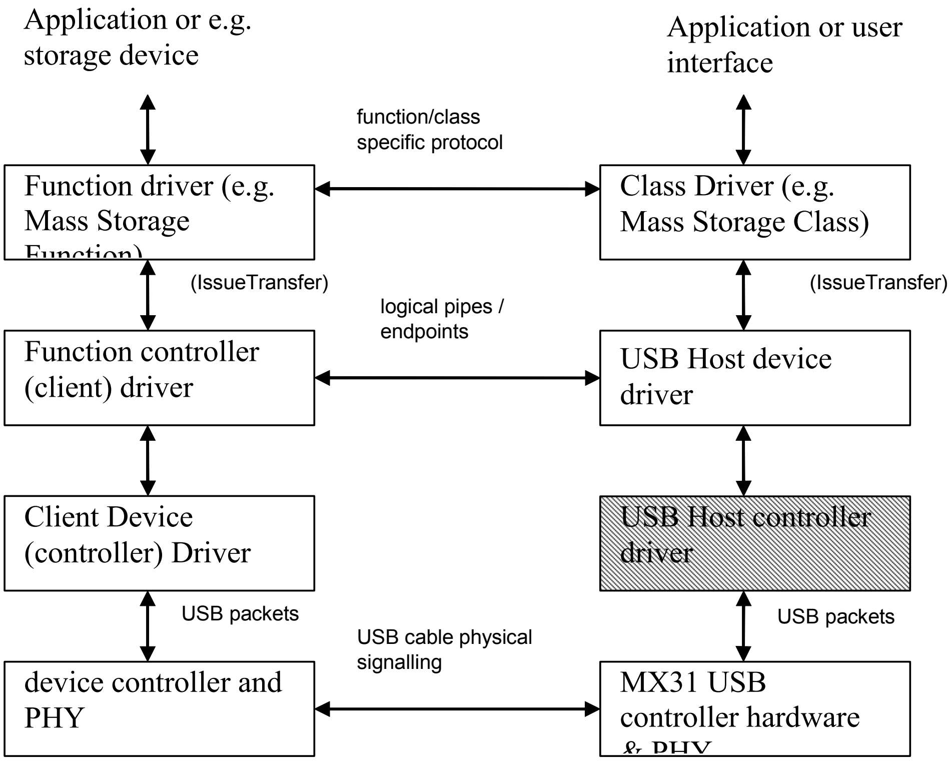
Application or e.g. storage device
Application or user interface
function/class specific protocol
Function driver (e.g. Mass Storage Function)
Class Driver (e.g. Mass Storage Class)
(IssueTransfer)
logical pipes / endpoints
(IssueTransfer)
Function controller (client) driver
USB Host device driver
Client Device (controller) Driver
USB Host controller driver
USB packets
USB cable physical signalling
USB packets
device controller and PHY
MX31 USB controller hardware & PHY

Figure 24-1. Windows USB Driver Architecture

For further details of the Windows CE USB driver architecture and usage, see the Platform Builder Windows CE 6.0 help topic:

Developing a Device Driver > **Windows Embedded CE Drivers** > **USB Host Drivers**

and

Developing a Device Driver > **Windows Embedded CE Drivers** > **USB Host Drivers** > **USB Host Controller Drivers** > **USB Host Controller Driver Development Concepts**

When transceiver mode is included, the host driver is activated when a USB Mini-A plug is connected to the Mini USB OTG socket. When Pure Host mode only is selected, the host driver is always in control of the relevant USB controller. When a USB device is connected to the Mini USB OTG socket, the host controller driver enumerates and activates the appropriate class driver (see Section 24.5.1, “USB OTG Host Controller Driver,”).

The BSP supports the following USB class drivers:

- Mass Storage—SD cards, CF cards, HDD drive, thumb drive (disk-on-key); some card reader firmware is not supported by the Microsoft standard Mass Storage class driver
- HID—Keyboard and mouse
- RNDIS—Network Device Interface communication class

Hubs are supported in all configurations with Full and Low Speed peripherals.

24.5.1.1 User Interface

User access to the USB host driver is by class drivers. For further details on these Host Client Drivers refer to the Windows CE 6.0 Platform Builder help topic:

Developing a Device Driver > **Windows Embedded CE Drivers** > **USB Host Drivers** > **USB Host Controller Drivers** > **USB Host Client Drivers.**

Where new class driver code is to be developed, refer to the Host client driver interface functions (for example IssueBulkTransfer) as documented in:

Developing a Device Driver > **Windows Embedded CE Drivers** > **USB Host Drivers** > **USB Host Controller Drivers** > **USB Host Client Drivers** > **Host Client Driver Reference.**

24.5.1.2 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 24.5.1.5, "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.MX25 Applications Processor Reference Manual*.

24.5.1.3 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.

24.5.1.4 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 24.5.1.8.1, "BSPCheckConfigPower," and located in:

```
\PLATFORM\<Target Platform>\SRC\DRIVERS\USBH\Common\hwinit.c
```

This function must be modified to correspond with the platform hardware capabilities.

The i.MX system can supply a total of 100 mA to attached devices on the OTG port and the default behavior does not need to be modified. All bus powered hubs that have been tested require 500 mA and therefore are not supported for use. Self-powered hubs are required to expand the number of USB sockets and also to support devices that require greater than 100 mA.

24.5.1.5 Registry Settings

The USB OTG host controller settings are values located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\HCD_HSOTG]
```

The values under this registry key are automatically included in the image by `platform.reg`. They do not normally require customization. Table 24-5 shows the default values contained in `hsotg.reg`.

Table 24-5. hsotg.reg Default Values

Value	Type	Content	Description
Dll	sz	hcd_hsotg.dll	Driver dynamic link library
OTGSupport	dword	01	This value must be set to 1 to enable host driver on the OTG. If no host support is required (client only) then this value can be set to 0, though the HCD_HSOTG key is not normally configured in the image at all when pure Host function is selected.
OTGGroup	sz	01	This unique string (example “00” to “99”) is used to combine/correlate instances of the host, function, and transceiver driver within one USB OTG instance.
HcdCapability	dword	4	HCD_SUSPEND_ON_REQUEST. Note: HCD_SUSPEND_RESUME is always assumed.
PhysicalPageSize	dword	20000	This value represents the number of bytes allocated for the physical memory pool of the OTG host driver, and defaults to 128 Kbytes. From this buffer, 75% are allocated for transfer descriptors and the remaining buffer is available for allocation to simultaneous transfers. In most cases, only one transfer is active at any time (for example, in the Mass Storage Class). A good value is at least 3x as large as the largest data buffer transferred using IssueTransfer(). This key is optional, if it does not exists in the registry, it takes the default value, otherwise a specific value can be assigned.

24.5.1.6 Host USB Test Modes

The USB 2.0 specification defines PHY-level test modes for the USB host ports (see definitions in USB 2.0 specification section 7.1.20). The i.MX USB host drivers support packet test mode. The test mode is configured by compiling the BSP with the compilation flag OTG_TEST_MODE defined within `bsp_cfg.h`:

```
#define OTG_TEST_MODE
```

This configures the appropriate host controller within the platform-specific hardware initialization function (ConfigOTG()), located in:

```
\PLATFORM\<Target Platform>\SRC\DRIVERS\USBH\Common\hwinit.c
```

The test mode is entered upon initialization, and cannot be exited. Normal USB operation is disabled when test mode support is compiled into the image.

24.5.1.7 Unit Test

The USB driver has many devices to be tested. Tests are performed manually and include connecting the devices, and confirming the attach, detach (on unplug) re-attach (on subsequent plug in of device), and transferring and verifying data (and/or functions).

To verify the RNDIS class device, a CEPC containing Netchip 2280 USB function is attached and used to access a remote file server on the CEPC. To verify the low-level transport for Bulk, Interrupt and Isochronous transfers, the CETK Host test kit is performed. This requires a CEPC configured with Netchip 2280 USB function and loopback driver.

24.5.1.7.1 USB Host Controller Driver Test

Documentation for the Windows CE 6.0 CETK USB Host tests is normally found under the Platform Builder Windows CE product documentation:

Debugging and Testing > **Windows CE Test Kit** > **CE Test Kit**

24.5.1.7.2 Build the Test Image

The following steps are used to build the image to be tested:

1. Checkout the RTM to be tested or install the MSI provided
2. Add the following components from the catalog:
    - Freescale <Target Platform> :ARMV4I-Device Drivers-USB Devices-USB High Speed Host1-High Speed Host 1
    - Core OS > Windows CE devices > Core OS Services > USB HOST Support; and all the sub-components of this catalog item (Sub-Components like USB Storage Class Driver.)
    - Core OS > Windows CE devices > File Systems And Data store > Storage Manager; (Sub-Components: FAT File System, Partition Driver, Storage Manager Control Panel Applet)
    - Device Drivers > USB Function > USB Function Clients-Serial.
3. Sysgen and build the image

24.5.1.7.3 Abstract

This test suite can be used to test USB host controller drivers that provide the same interface as Windows CE USB host controller driver does (for more information, see Section 24.5.1.1, “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 24-2.

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.

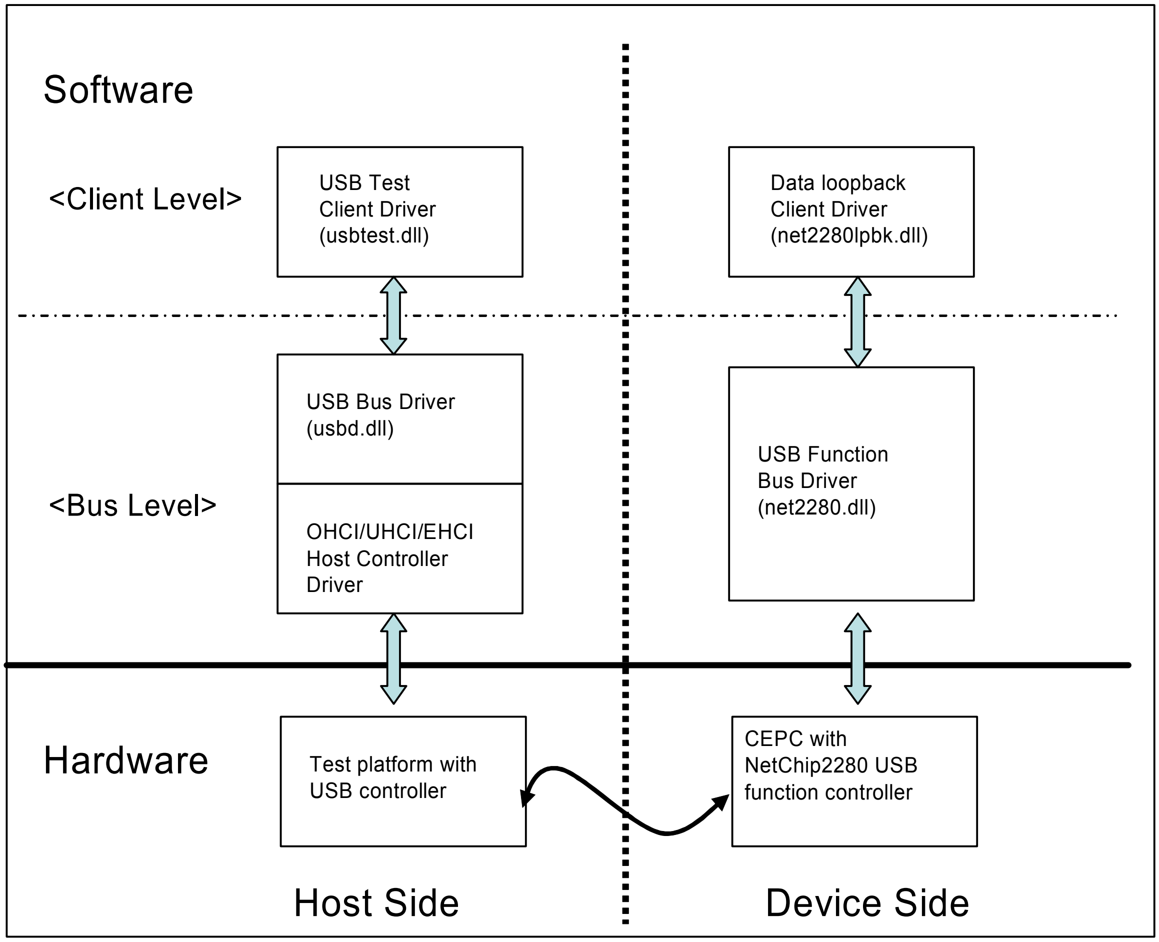
Software
<Client Level>
USB Test Client Driver (usbtest.dll)
Data loopback Client Driver (net2280lpbk.dll)
USB Bus Driver (usbd.dll)
<Bus Level>
OHCI/UHCI/EHCI Host Controller Driver
USB Function Bus Driver (net2280.dll)
Hardware
Test platform with USB controller
CEPC with NetChip2280 USB function controller
Host Side
Device Side

Figure 24-2. Test Setup

24.5.1.7.4 Unit Test Hardware

- Test platform
- Host Controller Card (if not onboard logic)
- CEPC
- Netchip2280 Card
- USB cable

24.5.1.7.5 Unit Test Software

Host side requirements:

- Tux.exe
- Ddlx.dll
- Usbtest.dll
- Tooltalk.dll
- Kato.dll
- USB component (usbd.dll, EHCI/OHCI/UHCI host controller driver(s)) must be included in the run time image

Device side requirements:

- Lufldrv.exe
- Net2280lpbk.dll
- NetChip2280 USB function support (net2280.dll) must be included in the CEPC run time image

24.5.1.7.6 Running the Test

The test procedure is as follows:

1. Download the runtime image to the CEPC (Windows Embedded CE PC-based hardware platform) with the Netchip2280 card on it
2. After the system is booted up, run `s lufldrv`, the tester should verify that `net2280lpbk.dll` is loaded
3. Download the runtime image to the test platform with a USB host controller on it
4. After the system is booted up, make sure there is no connection between the host side and the device through the USB cable. Then launch command `s tux -o -d ddlx -c “usbtest” “-xYYYY”`, where `YYYY` is the test case(s) to be run
5. The test indicates that there should be no connection between host and device side. Then after seven seconds, the test asks to connect two sides with a USB cable
6. The test main body starts to run
7. After test(s) is(are) done, and if other tests in the test suite are to be run, do not disconnect the two sides of the USB cable. Type the next test command, and the tests starts directly. If the USB connection was disconnected before the next test, the tests asks to make the connection again before the test begins

24.5.1.7.7 Test Cases

Table 24-6 shows the test cases contained in the test suite.

Table 24-6. USB Host Controller Driver Test Cases

Test Case ID	Test Description
1001-1315, 1501-1515	Data loopback tests: Concerning the transfer type, there are five categories: 1) Bulk pipe loopback tests (tests with ID end with 1, like xxx1) 2) Interrupt pipe loopback tests (tests with ID end with 2, xxx2) 3) Isochronous pipe loopback tests (tests with ID end with 3, xxx3) 4) All pipe transfer simultaneously (tests with ID end with 4, xxx4) 5) All three types transfers carry on simultaneously (tests with ID end with 5, xxx5) [1] There are five categories for how data is transferred: 1) Normal loopback tests (tests with ID start with 10, like 10) 2) loopback tests using physical memory (tests with ID start with 11, 11xx) 3) loopback tests using a part of allocated physical memory (tests with ID start with 12, 12xx) 4) Normal short transfer loopback tests (tests with ID start with 13, 13xx) 5) Stress short transfer loopback tests (tests with ID start with 15, 15xx) Also both synchronous and asynchronous transfer methods are exercised (test cases like xx1x using asynchronous transfer method, test cases like xx0x using synchronous method There are a total of 5×5×2 = 50 test cases
1401-1413	Additional data loopback tests. that mainly focus on testing APIs like GetTransferStatus(), AbortTransfer() and CloseTransfer()
2001-2013	Test related to Device requests
9001-9004	Special tests that test APIs such as SuspendDevice(), ResumeDevice() and DisableDevice()
9005	Test that stresses EP0 transfer (Vendor Transfer)

[1] This category of tests is designed for testing some other USB function devices which have more endpoints than host controller driver can handle. When using Netchip2280, it should be the same as category 4). Tester can just ignore this category.

By default, the data loopback device configures the endpoints with some often-used packet sizes that are DWORD aligned, and neither too big nor too small. By having all tests in Table 24-6 pass under this configuration is more than sufficient for a BVT-type test pass. However, testers can change the packet sizes (these values are hard-coded in the source code for `net2280lpbk.dll`) for each endpoint by themselves and run these test cases again for more comprehensive testing.

This test suite provides a way to change packet sizes of on NetChip2280 device on the fly. They are:

- Test case 3001—Using very small packet sizes in NetChip2280 device full speed configuration
- Test case 3002—Using very small packet sizes in NetChip2280 device high speed configuration
- Test case 3003—Using irregular packet sizes (like non DWORD-aligned size) in NetChip2280 device full speed configuration
- Test case 3004—Using irregular packet sizes (like non DWORD-aligned size) in NetChip2280 device high speed configuration

- Test case 3005 (High Speed only)—Using very large packet sizes (like 2×1024 for Isochronous endpoints) in NetChip2280 device full speed configuration. In the real world, Netchip2280 cannot handle transfers using such large packet size because its onboard FIFO buffer is small

Run one of the test case above, then after 15–20 seconds, `usbtest.dll` is unloaded and loaded again automatically through the Platform Builder. The packets sizes on netchip2280 side have already been changed. Then those normal tests can be run. Use test case 3011 (for full speed config) and 3012 (for high speed) to restore the default packet sizes.

Another category test that is important for USB2.0 host controllers and drivers is called the golden bridge tests, which means USB2.0 host controller is connected with a full speed (USB1.1) device. This is the only scenario that USB2.0 host controller performs split transfers.

NetChip2280 can be forced to be a full speed device. In the test setup stage, instead of run `s lufldrv` to load loopback driver, run `s lufldrv -f`. This forces the Netchip2280 to be configured as a full speed device.

Also testers are encouraged to do some manual tests. Here are some examples:

- Plug in real USB devices, suspend system, and then resume; USB devices should still be there
- Plug in real USB devices, suspend system, unplug it, plug in another device, then resume; system should enumerate that new device properly
- Run one of the data transfer tests, in the middle of transfer stage, suspend the system (host side), then resume; tests may fail, but system should not crash
- Run one of the data transfer tests, in the middle of transfer stage, disconnect the USB connection; tests should fail, but system should not crash

24.5.1.8 Platform-Specific API

This section describes the platform-specific API functions.

24.5.1.8.1 BSPCheckConfigPower

This function is used to evaluate whether a device can be supported on the specified USB port.

Parameters

UCHAR bPort	[in] Unused. Each USB controller has only one port
DWORD dwCfgPower	[in] Power requirement (number of milliamps) requested by the device being evaluated for attachment support on this port
DWORD dwTotalPower	[in] current total power (number of milliamps) used by other previously attached devices on this port
Return Value	Return TRUE if device requesting dwCfgPower can be safely attached Return FALSE if device can not be attached

24.5.1.8.2 BSPUsbSetWakeUp

This function enables or disables the wakeup on the USB port. This function does not actually enable wake-up when a device is currently attached to the port.

Parameters

BOOL bEnable	[in] TRUE to enable wakeup, FALSE to disable wakeup

24.5.1.8.3 BSPUsbCheckWakeUp

This function evaluates the wake-up condition for the relevant USB port, and clears the condition and interrupt.

Parameters	None
Return Value	Return TRUE when a wake-up condition was detected Return FALSE when no wake-up condition was present

24.5.1.8.4 SetPHYPowerMgmt

This function is called by the USB driver when transitioning to or from the suspended state (for example, during system suspend). The function does what is necessary to place the transceiver hardware into a suspended (fSuspend = TRUE) or running (fSuspend = FALSE) state.

The standard implementation for a i.MX system uses a ULPI-bus based ISP1504 transceiver for the HS OTG port, and this function configures the ULPI-bus for sleep state. If platform hardware uses other transceivers, this function must be modified appropriately.

Parameters

BOOL fSuspend	[in] TRUE: system/controller is going to suspend mode. FALSE: resuming

24.5.2 USB Client Driver

This driver enables the USB device functionality for the i.MX device. It is activated when a USB Mini B connector is connected to the Mini USB OTG socket. When the i.MX System is connected to a USB host system (for example, high speed or full speed port of PC), it is enumerated according to the current configuration settings, and the appropriate class driver is loaded on the PC. By default the system is configured for USB serial class. The system can be configured as one of the following USB functions by setting the appropriate environment variable during build (drag/drop from the catalog):

- Serial class—Serial ActiveSync
- Mass storage—expose local storage (ATA hard disk, RAMDISK or other store) as USB drive
- RNDIS class—Remote Network Driver Interface Specification

24.5.2.1 User Interface

The USB client driver provides a standard Windows CE USB driver implementation. For an overview see:

Developing a Device Driver > **Windows Embedded CE Drivers** > **USB Function Drivers** > **USB Function Controller Drivers.**

User access to the USB client driver is through function drivers such as Mass Storage or RNDIS. For further details on these USB Function drivers, refer to the Windows CE 6.0 Platform Builder help topic:

Developing a Device Driver > **Windows Embedded CE Drivers** > **USB Function Client Drivers.**

Where new function driver code is to be developed, refer to the Function controller driver interface functions (for example, IssueTransfer) as documented in:

Developing a Device Driver > **Windows Embedded CE Drivers** > **USB Function Controller Drivers** > **USB Function Controller Driver Reference.**

24.5.2.2 Client Driver Configuration

The OTG client driver is configured into the BSP build by dragging and dropping the appropriate catalog item (see Section 24.1.1, "USB OTG Client Driver Summary,"). When the Pure Client functionality is selected, the OTG port acts only as a device. When Full OTG functionality is selected, the OTG port can be either device or host (see transceiver driver configuration).

24.5.2.3 Registry Settings

The USB OTG function/client settings are values located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\UFN]
```

The values under this registry key are automatically included in the image through platform.reg. They do not normally require customization. Table 24-7 shows the USB OTG client registry settings.

Table 24-7. USB OTG Client Registry Settings

Value	Type	Content	Description
Dll	sz	usbfn.dll	Driver dynamic link library
OTGSupport	dword	01	This value must be set to 1 to enable the function/client on the OTG. If no client support is required (host only) then this value can be 0, though the UFN key is not normally configured in the image at all when pure Host function is selected
OTGGroup	sz	01	This unique string (example 00 to 99) is used to combine/correlate instances of the host, function, and transceiver driver within one USB OTG instance

24.5.2.4 Device USB Test Modes

The USB 2.0 specification defines PHY-level test modes for USB device ports (see definitions in USB 2.0 specification section 7.1.20). This mechanism allows a host to configure a device into test mode by commanding the device with a specific SET_FEATURE request. Once test mode is entered, the device is not able to leave test mode. The device port does not by default support the USB test modes. Sample code for test mode support (SET_FEATURE on the device) is included in:

```
..\PLATFORM\COMMON\SRC\SOC\<Target SOX>\MS\USBFN\CONTROLLER\MDD
```

In addition, USBFN_TEST_MODE_SUPPORT must be defined during compilation of the CSP USBD device driver library.

24.5.2.5 Unit Test

There is no CETK test case for USB client (function) drivers. The USB function is tested by configuring the i.MX system as either USB serial function, USB mass storage or RNDIS function and connecting it directly to a host PC. The test verifies basic USB peripheral/client functionality, including attach, detach, and data transfer. Separate images must be built and downloaded for each of the three peripheral function tests.

24.5.2.5.1 Unit Test Hardware

Table 24-8 lists the required hardware to run the unit tests.

Table 24-8. Hardware Requirements

Requirement	Description
Host system	To test if control reaches the Host controller driver
USB cable having Mini USB OTG plug A at one end and Mini USB OTG plug B on the other side	For connecting between the host and the device
ATA drive configured in UDMA mode 2 as DSK1	Required as a storage device when the board is configured as mass storage class

24.5.2.5.2 Unit Test Software

Table 24-9 shows the software requirements for the USB Function controller driver test.

Table 24-9. Software Requirements

Requirement	Description
ActiveSync 4.1 and above	Host side software that is required to be available for testing the Serial class functionality

24.5.2.5.3 Running the USB Function Controller Driver Tests

Table 24-10 lists USB function controller driver tests.

Table 24-10. USB Function Controller Driver Tests

Test Cases	Entry Criteria/Procedure/Expected Results
Board configured as USB Serial class and connected to a host system after the board boots up completely	Entry Criteria: Make sure there is no mini USB OTG plug B is connected and the board is turned on and wait until the board boots-up completely Procedure: 1. Connect the mini USB OTG plug B to the mini USB OTG socket 2. Observe that the ActiveSync on the host side gets connected and is synchronized 3. Copy files from Host system to the Mobile Device. Files are copied 4. Copy files from the Mobile Device to the Host system. Files gets copied 5. Unplug the mini USB OTG plug B from the i.MX mini USB OTG socket to unload the Serial class driver Expected Result: ActiveSync should get synchronized and copying of files should happen between the Host and the System
Board configured as USB Mass storage client, with ATA drive as DSK1 mounted, and connected to a host system after the board boots up completely	Entry Criteria: Make sure there is no mini USB OTG plug B is connected and the board is turned on and wait until the board boots-up completely Procedure: 1. Connect the mini USB OTG plug B to the mini USB OTG socket 2. Observe that a new disk in My Computer having as Removable Disk appearing in it 3. Copy files from Host system to the new disk drive. Files are copied 4. Copy files from the new disk drive to the Host system. Files gets copied 5. Unplug the mini USB OTG plug B from the mini USB OTG socket to unload the mass storage class driver Expected Result: Files copied into mass storage client device match those copied out (when compared on Windows XP PC using file compare utility). Note that files are not visible from within the System until the system has been reset. The file system should not be used inside the System when it is being accessed via USB as a mass storage client.
Board configured as USB RNDIS client and connected to a host system after the board boots up completely. Browsing the Internet	Entry Criteria: Make sure there is no mini USB OTG plug B is connected and the board is turned on and wait until the board boots-up completely. See to it that the NIC's local area connection is not having any IP address Procedure: 1. Connect the mini USB OTG plug B to the mini USB OTG socket 2. Observe that a new Local area connection in the Network and Dial up connections appears on the Windows XP machine. Bridge the NIC's local area connection and the RNDIS's local area connection 3. Configure the bridge by giving IP address, Subnetmask, Default gateway, DNS 4. On the System, a new Local area connection can be found in the Network and dial up connections. Configure the local area connection by giving IP address, Subnetmask, Default gateway, DNS 5. In the Internet explorer on the System, configure the Lan settings as per the local area settings Expected Result: Browsing the Internet should be possible

24.5.2.6 Platform-Specific API

This section describes the platform-specific API functions.

24.5.2.6.1 InitializeMux

This function is called to initialize the IOMUX connection within i.MX, from the USB controller to the appropriate device pins for the transceiver. This function is implemented for the Pure Client situation.

Parameters

int Speed	[in] Unused
Return Value	Return TRUE if device requesting dwCfgPower can be safely attached

24.5.2.6.2 HardwarePullupDP

This function is called by the USB client driver when D+ must be pulled-up, in preparation for connection to a USB host. The standard code configures for ISP1504/ISP1301 transceiver. It is possible to modify this routine to conditionally soft-disable USB connection.

Parameters

CSP_USB_REGS *pRegs	[in] pointer to the registers for the USB controller
Return Value	Return TRUE if D+ signal was pulled-up

24.5.3 USB Transceiver Driver (ID Pin Detect Driver—XCVR)

This driver is responsible for detecting the type of USB connector plugged into the Mini USB OTG socket of the system. Upon detection the driver activates the USB host controller driver or USB function controller driver.

24.5.3.1 User Interface

There is no user interface to the transceiver driver. This driver merely manages the USB host or client drivers, which provide the appropriate programming API. The driver can be configured through its platform-specific routines to provide different behavior for power management (wake-up, D+ soft connect.).

24.5.3.2 Transceiver Driver Configuration

The transceiver driver is configured into the BSP automatically upon dragging and dropping the USB HS OTG catalog item. If transceiver functionality is not required, it can be disabled as described below.

24.5.3.3 Registry Settings

The USB OTG transceiver settings are values located under the registry key:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\XVC]
```

The values under this registry key are automatically included in the image via platform.reg. They do not normally require customization. Table 24-11 shows the USB OTG transceiver registry settings.

Table 24-11. USB OTG Transceiver Registry Settings

Value	Type	Content	Description
Dll	sz	imx_xvc.dll	Driver dynamic link library
OTGSupport	dword	01	This value must be set to 1 to enable the transceiver-driven mode switching on the OTG. If no transceiver support is required (host or client only) then this value can be set to 0, though the XVC key are not normally configured in the image when OTG Pure Host or OTG Pure Client is configured
OTGGroup	sz	01	This unique string (example 00 to 99) is used to combine/correlate instances of the host, function, and transceiver driver within one USB OTG instance

24.5.3.4 Unit Test

There is no CETK test case for USB transceiver driver. The transceiver driver is tested using the Mini USB OTG plug A and Mini USB OTG plug B. The test is done by manually plugging in the Mini USB OTG plug into the Mini USB OTG socket of the system. The test verifies that the USB host or function controller driver is activated on cable plug-in.

24.5.3.4.1 Unit Test Hardware

Table 24-12 lists the required hardware to run the unit tests.

Table 24-12. Hardware Requirements

Requirement	Description
System to act as a device	System is configured as USB Mass storage class
USB LS Mouse	To test if control reaches the Host controller driver
USB cable having A-type plug at one end and Mini USB OTG plug B on the other end. To plug in USB LS mouse, a USB extension cable having mini-A at one end and USB A-type socket at the other end	For connecting between the host and the device

24.5.3.4.2 Running the Transceiver Test

Table 24-13 lists transceiver tests.

Table 24-13. Transceiver Tests

Test Cases	Entry Criteria/Procedure/Expected Results
Idle case when no cable plugged in	Entry Criteria: Make sure there is no mini USB OTG plug connected and the board is turned on and wait until the board boots-up completely Procedure: When the board is powered and completely booted-up, the board should be idle (and as mass storage client, not verifiable) Expected Result: Device boots up and is stable
Mass storage client visible from PC	Entry Criteria: Make sure there is no mini USB OTG plug connected and the board is turned on and wait until the board boots-up completely Procedure: When the board is powered and completely booted-up, verify that board responds as a mass storage client when plugged into PC. Expected Result: New storage must be visible on PC.
Mini USB OTG plug-A connected to the mini USB OTG socket of System and mouse plugged into OTG port via this cable	Entry Criteria: Unplug board from PC (in previous step) Procedure: 1. Connect the USB HID device (Mouse) which has a Mini USB OTG plug-A to it. The control goes to the USB Host controller driver 2. The corresponding device gets enumerated and starts functioning. For example, if a USB mouse is connected, on movement of the mouse, the pointer in the LCD screen is seen moving Expected Result: Device should start functioning

24.5.3.5 Platform-Specific API

The transceiver driver library code contains i.MX chip-specific implementation, and is located in:

```
..\PLATFORM\COMMON\SRC\SOC\<Target SOC>\USBXVR
```

The transceiver driver operation can be customized through the platform-specific code located in:

```
..\PLATFORM\<Target Platform>\SRC\Drivers\USBXVR
```

The standard implementation located in `hwinit.c` is provided for the System with ISP1504 transceiver attached to the High Speed OTG port. Customizations permit different power management and wake-up behavior, including when the device generates soft connect/disconnect (D+ pull-up) or what wake-up conditions are supported when nothing is attached to the OTG port.

The library USB transceiver code communicates with the platform-specific code by callback functions. Only one globally-defined specific routine (RegisterCallback) is required for using this interface. Standard code is supplied for full transceiver operation using the System Platform.

24.5.3.5.1 Structure BSP_USB_CALLBACK_FNS

Structure BSP_USB_CALLBACK_FNS is defined in `MX25_usb`.h. This is a structure containing all the USB callback functions as called by the USB CSP drivers. Currently only the transceiver driver (USBXVR) uses these callback functions. The callback functions are registered using RegisterCallback() (see Section 24.5.3.6.2, “RegisterCallback,”).

```
typedef struct {
// pfnUSBPowerDown - function pointer for platform to call during power down.
// pfnUSBPowerUp - function pointer for platform to call during power up.
// Parameter: 1) regs - USB registers
//            2) pUSBCoreClk - pointer to boolean to indicate the status of USB Core Clk
//              if it is on or off. Platform is responsible to update this value if they change
//               the status of USBCoreClk. [TRUE - USBCoreClk ON,  FALSE - USBCoreClk OFF]
//            3) pPanicMode - pointer to boolean to indicate the status of panic mode
//              if it is on or off. Platform is responsible to update this value if they change
//               the status of panic mode. [TRUE - PanicMode ON,  FALSE - PanicMode OFF]
void (*pfnUSBPowerDown)(CSP_USB_REGS *regs, BOOL *pUSBCoreClk);
void (*pfnUSBPowerUp)(CSP_USB_REGS *regs, BOOL *pUSBCoreClk);
// pfnUSBSetPhyPowerMode - function pointer for platform to call when they want to
suspend/resume the PHY
// Parameter: 1) regs - USB registers
//            2) bResume - TRUE - request to resume, FALSE - request suspend
void (*pfnUSBSetPhyPowerMode)(CSP_USB_REGS *regs, BOOL bResume);
} BSP_USB_CALLBACK_FNS;
```

24.5.3.5.2 pfnUSBPowerDown

This callback function is called during the Windows Embedded CE 6.0 power down sequence. The actual platform specific power down routine should be registered as this callback function. This function is only called if the system is in USB transceiver mode only (for example, when nothing is attached to the OTG port.).

There is no standard implementation for this callback, since by default the transceiver driver automatically suspends the port when nothing is attached. This enables wake-up on device or host attachment, and enables the D+ pull-up during the suspended condition.

Parameters

CSP_USB_REGS *regs	[in] Mapped pointer to the USB registers in i.MX, from physical address space to a non-paged, process-dependent address space. This is mapped during the transceiver initialization routine (XVC_Init).
BOOL *pUSBCoreClock	[in/out] Pointer to a Boolean variable in transceiver to indicate whether the USB Core Clock has been stopped. The platform-specific callback function must update this flag to reflect the current USB Core Clock status, if the status of the USB Core Clock is changed within the platform code (for example using DDKClockSetGatingMode()). This ensures consistency of the clock status within the CSP transceiver driver.
Return Value	TRUE—USB Core Clock is running FALSE—USB Core Clock is stopped

24.5.3.6 pfnUSBPowerUp

Similar to pfnUSBPowerDown, this is called during the Windows Embedded CE 6.0 power up sequence. The actual platform specific power up (resume) routine should be registered to this pointer. This is only called when USB is in transceiver mode (when nothing is attached to the OTG port).

There is no standard implementation for this callback, since by default the transceiver driver automatically suspends the port when nothing is attached and the port need not perform any wake-up activity until a device or host attachment is detected.

Parameters	For parameter details see pfnUSBPowerDown, Section 24.5.3.5.2, “pfnUSBPowerDown,”

24.5.3.6.1 pfnUSBSetPhyPowerMode

This function is called when the system is in USB transceiver mode with no USB activity. With standard implementation on the system, if the system is in transceiver mode and there is no activity in USB port for one second, the transceiver driver suspends the ULPI PHY (in this case, it is ISP1504, disable the USB Clock gating, and set the system to non-panic mode allowing core voltage to drop).

When there is USB activity (for example, device attach), the transceiver driver sets the system to panic mode (requiring core voltage to stay high using DDKClockEnablePanicMode(), supported for i.MX), enables USB Clock gating and puts the ULPI PHY transceiver to resume.

This callback function is responsible for handling the suspend and resume of ULPI PHY transceiver. The developer must register this pointer with the actual platform specific function for suspend and resume of ULPI PHY transceiver. Custom wake-up conditions can be enabled here.

Parameters

CSP_USB_REGS *regs	[in] Mapped pointer to the USB registers in i.MX, from physical address space to a non-paged, process-dependent address space. This is mapped during the transceiver initialization routine (XVC_Init).
BOOL resume	[in] This boolean variable indicates whether the callback function must resume or suspend the ULPI PHY transceiver.
Return Value	TRUE—callback function must resume transceiver activity FALSE—callback function must suspend transceiver activity

24.5.3.6.2 RegisterCallback

This is used to register all the callback functions defined in BSP_USB_CALLBACK_FNS. This function is called by the USB driver during the initialization process of the transceiver driver (XVC_Init). The developer must implement a function by this name in their platform directory. A standard implementation is provided for the ISP1504 transceiver of the System. When no callback function is required, those elements of the BSP_USB_CALLBACK_FNS structure should be initialized to NULL.

Parameters

BSP_USB_CALLBACK_FNS *pFn

[in/out] Pointer to BSP_USB_CALLBACK_FNS structure for the developer to register the callback function inside the BSP_USB_CALLBACK_FNS. The callback functions inside this structure is used by the CSP transceiver code.

24.5.4 Power Management

There are four aspects of power management for the USB device drivers:

- Special i.MX Vcore requirements
- Clock gating to the USB peripheral block within the i.MX
- Setting the transceiver to a lower power mode or suspend
- Transceiver auto-power-down on inactivity

The USB device driver(s) support an On and Off/Standby (low power) state, with wake-up capability. The On state is entered whenever a host or device is attached to the relevant USB port. The driver enters the standby state automatically after timeout with no device or host attached to the USB port. As well, the standby state is entered when the system suspends. (In the latter case, system wake-up capability is enabled for the port).

24.5.4.1 Special Vcore Requirements

When ULPI-bus transceivers are used with the USB controller (for example, ISP1504 transceivers for High Speed OTG port and High Speed Host 2 port on the i.MX System), normal DVFS scaling of the i.MX Vcore must be suspended whenever there is potential of ULPI bus communication. This is the case whenever a device is connected (in host mode) or the device is connected to a host (in client mode). The USB OTG transceiver driver, and USB host and client drivers constrain the DVFS behavior by calling DDKClockEnablePanicMode() whenever a device or host connection is detected, and calling DDKClockDisablePanicMode() when a timeout period expires with no device or host connected to the port. No configuration is required, just note the effect on the DVFS (DVFC driver) behavior.

24.5.4.2 Clock Gating

The USB driver(s) for the various USB ports automatically manages clock gating to the i.MX USB controller cores. The drivers for the ports coordinate their use of the USB core clock, and when nothing is connected on any of the ports (all drivers are in their lowest power state) the clock is gated on or off using:

```
DDKClockSetGatingMode(DDK_CLOCK_GATE_INDEX_USBOTG, DDK_CLOCK_GATE_MODE_ENABLED_ALL)
DDKClockSetGatingMode(DDK_CLOCK_GATE_INDEX_USBOTG, DDK_CLOCK_GATE_MODE_DISABLED)
```

24.5.4.3 Transceiver Auto Power Down

The USB transceivers automatically enter a lower-power/suspended mode when no USB traffic is detected for several milliseconds. This internally sets a suspended state for the USB port. Software timeout is used to establish whether the driver power mode can be switched to its lowest power state.

24.5.4.4 Transceiver Power Mode

Software timeout is used to establish whether the transceivers and their related bus needs to be set to a suspended condition. In the lowest-power state, the transceiver is configured to generate wake-up signalling on attachment of devices or host (to the OTG port). The transceiver driver provides callback routines to manage this transition.

24.5.4.5 PowerUp

Each of the OTG client, host and transceiver drivers have PowerUp routine associated. (For the host driver, this is referenced by the MDD to a function PowerMgmtCallback()).

For the host, the routine does the following:

- Ungate the USB peripheral block clock
- Force the port to resume and clear PHCD bit in the portsc register
- Reset and configure USB host controller
- Disable the wake-up conditions
- Set the PHY to normal work mode using SetPHYPowerMgmt(FALSE) platform routine
- Enable the interrupts and start the USB controller

For the client, the routine does the following:

- Ungate the USB peripheral block clock
- Force the port to resume
- Disable the wake-up conditions
- Enable the interrupts and start the USB controller

For the transceiver driver, the PowerUp routine calls the relevant platform-specific callback routine, pfnUSBPowerUp().

Under normal circumstances there is nothing to be done in this routine, since the OTG port is normally in a suspended state within the transceiver mode. (It is only in transceiver mode when nothing is connected to the port, and thus has already been automatically suspended).

24.5.4.6 PowerDown

As for the PowerUp routine, OTG client, host and transceiver drivers have PowerDown routine associated. (For the host driver, this is referenced via the MDD to a function PowerMgmtCallback()).

For the host, the routine does the following:

- Verify the wake-up conditions using the BSPUsbCheckWakeUp() platform routine
- Stop the host controller
- Suspend the relevant port
- Set the PHY to low power mode using SetPHYPowerMgmt(TRUE) platform routine
- Gate the USB peripheral block clock

For the client, the routine does the following:

- Stop the USB controller
- Clear any outstanding interrupts
- Enable appropriate wake-up condition
- Suspend the relevant port (suspends the PHY)
- Gate the USB peripheral block clock

For the transceiver driver, the PowerDown routine calls the relevant platform-specific callback routine, pfnUSBPowerDown().

Under normal circumstances there is nothing to be done in this routine, since the transceiver remains in its suspended state while nothing is connected to the port. Should any attachment have been made, the transceiver wakes through its wake-up mechanism and launch the appropriate (client or host) driver.

24.5.4.7 Suspend/Resume Operations

- Mass Storage Host/Client—Device is mounted automatically, but any unfinished browse/copy is terminated
- ActiveSync Client—Once browsing into the content of device. A system suspend/resume causes device to not be mounted until unplug and plug cable again
- HID Host—Client is recognized again automatically

24.5.5 Function Drivers

The function drivers can be configured into the image using the Windows CE 6.0 Platform Builder catalog, and are located at:

Device Drivers > **USB Function** > **USB Function Clients**

The default function driver is launched when the USB device port is attached to a host. This default function driver is selected by the registry key (the last instance of this value in reginit.ini applies):

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers]
   "DefaultClientDriver"=-; erase previous default
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers]
   "DefaultClientDriver"="Mass_Storage_Class"
```

or

```
   "DefaultClientDriver"="RNDIS"
```

or

```
   "DefaultClientDriver"="Serial_Class"
```

Unless the BSP is configured with persistent registry storage, it only makes sense to configure a single function driver into the image, and this one becomes default.

NOTE

When no USB client functionality is included in the image (No OTG port, or OTG Pure Host only), ensure that no function drivers have been configured. If function drivers are configured, then USB client driver libraries are also included in the image through logic in:

```
PUBLIC\CEBASE\OAK\Misc\winceos.bat
```

24.5.5.1 Mass Storage Function

Table 24-14. Mass Storage Function

Driver Attribute	Definition
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<*Common SOC*>\ms\USBFN\CLASS
CSP Static Library	N/A
Platform Driver Path	N/A
Import Library	USBMSFN_LIB_<*Common SOC*>.lib UFNCLIENTLIB.LIB
Driver DLL	usbmsfn.dll
Catalog Item	Device Drivers > USB Function > USB Function Clients > Mass Storage
SYSGEN Dependency	SYSGEN_USBFN_STORAGE

The Mass Storage function exposes a local data store as a USB peripheral storage device. The device used can be specified in registry. In platfrom.reg, the following template is provided:

```
PUBLIC\Common\OAK\Files\common.reg
"DeviceName"=-;
; "DeviceName"="ATA HARD DISK"
; "DeviceName"="SDMEMORY CARD"
; "DeviceName"="MMC CARD"
; "DeviceName"="USB HARD DISK"
; "DeviceName"="NAND FLASH"
```

Any item from this list can be specified to act as the mass storage medium. Uncomment the corresponding line and rebuild the BSP to make that item active. If none of the items are specified explicitly, a pre-coded priority is used to determine what active drive acts as mass storage medium. The priority is described as the following:

ATA HARD DISK > SDMEMORY CARD (MMC CARD) > USB HARD DISK > NAND FLASH

platform.reg can also over-ride other USBMSFN related default settings. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Mass_Storage_Class]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
   "idVendor"=dword:045E
   "Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
   "idProduct"=dword:FFFF
   "Product"="Generic Mass Storage (PROTOTYPE--Remember to change idVendor)"
   "bcdDevice"=dword:0
```

24.5.5.2 Serial Function

The primary use for the serial function is ActiveSync.

Table 24-15. Serial Function

Driver Attribute	Definition
CSP Driver Path	N/A
PUBLIC driver path	PUBLIC\Common\OAK\Drivers\USBFN\CLASS\SERIAL
CSP Static Library	N/A
Platform Driver Path	N/A
Export Library	serialusbfn.lib
Import Library	com_mdd2.lib serpddcm.lib ufnclientlib.lib
Driver DLL	SerialUsbFn.dll
Catalog Item	Device Drivers > USB Function > USB Function Clients > Serial Client
SYSGEN Dependency	SYSGEN_USBFN_SERIAL

NOTE

> ActiveSync has been tested using connection to a PC with ActiveSync version 4.1 installed. See Microsoft.com to download the latest ActiveSync software for the PC. In some cases, DEBUGCHK may be triggered during attachment to ActiveSync in DEBUG builds.

When SYSGEN_USBFN_SERIAL is defined, the default registry entry is automatically included from:

```
PUBLIC\Common\OAK\FILES\common.reg
```

For commercial products, this registry entry must be copied into platform.reg and modified to over-ride the defaults. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\Serial_Class]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
   "idVendor"=dword:045E
   "Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
   "idProduct"=dword:00ce
   "Product"="Generic Serial (PROTOTYPE--Remember to change idVendor)"
   "bcdDevice"=dword:0
```

24.5.5.3 RNDIS Function

The RNDIS function allows communication over USB to be supplied to ethernet NDIS interface of protocol stack.

Table 24-16. RNDIS Function

Driver Attribute	Definition
CSP Driver Path	N/A
CSP Static Library	N/A
Platform Driver Path	N/A
PUBLIC Driver Path	PUBLIC\COMMON\OAK\Drivers\USBFN\Class\RNDIS
Import Library	ndis.lib
Driver DLL	RNDISFN.DLL
Catalog Item	Device Drivers > USB Function > USB Function Clients > RNDIS Client
SYSGEN Dependency	SYSGEN_USBFN_ETHERNET

RNDIS function has been tested using Freescale RNDIS class driver as located at:

```
Support\RNDIS\ce6_rndis.inf
%WINDIR%\System32\drivers\usb8023x.sys
```

When SYSGEN_USBFN_ETHERNET is defined, the default registry entry is automatically included from:

```
PUBLIC\Common\OAK\FILES\common.reg
```

For commercial products, this registry entry must be copied into platform.reg and modified to over-ride the defaults. This allows customizing the following values which must be properly configured for a commercial device:

```
[HKEY_LOCAL_MACHINE\Drivers\USB\FunctionDrivers\RNDIS]
; idVendor must be changed. 045E belongs to Microsoft and is only to be used for
; prototype devices in your labs. Visit http://www.usb.org to obtain a vendor id.
"idVendor"=dword:045E
"Manufacturer"="Generic Manufacturer (PROTOTYPE--Remember to change idVendor)"
"idProduct"=dword:0301
"Product"="Generic RNDIS (PROTOTYPE--Remember to change idVendor)"
"bcdDevice"=dword:0
```

24.5.6 Class Drivers

All host ports (OTG Host, High Speed Host (H2), and Full Speed Host (H1)) support the same class drivers, and this configuration is common to all host ports. Class drivers must also be configured for the USB host ports. Class driver configuration is common to all host ports—there is no port-specific configuration to be completed on any class driver.

Table 24-17 shows the standard Microsoft-supplied drivers that are available by drag and drop from the catalog.

Table 24-17. Class Drivers

Class Driver	Configuration Flag	Catalog Item
HID	SYSGEN_USB_HID	Core OS > Windows CE devices > Core OS Services > USB Host Support > USB Human Input Device (HID) Class Driver
Printer	SYSGEN_USB_PRINTER	.. > USB Printer Class Driver[1]
Keyboard	SYSGEN_USB_HID_KEYBOARD	.. > USB HID Keyboard Only[1]
	SYSGEN_USB_HID_MOUSE	.. > USB HID Mouse Only[1]
RNDIS	SYSGEN_ETH_USB_HOST	Core OS > Windows CE devices > Core OS Services > USB Host Support > USB Remote NDIS Class Driver
Storage	SYSGEN_USB_STORAGE	Core OS > Windows CE devices > Core OS Services > USB Host Support > USB Storage Class Driver

[1] See additional configuration in Section 24.6.2, “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`

24.5.6.1 HID Mouse

For mouse support, the cursor is required to test and use the mouse as shown in Table 24-18.

Table 24-18. HID Mouse Class Driver

Catalog Item	Configuration Flag	Catalog Item
HID	SYSGEN_CURSOR	Core OS > Shell and User Interface > User Interface > Mouse

24.5.6.2 HID Keyboard

The system keyboard key mapping conflicts with that used for the HID keyboard. When USB keyboard support is included, remove the System keyboard (Table 24-19) and include the appropriate stub keyboard and keyboard .dll (Table 24-20)

Table 24-19. HID Keyboard Driver to Remove

Remove Item	Remove Catalog Item
Keyboard	Third Party > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Input Devices > Keyboard/Mouse

Include stub keyboard:

Table 24-20. ID Keyboard Driver to Include

Catalog Item	Configuration Flag	Catalog Item
NOP Stub Keyboard	BSP_KEYBD_NOP	Device Drivers > Input Devices > Keyboard/Mouse > NOP (Stub) Keyboard/Mouse English

Also, include the appropriate keyboard .dll. For example, define SYSGEN_KBD_US and add the following lines in the platform.bib (immediately before the FILES section):

```
IF BSP_KEYBD_NOP
    kbdmouse.dll    $(_FLATRELEASEDIR)\KbdnopUs.dll                    NK SH
ENDIF; BSP_KEYBD_NOP
```

24.6 Basic Elements for Driver Development

This section provides details of the basic elements for driver development in the Platform System.

24.6.1 BSP Environment Variables

Table 24-21 shows the system environment variables.

Table 24-21. System Environment Variables Summary

Name	Definition
BSP_USB	Set to configure USB in BSP
BSP_USB_HSOTG_XVC	Set to enable Full OTG functionality (transceiver host-client switching) on the High Speed OTG port
BSP_USB_HSOTG_CLIENT	Set to include USB client functionality on High Speed OTG port
BSP_USB_HSOTG_HOST	Set to include USB host functionality on High Speed OTG port.

Pin conflicts between default driver implementations for the pin muxing (platform-specific implementation) mean certain configurations are mutually exclusive, as listed in Table 24-22.

Table 24-22. Mutual Exclusive Driver Summary

Functionality[1]	BSP_ATA	BSP_CSPIBUS	BSP_USB	BSP_USB_HSOTG_XVC	BSP_USB_HSOTG_CLIENT	BSP_USB_HSOTG_HOST
ATA disk drive	yes	no	—	—	—	—
High Speed OTG Port full function (Host + Client)	—	—	yes	yes	yes	yes
High Speed OTG Port Pure Host only	—	—	yes	—	—	yes

Table 24-22. Mutual Exclusive Driver Summary (continued)

High Speed OTG Port Pure Client only	—	—	yes	—	yes	—
Full Speed Host (H1)	no	no	—	—	—	—
High Speed Host (H2)	no	no	—	—	—	—

[1] yes = Required, no = Not permitted, — = Do not care

24.6.2 Dependencies of Drivers

Table 24-23 summarizes the Microsoft-defined environment variables used in the BSP.

Table 24-23. USB Driver

Name	Definition
SYSGEN_USBFN_SERIAL	Set to support serial class for USB Function controller
SYSGEN_USBFN_STORAGE	Set to support mass storage class for USB Function controller
SYSGEN_USBFN_ETHERNET	Set to support RNDIS class for USB Function controller
SYSGEN_CURSOR	Set to support mouse cursor
SYSGEN_FATFS	Set to support FAT16 file system
SYSGEN_PCL	Set to support PCL printing
SYSGEN_PRINTING	Set to support printer
SYSGEN_STOREMGR	Set to support storage manager
SYSGEN_UDFS	Set to support Universal Disc File System
SYSGEN_USB	Set to support USB driver
SYSGEN_USB_HID	Set to support Human Interface driver (HID) class
SYSGEN_USB_HID_CLIENTS	Set to support HID clients
SYSGEN_USB_HID_KEYBOARD	Set to support HID keyboards (keyboard stub and associated .dll are required)
SYSGEN_USB_HID_MOUSE	Set to support HID mouse
SYSGEN_USB_PRINTER	Set to support Printer (printer driver support, such as PCL (SYSGEN_PCL), may be required)
SYSGEN_USB_STORAGE	Set to support storage medium

Chapter 25
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.

25.1 USB Boot and KITL Summary

Table 25-1 identifies the source code location, library dependencies, and other BSP information.

Table 25-1. USB Boot and KITL Summary

Driver Attribute	Definition
Target Platform	iMX25-3DS-PDK1_7
Target SOC	MX25_FSL_V2_PDK1_7
SOC Common Path	WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\MS\RNE_MDD WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\MS\USBKITL
SOC Specific Path	WINCE600\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\USBD\KITL
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\COMMON\USBFN ..\PLATFORM\<*Target Platform*>\SRC\KITL
Driver DLL	fsl_usbfn_rndiskitl_PDK1_7.lib
SDK Library	N/A
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variable	N/A

25.2 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

25.3 Hardware Operation

For detailed operation and programming information of the USB OTG, see the chapter on the High-Speed USBOTG_UTMI in the corresponding platform Users Guide.

25.3.1 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.

25.4 Software Operation

25.4.1 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 25-1 illustrates the USB Boot and KITL software architecture.

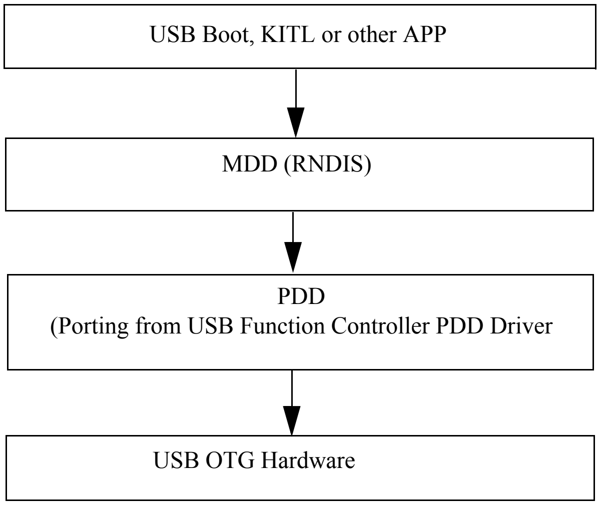
USB Boot, KITL or other APP
MDD (RNDIS)
PDD
(Porting from USB Function Controller PDD Driver
USB OTG Hardware

Figure 25-1. USB Boot and KITL Software Architecture Block Diagram

Microsoft has implemented a RNDIS client MDD driver in Windows CE 6.0. The code is in following location:

`%_WINCEROOT%\Public\Common\Oak\Drivers\Ethdbg\Rne_mdd`

It generates the static library `Rne_mdd.lib`.

The USB function controller PDD driver is ported to eboot and KITL to support USB Boot and KITL. For details of USB function controller PDD driver refer to the Platform Builder for Windows CE 6.0 Help under the topic:

Developing a Device Driver > **Windows Embedded CE Drivers** > **USB Function Drivers** > **USB Function Controller Drivers** > **USB Function Controller Driver Reference** > **USB Function Controller PDD Functions.**

Windows CE 6.0 provides an example of USB Boot. It is located at:

`%_WINCEROOT%\Platform\MainstoneIII\Src\Common\Usbfn`

25.4.2 Source Code Layout

Some files are modified or added to support USB Boot and KITL. They are as follows:

- RNDIS PDD driver

  `%_WINCEROOT%\Platform\COMMON\SRC\SOC\COMMON_FSL_V2_PDK1_7\MS\USBKITL\RNDIS`
- USB function controller shared with OS driver

  `%_WINCEROOT%\Platform\COMMON\SRC\SOC\<Target SOC>\USBD\COMMON`
- Add RNDIS device to EBOOT ethernet initialization routines

  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Common\ether.c`
- Setup KITL device LogicalLoc and PhysicalLoc to USBOTG physical address if USB KITL option in EBOOT menu is selected by user

  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Common\main.c`
- Implement private OS functions, such as `NKCreateStaticMapping()`. `NKCreateStaticMapping` is defined in OS. It is not defined for EBOOT while USB Boot requires this function. So it is manually defined. This function just calls OALPAtoUA()

  `%_WINCEROOT%\Platform\COMMON\SRC\SOC\<Target SOC>\USBD\KITL`
- Add USB Boot and KITL options into EBOOT menu

  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Eboot\menu.c`
- Add `fsl_rne_mdd_$(_COMMONSOCDIR).lib, fsl_rne_pdd_$(_COMMONSOCDIR).lib, usb_usbfn_$(_SOCDIR).lib, usb_usbfn_eboot_$(_SOCDIR).lib`

  `%_WINCEROOT%\Platform\<Target Platform>\Src\Bootloader\Eboot\sources`
- Add USB RNDIS KITL device in KITL initialization routines

  `%_WINCEROOT%\Platform\<Target Platform>\Src\Kitl\kitl.c`

  `%_WINCEROOT%\Platform\<Target Platform>\Src\Kitl\sources`

25.4.3 Power Management

Power management is not implemented in USB Boot and KITL.

25.4.4 Registry Settings

There are no related register settings for the USB Boot and KITL.

25.4.5 DMA Support

Physical contiguous memory is required to support USB DMA. This memory region is hard coded in:

`%_WINCEROOT%\Platform\Common\SRC\SOC\<Common Soc>\ms\Usbkitl\Rndis\rndis_pdd.c`

It uses the BSP reserved IPL RAM image region (Starting from IMAGE_USB_KITL_RAM_PA_START). This region is not used by other modules in the BSP, so it can be used by USB boot and KITL.

25.5 Unit Test

25.5.1 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.

25.5.2 Testing USB Boot and KITL on i.MX25 3-Stack

There are totally three Ethernet transport devices available on i.MX25 3-Stack for image download with Boot and KITL connection, LAN9217, FEC and USB RNDIS. LAN9217 is the default. Follow the steps below to use USB RNDIS for Boot and KITL:

1. Connect target board to PC with USB cable and power on the board
2. Select USB RNDIS as the Ethernet transport device in EBOOT

```
0) IP address: 192.168.0.2
1) Subnet Mask: 255.255.255.0
3) DHCP: Disabled
6) Set MAC Address: 0-12-34-56-78-12
9) Ethernet Device: USB RNDIS
```

3. Press ‘d’ to download image over USB. If this is the first time running USB Boot or KITL with the PC, the PC shows up a Found New Hardware Wizard dialog box and prompt to install the driver for Windows CE RNDIS virtual adapter on the Windows PC. Refer to `WINCE600\PUBLIC\COMMON\OAK\DRIVERS\ETHDBG\RNDISMINI\HOST\howto.txt` for how to install the driver
4. After the driver is installed successfully, the Windows CE RNDIS virtual adapter should be displayed in Network Connections on the PC. Configure this network connection properly. Use a static IP address (such as 192.168.0.3) in the same subnet as the target board.
5. Check the **Platform Builder Target** > **Connectivity** options to make sure the target device is selected.

If KITL is being enabled in the run-time image, the connection is established on USB after the image downloading is finished.