i.MX51 EVK
Windows Embedded CE 6.0

Reference Manual

Part Number: 924-76370
Rev. WCE600_MX51_ER_1104
05/2011

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

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

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

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

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

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

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
P.O. Box 5405
Denver, Colorado 80217
+1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

freescale™
semiconductor

Contents

Chapter 3
Battery Driver

Chapter 4
Bluetooth USB Adapter Driver

Chapter 5
Boot from Secure Digital/MultiMedia Card (SD/MMC)

Chapter 6

Camera Driver for IPUv3

Chapter 7
Chip Support Package Driver Development Kit (CSPDDK)

Chapter 8
Display Driver for IPUv3

Chapter 9
Dynamic Voltage and Frequency Control (DVFC) Driver

Chapter 10
Enhanced Configurable Serial Peripheral Interface (eCSPI) Driver

Chapter 11
Enhanced Secure Digital Host Controller (eSDHC) Driver

Chapter 12
Fast Ethernet Controller (FEC) Driver

Chapter 13
General Purpose Timer (GPT) Driver

Chapter 14
Graphics Processing Unit (GPU)

Chapter 15
Inter-Integrated Circuit (I^2C) Driver

Chapter 16
Keypad Driver

Chapter 17
Notification LED Driver

Chapter 18
One-Wire (OWIRE) Driver

Chapter 19
Power Management IC (PMIC)

Chapter 20
Serial Driver

Chapter 21
Sony/Philips Digital Interface (SPDIF) Driver

Chapter 22
Touch Panel Driver

Chapter 23
TV Encoder (TVE)

Chapter 24
Universal Serial Bus (USB) OTG Driver

Chapter 25
USB Boot and KITL

Chapter 26
UUT Driver

Chapter 27
Video Processing Unit (VPU)

About This Book

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

Audience

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

Suggested Reading

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

- i.MX51 Applications Processor Reference Manual
- i.MX51 EVK Windows Embedded CE 6.0 Release Notes
- i.MX51 EVK Windows Embedded CE 6.0 User's Guide
- Microsoft Platform Builder for Windows Embedded CE 6.0 Help

Conventions

This document uses the following notational conventions:

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

Definitions, Acronyms, and Abbreviations

Table i contains acronyms and abbreviations used in this document.

Table i. Acronyms and Abbreviated Terms

Term	Meaning
API	Application programming interface
BSP	Board support package
CSP	Chip support package

Table i. Acronyms and Abbreviated Terms (continued)

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

Table i. Acronyms and Abbreviated Terms (continued)

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

Chapter 1
Introduction

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

- i.MX51 EVK Development System

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

NOTE

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

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

1.1 Getting Started

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

1.2 Windows Embedded CE 6.0 Architecture

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

Welcome to Windows Embedded CE 6.0 > Windows Embedded CE Architecture

Chapter 2
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 or WaveIn functionality, see the Platform Builder Help at the following location:

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

2.1 Audio Driver Summary

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

Table 2-1. Audio Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\WAVEDEV2
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<Target SOC>\WAVEDEV2
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\WAVEDEV2\SGTL5000
Driver DLL	wavedev2_sgtl5000.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX51-EVK:ARMV4I > Device Drivers > Audio > SGTL5000 Audio Driver
SYSGEN Dependency	SYSGEN_AUDIO
BSP Environment Variables	BSP_NOAUDIO= BSP_AUDIO_SGTL5000=1

NOTE

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

2.2 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
8. Headphone detection

2.3 Hardware Operation

This section describes about the audio hardware operation.

2.3.1 Audio Hardware Design

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

2.3.1.1 i.MX51 EVK Audio Hardware Design

The Synchronous Serial Interface is a full-duplex serial port and the i.MX51 SoC uses instance 2 (SSI2) for both audio playback and recording. The external stereo codec SGTL5000 is connected to AUDMUX port 3 (external) while SSI2 is internally connected to AUDMUX port 2 (internal). Both ports are configured to operate in synchronous 4-wire mode.

The i.MX51 uses the I^2C bus interface to access the SGTL5000 control registers, so that the SGTL5000 can be configured by the i.MX51 as per hardware design and software configuration. The SGTL5000 stereo codec on the i.MX51 EVK supports headphone, line out and speaker outputs as well as microphone and line in inputs.

For operation and programming, see the chapters in the *i.MX51 Applications Processor Reference Manual*, for the SSI, SDMA, AUDMUX, and IOMUX components, as well as the *SGTL5000 Datasheet* for the external audio codec.

Figure 2-1 shows the signal connections between the i.MX51 and the SGTL5000.

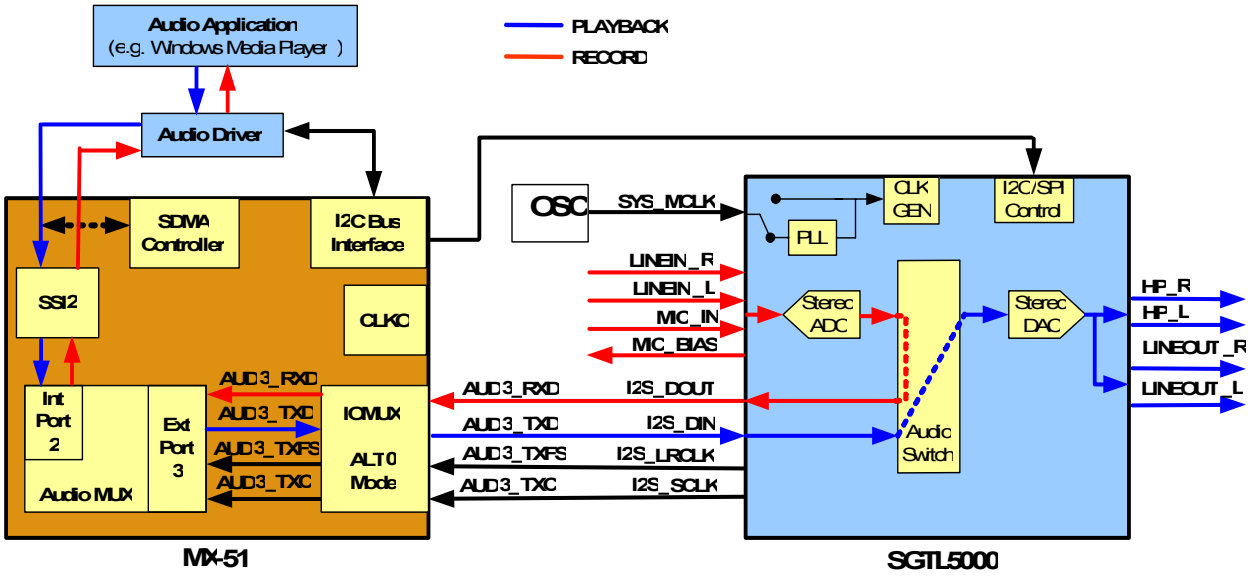


Figure 2-1. SGTL Hardware Connections

2.3.2 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 3 is used to connect the Stereo Codec on the i.MX51 EVK 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 FIFOs level reaches the empty watermark level. The SDMA controller then refills FIFOs 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 FIFOs 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.

2.3.3 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 FIFO.
- 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 FIFO level reaches the full watermark level. The SDMA controller then transfers the data from the receiver FIFO 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 FIFO 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.

2.3.4 Required SoC Peripherals

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

Table 2-2. 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

2.3.5 Conflicts with SoC Peripherals

No conflicts.

2.3.6 Conflicts with Board Peripherals

The following section explains about the conflicts of the audio driver with board peripherals:

2.3.6.1 i.MX51 EVK Peripherals Conflicts

No conflicts.

2.3.7 Known Issues

The following section explains about the known issues in the audio driver:

2.3.7.1 i.MX51 Known Issues

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

2.4 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 at the following location:

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

2.4.1 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.

2.4.2 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, discard any extra data. The application is signaled using a callback function when the application-supplied buffer is full.

2.4.3 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.

2.4.3.1 i.MX51 Audio Driver Configuration Options

Table 2-3 gives the compile-time configuration options for the i.MX51 stereo audio driver.

Table 2-3. i.MX51 Audio Driver Configuration Options (oemsettings.h)

Configuration Setting	Description
INCHANNELS	Defines the number of input/recording channels that are available. Can be set to either 1 or 2. Default is 1.
OUTCHANNELS	Defines the number of output/playback channels that are available. Can be set to either 1 or 2. Default is 2.
HWSAMPLE	A typedef that defines the size of each audio data word. This must match the BITSPERSAMPLE and AUDIO_SAMPLE_MAX/AUDIO_SAMPLE_MIN values. Default is INT16.
USE_MIX_SATURATE	Enable a check in the software mixer code to guard against saturation. Default is 1.
AUDIO_SAMPLE_MAX AUDIO_SAMPLE_MIN	The valid range of each audio data word. Values that are outside of this range are 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).

Table 2-3. i.MX51 Audio Driver Configuration Options (oemsettings.h) (continued)

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, supports 16-bit sample; If set to 24, supports 24-bit sample. (in sgtl5000codec.h)

2.4.4 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 the internal memory (which is provided by on-chip internal RAM) or external memory (which is provided by off-chip external DRAM).

Table 2-4 describes the issues and considerations for the type of memory to use for the DMA data buffers.

Table 2-4. 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

2.4.4.1 i.MX51 Audio DMA Buffer Use

The i.MX51 audio driver supports both playback and recording. Playback function always uses internal memory as DMA buffer, while recording function allocates DMA buffer from external memory.

Table 2-5 describes how to configure the build so that the audio driver allocates its DMA data buffers from either the internal or external memory. The DMA buffer descriptors can also be allocated either from internal or external memory. However, the choice is made automatically through the use of the CSPDDK APIs, specifically DDKSdmaAllocChain(). See Chapter 6, "Chip Support Package Driver Development Kit (CSPDDK)," for additional information about the DDKSdmaAllocChain() API.

Table 2-5. Configuration Options for Internal or 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

2.4.5 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 are 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.

2.4.5.1 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.

2.4.5.2 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. So, first thing 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.

2.4.5.3 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.

2.4.6 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.

2.4.6.1 i.MX51 Audio Driver Registry Settings

The following registry keys are required for the Device Manager to properly load the i.MX51 audio device driver during the device normal boot process. These registry settings should typically not be modified. If they are missing or incorrectly defined, then the audio driver may not be loaded and all audio functions are disabled.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio]
         "Prefix"="WAV"
         "Dll"="wavedev2_sgtl5000.dll"
         "Index"=dword:1
         "Order"=dword:7
         "Priority256"=dword:95
         "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:5
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\WAPIMAN_ACM]
         "Order"=dword:5
```

2.5 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.

2.5.1 Unit Test Hardware

Table 2-6 identifies the hardware needed to run the unit tests.

Table 2-6. 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	—

2.5.2 Unit Test Software

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

Table 2-7. 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

2.5.3 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
```

2.5.4 Running the Audio Driver CETK Tests

The command line for running the audio driver test is:

```
tux –o –d wavetest -x 100
tux -o -d wavetest.dll -x 1000-2008 -c "-c audio_playback_latency_test_results.csv"
tux -o -d wavetest.dll -x 3000-3008 -c "-c audio_capture_latency_test_results.csv"
tux -o -d wavetest.dll -x 4000-4001
tux -o -n -d wavetest.dll -x 5000-5004 -c "-p"
tux -o -n -d wavetest.dll -x 6000-6001 -c "-t 3"
tux -o -d wavetest.dll -x 8000-8002
```

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

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

2.6 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.

2.6.1 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.

2.6.2 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.

2.6.3 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.

2.6.4 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 at the following location:

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

2.7 Audio Driver API Reference

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

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

2.8 Audio Driver Troubleshooting Guide

This section describes the techniques to identify and fix the most common problems involving the audio driver.

2.8.1 Checking Build-Time Configuration Options

Compile-time or link-time errors are probably occur due to incorrect or invalid configuration settings defined in `hwctxt.h` or `hwctxt.cpp`. See Section”i.MX51 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 2-1 for a list of the required and recommended audio driver-related catalog items.

2.8.2 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 more information on this topic, see the Platform Builder Help at the following location:

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

2.8.3 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.

Chapter 3
Battery Driver

The battery driver module provides information about the battery level to the OS. The battery driver is essentially a stub in this platform.

3.1 Battery Driver Summary

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

Table 3-1. Battery Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
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	battery.dll
Catalog Item	Third Party > BSP > Freescale i.MX51-EVK :ARMV4I> Device Drivers > Battery > Fake Battery Driver
SYSGEN Dependency	SYSGEN_BATTERY
BSP Environment Variables	BSP_NOBATTERY= BSP_FAKE_BATTERY=1

3.2 Supported Functionality

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

1. Conforms to the battery driver interface

3.3 Hardware Operation

The current i.MX51 EVK does not support battery monitoring or charging.

3.3.1 Conflicts with Other SoC Peripherals

No conflicts.

3.4 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.

3.4.1 Battery Driver Registry Settings

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

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

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Battery]
         "Prefix"="BAT"
         "Dll"="battery.dll"
         "Flags"=dword:8            ; DEVFLAGS_NAKEDENTRIES
         "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"
```

3.4.2 Power Management

There is no additional power management implementation for battery driver.

3.5 Unit Test

The battery driver does not include any unit tests.

3.6 Battery API Reference

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

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

Chapter 4
Bluetooth USB Adapter Driver

The Bluetooth USB adapter driver is used to drive BU 2073-J USB Bluetooth adapter to implement Bluetooth functionality compatible with Bluetooth v2.0. Bluetooth exchanges data with the i.MX51 through the USB host port. The BU 2073-J adapter adopts BlueCore4 Bluetooth solution of Cambridge Silicon Radio company with USB v1.1 interface.

4.1 Bluetooth USB Adapter Driver Summary

The Bluetooth USB adatper driver is provided in binary form instead of source codes. Table 4-1 provides a summary of the source code location, library dependencies, and other BSP information.

Table 4-1. Bluetooth Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\BLUETOOTH
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\BLUETOOTH
Driver DLL	bthbcsp.dll btp_bchs.dll btp_modules.dll bth_avdrv.dll
SDK Library	N\A
Catalog Item	Third Party –> BSP –> Freescale <*Target Platform*>: ARMV4I –> Device Drivers –> BlueTooth –> Bluetooth USB Adatper Third Party –> BSP –> Freescale <*Target Platform*>: ARMV4I –> Device Drivers –> USB Devices –> USB High Speed Host1–>High Speed Host1 Core OS –> CEBASE –> Communication Services and Networking –> Networking - Personal Area Network(PAN) –> Bluetooth –> Bluetooth Protocol Stack with Transport Driver Support –> Bluetooth Stack with Integrated USB Driver Core OS –> CEBASE –> Applications and Services Development –> .NET Compact Framework 2.0 > .NET Compact Framework 2.0 Core OS –> CEBASE –> Applications and Services Development –> Object Exchange Protocol(O–BEX) –> OBEX Client Core OS > CEBASE > Applications and Services Development –> Object Exchange Protocol(OBEX) –> OBEX Server –> OBEX File Browser Core OS –> CEBASE –> Applications and Services Development –> Object Exchange Protocol(OBEX) –> OBEX Server –> OBEX Inbox Core OS –> CEBASE –> Applications and Services Development –> Component Services(COM and DCOM) –> Component Object Model –> DCOM

SYSGEN Dependency	SYSGEN_BTH_USB_ONLY=1 SYSGEN_DOTNETV2=1 SYSGEN_OBEX_FILEBROWSER=1 SYSGEN_OBEX_CLIENT=1 SYSGEN_OBEX_INBOX=1
BSP Environment Variables	BSP_NOBLUETOOTH= BSP_USB_BLUETOOTH =1 BSP_USB_HSH1=1

The Catalog Items in Table 4-1 should be included in the OS design in order to provide Bluetooth Profiles.

NOTE: please select ***Clean Sysgen*** for the first building after add bluetooth features

4.2 Supported Functionality

The Bluetooth driver enables the MX51 EVK board to provide the following software and hardware support:

1. Drives BU 2073-J Bluetooth USB adapter
2. Provides communication between Bluetooth USB adapter and USB host driver
3. Supports A2DP SOURCE (Advanced Audio Distribution Profile)
4. Supports AVRCP (Audio Video Remote Control Profile)
5. Supports FTP server(File Transfer Profile)

4.3 Hardware Operation

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

4.3.1 Conflicts with Other Peripherals and Catalog Items

4.3.1.1 Conflicts with SoC Peripherals

4.3.1.1.1 i.MX51 Peripheral Conflicts

None

4.3.1.2 Conflicts with EVK Peripherals

4.3.1.2.1 i.MX51 EVK Peripheral Conflicts

None

4.4 Software Operation

The overall software architecture with existing Microsoft Bluetooth stack and CSR BCHS stack is shown in Figure 4-1

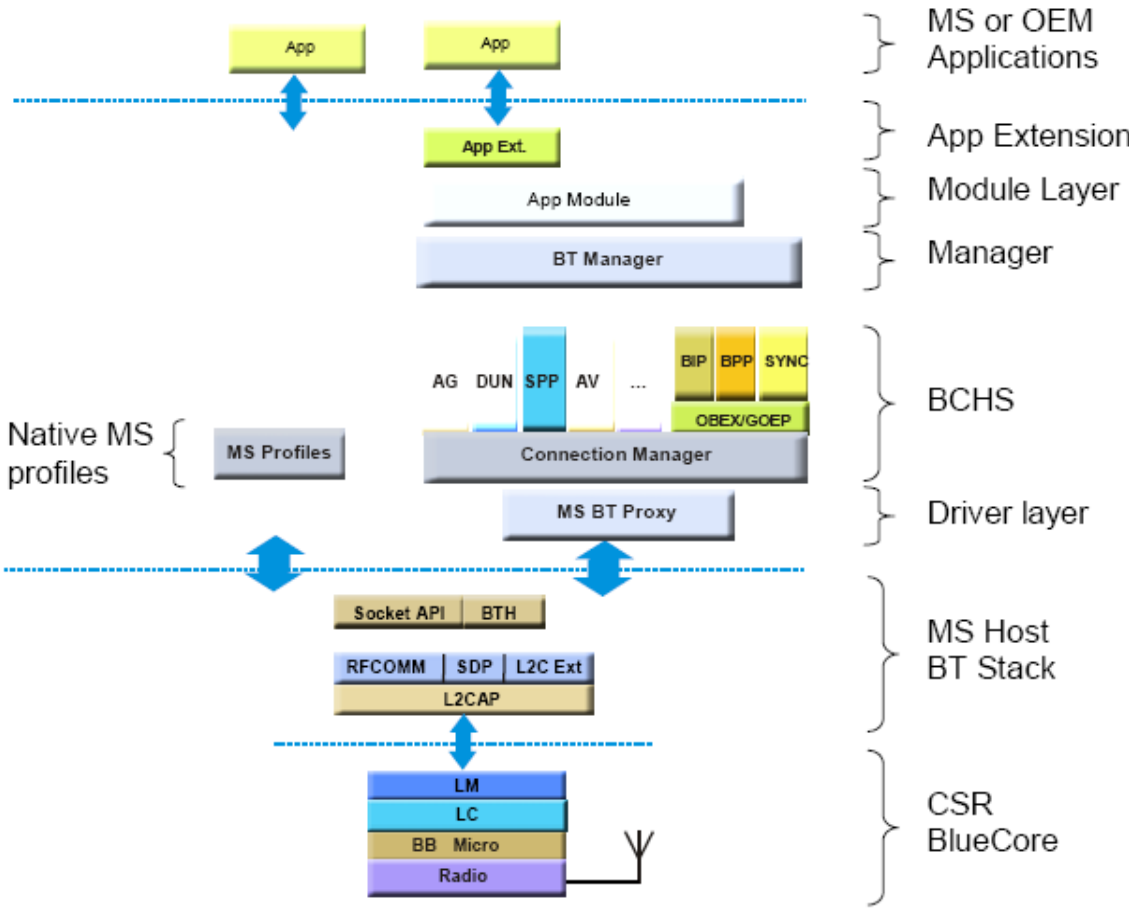


Figure 4-1. Software Architecture of Bluetooth Driver and Protocol

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

4.4.1 Registry Settings

4.4.1.1 i.MX51 Registry Settings

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

```
IF BSP_NOBLUETOOTH !
#if (defined BSP_CSR_BLUETOOTH || defined BSP_USB_BLUETOOTH)
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\bta_mp3player.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\bta_mp3player.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_av.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_bipc.reg"
```

```
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_bips.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_bpp.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_driver.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_dundrv.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_ftcm.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_ftsm.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_hfm.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_hidda.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_pacm.reg"
    #include "$(_TARGETPLATROOT)\SRC\DRIVERS\BLUETOOTH\btp_saps.reg"
#endif
Because Bluetooth Audio driver encapsulates the audio driver(wavedev2_stgl5000.dll) for A2DP
feature, the following registry must be included:
#if (defined BSP_CSR_BLUETOOTH || BSP_USB_BLUETOOTH)
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Wavedev]
    "olddll"="wavedev2_sgtl5000.dll"
    "Dll"="btp_avdrv.dll"
#else
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Audio]
   "Dll"="wavedev2_sgtl5000.dll"
#endif
NOTE: please don’t add BSP_CSR_BLUETOOTH variable into workspace
```

4.5 Unit Test

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

4.5.1 Unit Test Hardware

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

Table 4-2. Hardware Requirements

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

4.5.2 Unit Test Software

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

Table 4-3. Software Requirements

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

Table 4-3. Software Requirements

Requirement	Description
Netall.dll	Provides functions that generate random numbers, output data, and parse command lines
Btwsvr22.exe, Btw22.exe	CETK MS Bluetooth Test
bthapitst.dll	CETK Bluetooth API Test
Perflog.dll, Perf_bluetooth.dll	CETK Bluetooth Performance Test
hciqa_con.dll,ddlx.dll	CETK Bluetooth HCI Transport Driver Test

4.5.3 Running the Unit Tests

4.5.3.1 Running Bluetooth CETK

4.5.3.1.1 Running the CETK MS Bluetooth Test

The MS Test requires two Bluetooth boards with network feature: one for the client and one for the server. The test steps are as follows:

1. Bootup the two EVK boards with the adapter is plugged in, and FEC enabled and KITL disabled, and ensure the two boards are within the same network.
2. Copy Kato.dll, Tooltalk.dll, Netall.dll and Btwsvr22.exe into Windows directory in server board
3. Copy Kato.dll, Tooltalk.dll, Netall.dll and Btw22.exe into Windows directory in client board
4. In the EVK server board, modify the device name to “server” via the System tools in the Control Panel. Open btwsvr22.exe.
5. In the EVK client board, open **Run** from START and enter **tux -o -d btw22 -c “server”** command to execute this test.

4.5.3.1.2 Running the CETK Bluetooth API Test

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

1. Bootup the two EVK boards with the adapter is plugged in
2. Copy Kato.dll, Tooltalk.dll, Netall.dll and bthapitst.dll into the Windows directory in client board
3. In the client board, open **Run** from START and enter **tux -o -d bthapitst.dll -c**“**-s** *server_bt_addr*” command to execute this test. Where *server_bt_addr* is the Bluetooth address of the Windows Embedded CE based device running as a server. For example, if the server address is 0123456789ab, the command line should read: **tux -o -d bthapitst.dll -c“-s 0123456789ab”.**

4.5.3.1.3 Running the CETK Bluetooth Performance Test

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

1. Bootup two EVK boards with the adapter is plugged in.

2. Copy Kato.dll, Tooltalk.dll, Perflog.dll and Perf_bluetooth.dll into the Windows directory in both boards
3. In the server board, open **Run** from START and enter **tux -o -d perf_bluetooth -c "-i** *NumberOfIteraions* **-b** *NumberOfBuffers* **-p** *ServerChannelNumber*" command to execute this test. Such as **tux -o -d perf_bluetooth -c "-i 10 -p 6 -b 163840"**
4. In the client board, open **Run** from START and enter **tux -o -d perf_bluetooth -c** "**-s** *server_bt_addr* **-i** *NumberOfIteraions* **-b** *NumberOfBuffers* **-p** *ServerChannelNumber*" command to execute this test. Such as **tux -o -d perf_bluetooth -c "-s 0123456789ab -i 10 -p 6 -b 163840"**

To view the test results:

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

4.5.3.1.4 Running the CETK Bluetooth HCI Transport Driver Test

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

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

NOTE

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

4.5.3.2 Manual Test Bluetooth

NOTE

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

4.5.3.2.1 Running the Bluetooth A2DP Test

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

1. Make sure Bluetooth usb adapter plugged in

2. Make Bluetooth headset entering *pairing* mode
3. Open the Bluetooth Device Properties tools in the control panel and click ***scan device*** icon
4. If your headset is scanned, a dialog box appears in top left of the screen. Quickly enter the default password “0000“ in the Authentication Request dialog box for your Bluetooth headset, then click OK. In the Bluetooth Manager window, you will see your Bluetooth headset item, as in Figure 4-2.

NOTE

Ensure the Bluetooth headset icon is correct and not a question mark. Otherwise repeat the steps above.

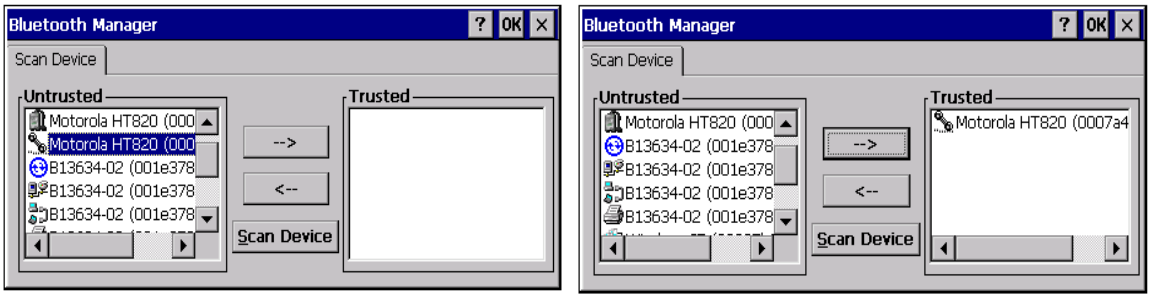


Figure 4-2. Bluetooth A2DP Test

5. After the Bluetooth headset is selected (click A2DP icon), click –> to move the Bluetooth headset icon into right block. Click **No** in the dialog box. Double-click the icon in the right block and select **Active** (as in Figure 4-3). A red check mark should be marked on your Bluetooth headset icon. You may close the Bluetooth Manager window.

NOTE

If you move Bluetooth headset from the right block to the left block before the headset is activated, do not move the headset to the right blockagain. This will disable the Bluetooth function. Repeat steps 1-3 above to repair/retrust/active the headset.

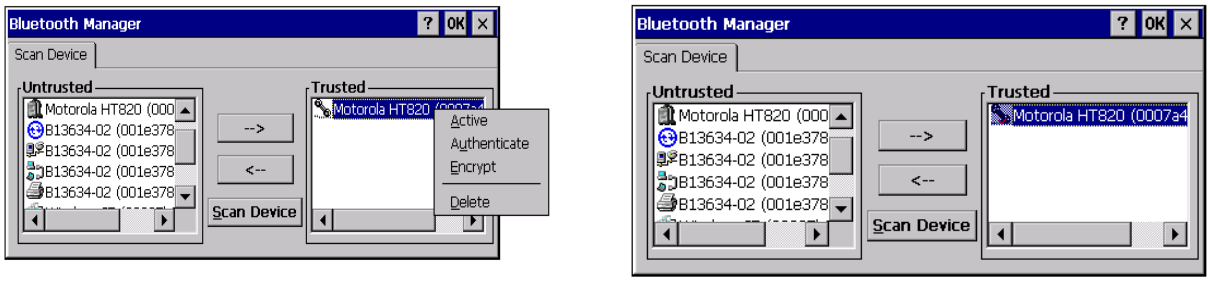


Figure 4-3. Bluetooth A2DP Test

6. The EVK board sets up the audio connection with Bluetooth headset. Music played by Media Player, may be listened from your Bluetooth headset.

NOTE

Before plug out adapter, must close mediaplayer or other audio playing applications and then delete A2DP icon from right trusted window

4.5.3.2.2 Running the Bluetooth AVRCP Test

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

4.5.3.2.3 Running the Bluetooth FTP Server Test

1. Enable Bluetooth function in mobilephone
2. Select the sent file and select send by Bluetooth. Then you may search Bluetooth device, and if the i.MX51 platform is scanned, then send this file.
3. If a window jumps in i.MX51 platform, and enquires “Yes“ or “No“ about receiving this file, please click “Yes“. This file will be transfered and saved under My Document

Note: different mobilephone sand laptops, the operation steps are different. Please according the instruction to operate.

4.5.4 Operation Attention Items and Tips

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

- Ensure that the Bluetooth headset is in pairing mode, then begin to scan the device from the Bluetooth Manager Window
- After the Bluetooth headset is scanned, a password window appears. Quickly input the default password ‘0000’. If the headset icon in Bluetooth Manager window is a question mark and headset is not in pair mode, the password was inputted to slow. Set the headset in pair mode and re-scan. If the headset icon in the Bluetooth Manager window is a question mark, and headset is in pair mode, the password is incorrect. In this case, trust the headset icon in the manager window, then un-trust it to delete the headset password information. Then set the headset in pair mode and re-scan it and input the correct password.
- Pay attention to the A2DP which may refer to WINCE600\PUBLIC\COMMON\OAK\DRIVERS\NETUI and use the right icon.
- Before plug out bluetooth usb adapter, must close mediaplayer or other audio playing applications and then delete A2DP icon from right trusted window. Also delete other actived icons.

4.5.5 Known Issues

- If you move the Bluetooth headset from the right block to the left block before the headset is activated, do not again move the headset to the right block. This confuses the Bluetooth. The correct operation is the following steps after headset is removed into left block

  - Ensure headset in pair mode
  - Rescan and input password
  - Move headset into right block and active it.

  The reason is that the Bluetooth Property Application provided by Microsoft deletes the trusted Bluetooth headset security register.
- When scan is running, do not reopen the Bluetooth Property Application in the control panel, otherwise the Bluetooth Property Application will be in an unexpected state, such as cannot stop or cannot reopen if you close this window. This reason is that Bluetooth Property Application provided by Microsoft is not handled well for unique instance.
- Bluetooth API CETK fails in hold mode test.

Chapter 5
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 96 Mbytes.

5.1 Boot from SD/MMC Summary

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

Table 5-1. Boot from SD/MMC Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX51-EVK
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\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

5.2 Supported Functionality

The boot support from SD/MMC includes:

1. Boot from low or high capacity SD/MMC card at least 96 Mbytes in size 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 on TO1.1 and later
7. eSD2.1 and eMMC 4.3 boot from boot partition if boot partition can be configured to be at least 3664 Mbytes in size; otherwise, boot from user partition on these devices is supported

5.3 Hardware Operation

5.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts for eSDHC1 with other on-chip peripherals.

5.4 Software Operation

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

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

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

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

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

5.4.1 Card Memory Layout

SD cards that do not meet the v2.1 spec and MMC cards that do not meet the v4.3 spec have only one physical partition. To allow storage of boot images as well as file system on these card, EBOOT can add a partition table (MBR) to the card that reserves the initial 96 Mbytes for boot images (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 TO2, 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.

5.4.1.1 i.MX51 Card Memory Layout

Figure 5-1 shows the card memory layout for the i.MX51.

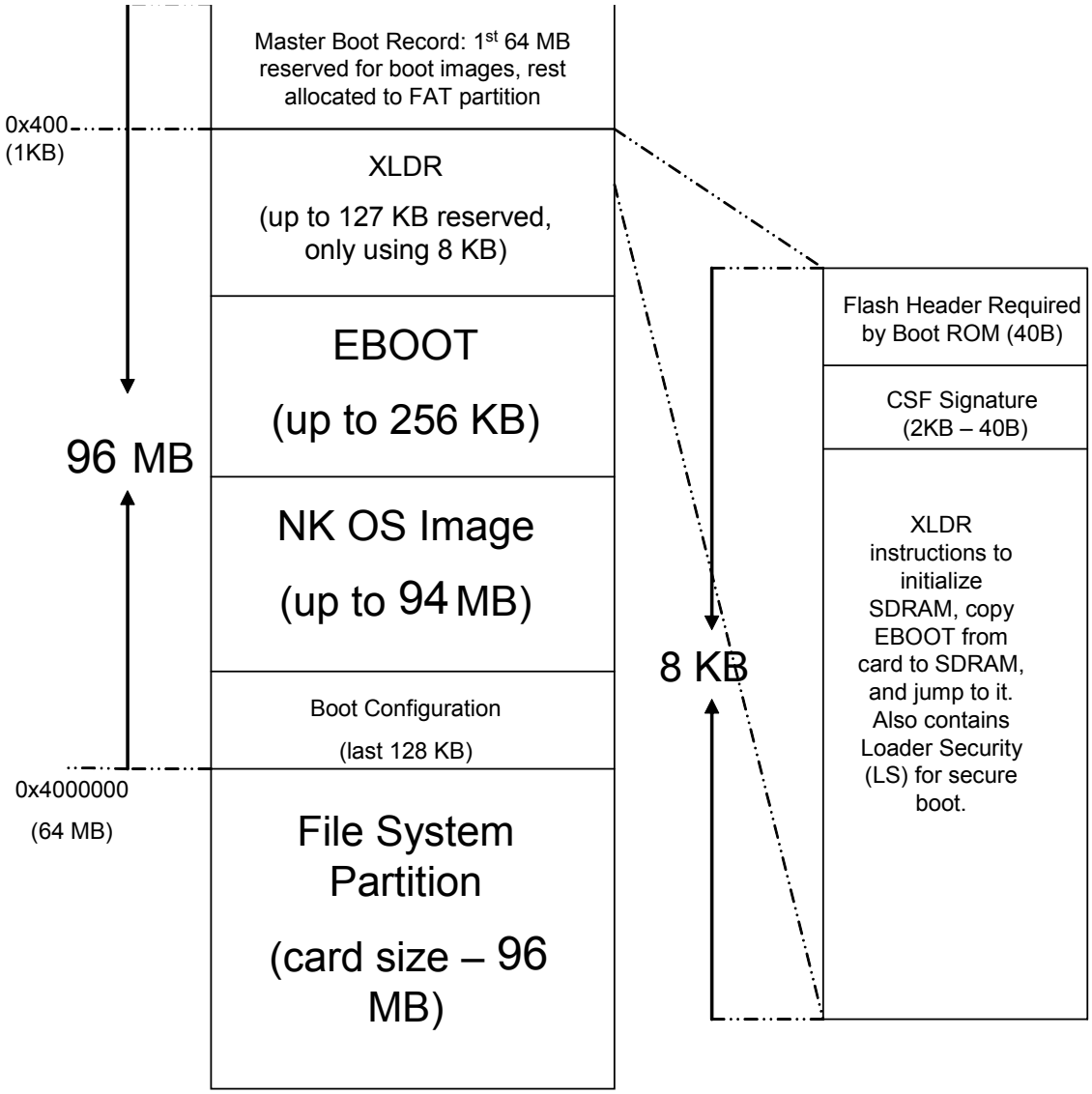


Figure 5-1. Card Memory Layout

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

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

Chapter 6
Camera Driver for IPUv3

The camera driver is based on the Windows CE Camera Device Driver Interface. This interface provides basic support for video and still image capture devices. The camera driver conforms to the architecture for Windows CE stream interface drivers and can support two camera instances, It allows applications to use the middleware layer provided by the DirectShow video capture infrastructure to communicate with and control the camera.

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

- Communicating with and configuring the camera device through the HI2C interface
- Configuring the submodules (CSI, SMFC and so on) of the Image Processing Unit v3 (IPUv3) for captured images
- Performing post-processing tasks with IPUv3 for the video preview data

The camera driver is compatible with the camera sensor OV3640 and requires the MCIMX51EXP expansion board for the camera interface.

The camera driver can support two camera instances. Camera1 use sensor OV3640, Camera2 use CSI test mode. Of course, if want use other sensor mode, the sensor special control code must be implemented and driver register `"CameraId"` must be changed.

Sensor special control code for OV5642 and ADV7180 is exist in current camera driver, so they can be supported by current camera driver in software level, if want to enable them, hardware supported in board level is needed.

6.1 Camera Driver Summary

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

Table 6-1. Camera Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\IPUV3\CAMERA
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\IPUV3\CAMERA
Driver DLL	camera.dll
SDK Library	N/A

Table 6-1. Camera Driver Summary (continued)

Catalog Item	Third Party > BSP > 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_I2CBUS1 = 1 BSP_PP = 1 For Camera driver 1: BSP_CMOS_OV3640 = 1 or BSP_CMOS_OV5642 = 1(For OV5642 sensor support, need hareware support) or BSP_TVIN_ADV7180 = 1(For ADV7180 Tvin support, need hareware support) For Camera driver 2: BSP_CSI_TESTMODE = 1

6.2 Supported Functionality

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

1. Windows CE Camera Device Driver Interface
2. Preview and Capture/Sill pins for camera1 application
3. Capture/Sill pins for camera2 application
4. OV3640 camera sensor for camera1 driver
5. Format from sensor output to CSI input (RGB888, RGB565, YUV422)
6. Output resolution for Preview and Still pin
   - 640×480 for VGA
   - 320×240 for QVGA
   - 160×120 for QQVGA
   - 352×288 for CIF
   - 174×144 for QCIF
7. Output resolution for Capture pin
   - 720×576 for PAL
   - 720×480 for NTSC
   - 640×480 for VGA
   - 320×240 for QVGA
   - 160×120 for QQVGA
   - 352×288 for CIF

— 174×144 for QCIF

8. Output format for Preview pin (RGB565, UYVY)
9. Output format for Still pin (RGB565, YV12, UYVY)
10. Output format for Capture pin (YV12, NV12, UYVY, RGB565)

6.3 Hardware Operation

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

For detailed operation and programming information, see the chapter on the Image Processing Unit (IPUv3) in the i.*MX51 Applications Processor Reference Manual*.

6.3.1 IPUv3 Overview

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

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

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

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

6.3.2 Conflicts with Other Peripherals and Catalog Items

6.3.2.1 Conflicts with SoC Peripherals

No conflicts.

6.3.2.2 i.MX51 Peripheral Conflicts

No conflicts.

6.4 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.

6.4.1 Software Architecture

6.4.1.1 Software Driver Components

Figure 6-1 shows the relationship between software components in the camera driver architecture.

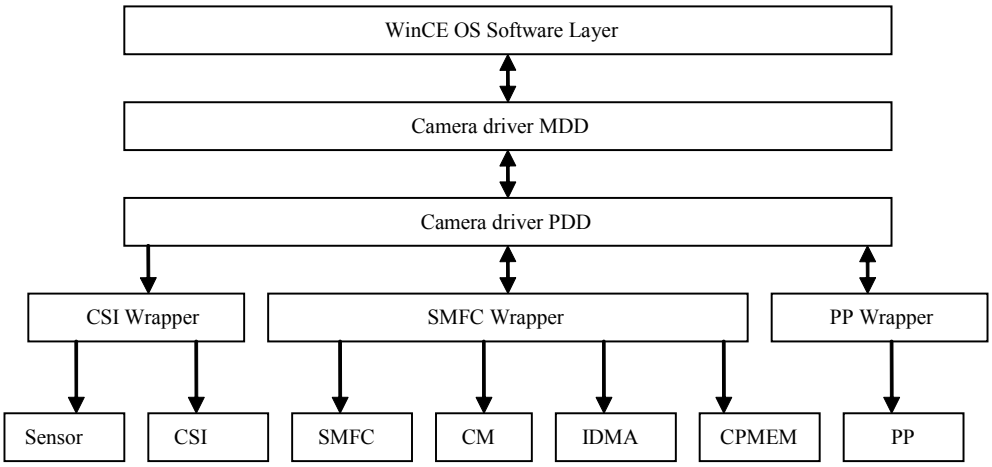


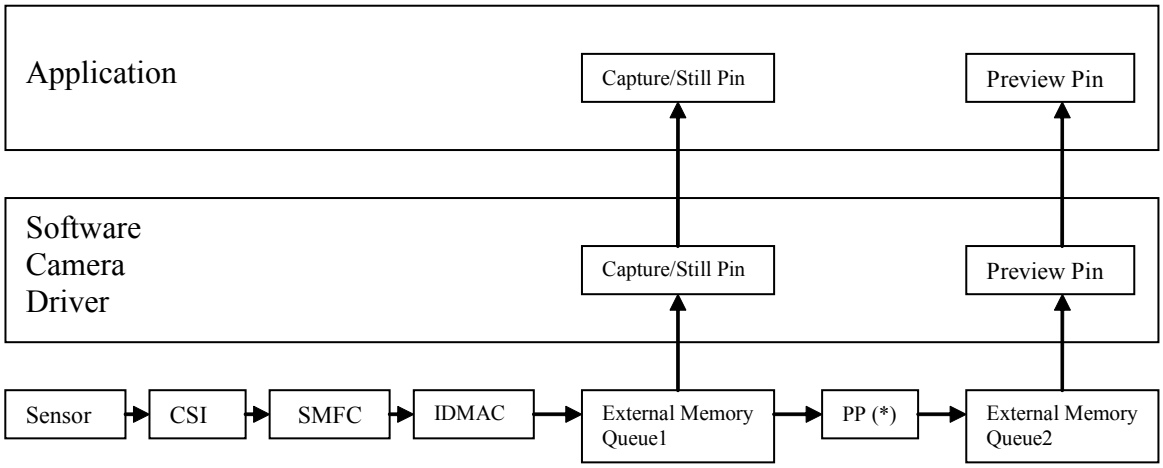
Figure 6-1. Camera Driver Architecture

Figure 6-1 shows the following main elements of the camera driver architecture:

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

6.4.1.2 Data Flow

Figure 6-2 shows the data flow of the camera driver. The sensor passes the frame data to the CSI module, which then passes the data to the SMFC. The SMFC sets up the data for the IDMAC. The camera driver sets a pointer to an external memory buffer which is filled by the DMA after the IDMAC is complete. The camera driver uses the frame data in the external memory as the Capture/Still Pin output. Simultaneously, this frame data is used as the PP input for the Color Space Conversion (CSC), size change, and rotation/flip/mirror operation. The camera driver uses the PP output as the Preview Pin output. Since the frame data in the Capture/Still Pin does not pass the PP module, rotation, flip, or mirror operations cannot be achieve on the Capture/Still Pin.



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

Figure 6-2. Camera Driver Data Flow

NOTE

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

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

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

6.4.1.3 Buffer Management

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

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

6.4.1.3.1 Buffer Allocated by the Driver

If the camera pin is running under CSPROPERTY_BUFFER_DRIVER mode, buffers are allocated by the driver. The buffer state includes three mode: Idle, Busy, and Filled. The camera driver uses a queue to keep the buffer state, which means if one buffer is in the Idle Queue, it is in the Idle State. Figure 6-3 shows the buffer state diagram for this mode.

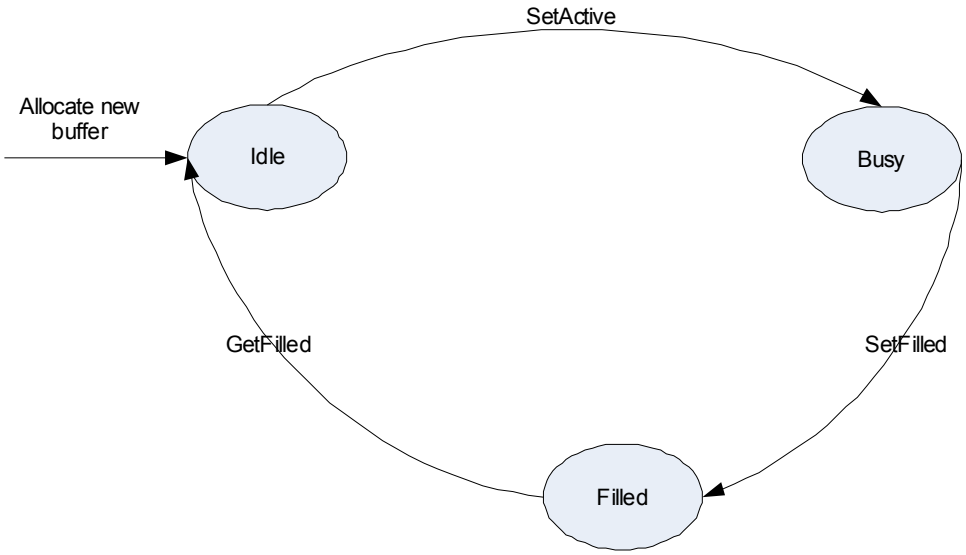


Figure 6-3. CSPROPERTY_BUFFER_DRIVER Mode Buffer State Diagram

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

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

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

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

6.4.1.3.2 Buffer Allocated by the Client

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

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

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

Figure 6-4 shows the buffer state diagram for this mode.

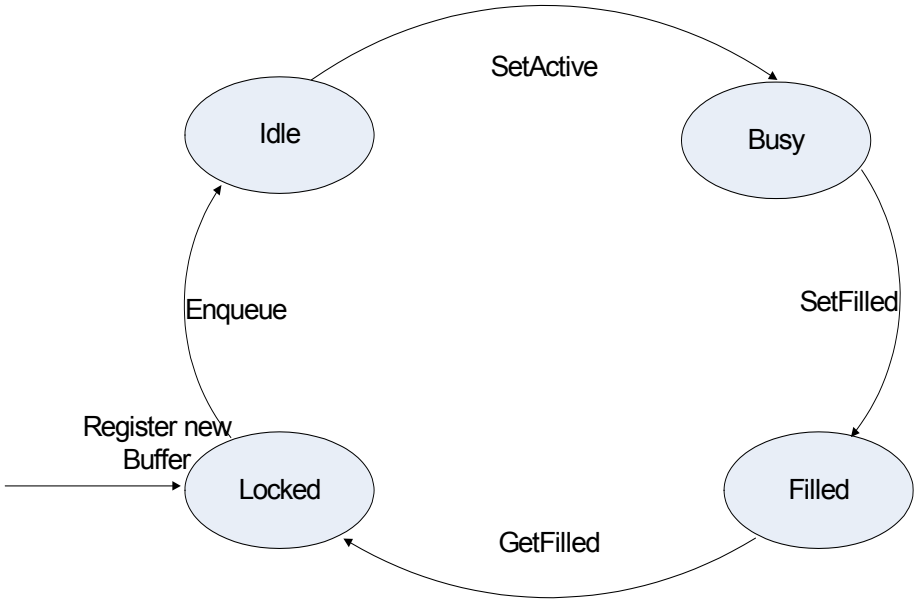


Figure 6-4. CSPROPERTY_BUFFER_CLIENT_UNLIMITED Mode Buffer State Diagram

The following steps describe the procedure of client allocating buffer:

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

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

6.4.2 Communicating with the Camera

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

6.4.2.1 Using the Windows CE Video Camera Device Driver Interface

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

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

6.4.2.2 Using DirectShow for Video Capture

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

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

6.4.3 Sensor Frame Rate Setting

Camera driver can support two frame rates: 15 fps and 30 fps. The default setting is 15 fps.

The frame rate can be changed by calling DeviceIoControl with IOCTL_CS_PROPERTY and specifying the property set PROPSETID_VIDCAP_VIDEOCONTROL and the property ID CSPROPERTY_VIDEOCONTROL_FRAME_RATES.

6.4.4 Registry Settings

Two sets of registry settings are important for proper camera driver operation. One set is for the camera driver and the other is for the DirectShow Capture Pins. This section describes the registry keys used to select the camera sensor used on the SoC.

6.4.4.1 i.MX51 Registry Settings

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

```
#if (defined BSP_CMOS_OV3640 || defined BSP_CMOS_OV5642 || defined BSP_TVIN_ADV7180)
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Camera1]
   "Prefix"="CAM"
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:1

IF BSP_CMOS_OV3640
   "CameraId"=dword:0
ENDIF BSP_CMOS_OV3640

IF BSP_CMOS_OV5642
   "CameraId"=dword:2
ENDIF BSP_CMOS_OV5642

IF BSP_TVIN_ADV7180
   "CameraId"=dword:4
ENDIF BSP_TVIN_ADV7180

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

[HKEY_LOCAL_MACHINE\Software\Microsoft\DirectX\DirectShow\Capture1]
   "Prefix"="PIN"
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:1
   "PinCount"=dword:3 ;Pin count. Max = 3; default = 2
   "MemoryModel"=dword:1 ; Pin memory mode.
   "IClass"=multi_sz:"{C9D092D6-827A-45E2-8144-DE1982BFC3A8}",
                     "{A32942B7-920C-486b-B0E6-92A702A99B35}"
#endif

#if (defined BSP_CSI_TESTMODE)
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\Camera2]
   "Prefix"="CAM"
```

```
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:2

IF BSP_CSI_TESTMODE
   "CameraId"=dword:9
ENDIF BSP_CSI_TESTMODE

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

[HKEY_LOCAL_MACHINE\Software\Microsoft\DirectX\DirectShow\Capture2]
   "Prefix"="PIN"
   "Dll"="camera.dll"
   "Order"=dword:20
   "Index"=dword:2
   "PinCount"=dword:2 ;Pin count. Max = 3; default = 2
   "MemoryModel"=dword:1 ; Pin memory mode.
   "IClass"=multi_sz:"{C9D092D6-827A-45E2-8144-DE1982BFC3A8}",
                     "{A32942B7-920C-486b-B0E6-92A702A99B35}"
#endif
```

6.5 Power Management

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

6.5.1 PowerUp

This function is not implemented for the camera driver.

6.5.2 PowerDown

This function is not implemented for the camera driver.

6.5.3 IOCTL_POWER_SET

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

These states are handled in the following manner:

- D0—Action is only taken when resuming from the D4 state. If the camera is running when the transition to the D4 state occurs, the camera returns to a running state, re-enabling the sensor and IPUv3 submodules.
- D4—Action is only taken if the camera is running when the request to transition to the D4 state occurs.

6.6 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 following:

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

Additionally, for Windows Embedded CE 6.0, a camera application may be used to preview and capture images.

6.6.1 Unit Test Hardware

Table 6-2 lists the required hardware to run the unit tests.

Table 6-2. Hardware Requirements

Requirement	Description
Camera sensor	Expansion Board with OV3640 Camera Sensor

6.6.2 Unit Test Software

6.6.2.1 CETK Test

Table 6-3 lists the required software to run the camera test.

Table 6-3. 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

Table 6-3. Software Requirements

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

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

6.6.2.2 Custom Camera Test

The `camapp.exe` executable file is needed to run the custom camera application.
The `camapp1_preview.exe` and `camapp2_capture.exe` executable files are needed to validate dual camera driver.

6.6.2.3 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.

6.6.3 Building the Unit Tests

6.6.3.1 CETK Test

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

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

6.6.3.2 Custom Camera Application Test

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

Build an OS image for the desired configuration:

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

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

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

Change

```
//#define DIRECT_DRAW_MODE
```

to

```
#define DIRECT_DRAW_MODE
```

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

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

1. Select the Solution Explorer of the Platform Builder Workspace window
2. Select Subprojects in Solution Explorer
3. Right-click Subprojects and select Add Existing Subproject to add the CAMAPP project
4. Right-click on the CAMAPP project and select Rebuild

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

If want to validate dual camera driver, please build the dual camera application following steps:

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

The Dual camera application execution files (camapp1_preview.exe camapp2_capture.exe) are created copied to the workspace release directory.

6.6.4 Running the Unit Tests

6.6.4.1 Running the Camera Unit Tests

6.6.4.1.1 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** > **CETK Tests** > **Camera Tests**

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

```
tux –o –d CameraGraphTests.dll -X!508
```

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

Please run camera CETK for Camera1 and Camera2 separately. If run CETK for Camera1, make sure BSP_CAMERA2 isn't selected, and when run for Camera2, make sure BSP_CAMERA1 isn't selected.

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

```
SYSGEN_IMAGING_BMP_ENCODE
SYSGEN_IMAGING_JPG_ENCODE
SYSGEN_IMAGING_BMP_DECODE
SYSGEN_IMAGING_JPG_DECODE
SYSGEN_DSHOW_DISPLAY
SYSGEN_DSHOW_CAPTURE
SYSGEN_DSHOW_DMO
SYSGEN_DSHOW_VIDREND
```

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

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

Some CETK Camera and DirectShow Integration Test fail:

- 308 and 309 fail on MX51 TO2 platform, because Microsoft DirectShow can not support the NV12 format which can be supported by the i.MX51 camera driver Preview pin.
  But on TO3 platform, these two subcase can pass. Because on TO3, preview pin support YV12 format instand of NV12.
- 508 fails because of a CETK code problem. In the CETK code file `captureframework.cpp` line 3619, after the capture pin runs for 1, 2, 4, and 7 s, the test waits MAXIMUM_MEDIAEVENT_TIMEOUT = 3 min to let the graph encode video. If in three minutes, the graph does not complete, CETK assumes there is problem. But in fact, when the frame size is large and the frame rate is high, encoding takes longer than three minutes to finish. On the i.MX51 platform, the camera driver supports

720×576 size frame and up to 30 fps. So when using large frame size and high frame rate, the CETK case 508 fails.

- For the i.MX51 EVK, when configure 512M RAM at 30fps, 510 fails. In 510 sub-case testing combination four, the video encode buffer depth is set to MAX, so all system memory is used to store video frames. On the i.MX51 EVK, RAM memory size is 512 Mbytes by default, so when camera is running at 30fps, it can store too many video frames to let the graph encode complete in three minutes. This is the same reason as test case 508. If the RAM is configured to 256 Mbytes or camera is running at 15fps, this case passes.

6.6.4.1.2 Running the Custom Camera Application Test

The following command executes the Custom Camera Application:

```
camapp.exe
```

6.7 Camera Driver API Reference

For the camera driver API reference, see the Windows Embedded CE 6.0 documentation. 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 7
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)

7.1 CSPDDK Driver Summary

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

Table 7-1. CSPDDK Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFROM\COMMON\SRC\SOC\COMMON_FSL_V2\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=

7.2 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)

7.3 Hardware Operation

Refer to the *i.MX51 Applications Processor Reference Manual* for detailed operation and programming information.

7.3.1 Conflicts with Other Peripherals and Catalog Items

7.3.1.1 Conflicts with SoC Peripherals

Refer to the *i.MX51 Applications Processor Reference Manual* for possible conflicts.

7.3.1.2 Conflicts with Board Peripherals

No conflicts.

7.4 Software Operation

7.4.1 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.

7.4.2 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 7-2 describes the available CSPDDK compile options.

Table 7-2. 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.

Table 7-2. CSPDDK Compile Options (continued)

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

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

To decrease power consumption in system uses cases such as audio playback, it is beneficial to configure BSP_SDMA_MC0PTR to point to a reserved internal RAM (IRAM) region and allocate the audio buffers in IRAM. This configuration does not require external memory cycles in the data flow from the audio buffers to the SSI and allows the CSPDDK to utilize EMI clock gating to significantly reduce the power consumption. Refer to Chapter 2, “Audio Driver,” for more information on configuring audio DMA support.

7.4.3 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.

7.4.4 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.

7.5 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.

7.5.1 Unit Test Hardware

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

Table 7-3. Hardware Requirements

Requirement	Description
No additional hardware required	

7.5.2 Unit Test Software

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

Table 7-4. 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

7.5.3 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\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.

7.5.4 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 7-5 describes the test cases contained in the DDK_SDMA tests.

Table 7-5. 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.

7.6 CSPDDK DLL Reference

7.6.1 CSPDDK DLL System Clocking (DDK_CLK) Reference

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

7.6.1.1 DDK_CLK Enumerations

Table 7-6. DDK_CLK Enumerations

Programming Element	Description
DDK_CLOCK_SIGNAL	Clock signal name for querying/setting clock configuration
DDK_CLOCK_GATE_INDEX	Index for referencing the corresponding clock gating control bits in the CCM
DDK_CLOCK_GATE_MODE	Clock gating modes supported by CCM clock gating registers
DDK_CLOCK_BAUD_SOURCE	Input source for baud clock generation
DDK_CLOCK_CKO1_SRC	Clock output source one (CKO1) signal selections
DDK_CLOCK_CKO2_SRC	Clock output source two (CKO2) signal selections
DDK_CLOCK_CKO_DIV	Clock output source (CKO) divider selections

Table 7-6. DDK_CLK Enumerations (continued)

DDK_CLOCK_OVERRIDE_ENABLE_INDEX	Index for referencing the corresponding clock enable signal to be overridden
DDK_CLOCK_OVERRIDE_MODE	Clock enable signal override mode supported by CCM Enable Override Register
DDK_DVFC_SETPOINT	Frequency/voltage setpoints supported by the DVFC driver

7.6.1.2 DDK_CLK Functions

7.6.1.2.1 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

7.6.1.2.2 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

7.6.1.2.3 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

7.6.1.2.4 DDKClockSetFreq

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

```
BOOL DDKClockSetFreq(
        DDK_CLOCK_SIGNAL sig,
        UINT32 freq)
```

Parameters

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

7.6.1.2.5 DDKClockConfigBaud

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

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

Parameters

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

7.6.1.2.6 DDKClockSetCKO1

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

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

Parameters

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

7.6.1.2.7 DDKClockSetCKO2

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

```
BOOL DDKClockSetCKO2(
        BOOL bEnable,
        DDK_CLOCK_CKO2_SRC index,
```

```
        DDK_CLOCK_CKO_DIV div)
```

Parameters

bEnable	[in] Set to TRUE to enable CKO2 output; set to FALSE to disable CKO2 output
index	[in] Selects the CKO2 source signal
div	[in] Specifies the CKO2 divide factor
Return Values	Returns TRUE if successful, otherwise returns FALSE

7.6.1.2.8 DDKClockSetOverride

This function sets the override mode for clock enable mode.

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

Parameters

index	[in] Index for referencing the clock enable signal
mode	[in] Requested override mode for the clock enable signal
Return Values	Returns TRUE if successful, otherwise returns FALSE

7.6.1.2.9 DDKClockGetOverride

This function gets the override mode for clock enable mode.

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

Parameters

index	[in] Index for referencing the clock enable signal
pMode	[out] Pointer to the buffer to save current override model for clock enable signal
Return Values	Returns TRUE if successful, otherwise returns FALSE

7.6.1.2.10 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

7.6.1.2.11 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

7.6.1.2.12 DDKClockGetSharedConfig

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

```
PDDK_CLK_CONFIG DDKClockGetSharedConfig(VOID)
```

Parameters None

Return Values Returns a pointer to the clock configuration data structure

7.6.1.2.13 DDKClockLock

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

```
VOID DDKClockLock(VOID)
```

Parameters None

Return Values None

7.6.1.2.14 DDKClockUnLock

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

```
VOID DDKClockUnLock(VOID)
```

Parameters None

Return Values None

7.6.1.3 DDK_CLK Examples

Example 7-1. 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 7-2. CSPDDK Clock Rate Query

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

UINT32 freq;

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

7.6.2 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.

7.6.2.1 DDK_GPIO Enumerations

Table 7-7. 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

7.6.2.2 DDK_GPIO Functions

7.6.2.2.1 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

7.6.2.2.2 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

7.6.2.2.3 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

7.6.2.2.4 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

7.6.2.2.5 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

7.6.2.2.6 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

7.6.2.2.7 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

7.6.2.2.8 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

7.6.2.3 DDK_GPIO Example

Example 7-3. 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);
```

7.6.3 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.

7.6.3.1 DDK_IOMUX Enumerations

Table 7-8. 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_INMODE	Specifies the CMOS/open drain mode for a pad; if no DDR_INPUT bit for a PAD, the DDK_IOMUX_PAD_INMODE_NULL should be set
DDK_IOMUX_PAD_HYSTERESIS	Hysteresis mode for a pad; if no HYS bit for a PAD, the DDK_IOMUX_PAD_HYSTERESIS_NULL should be set
DDK_IOMUX_PAD_OUTVOLT	Specifies the output voltage mode for a pad; if no HVE bit for a PAD, the DDK_IOMUX_PAD_OUTVOL_NULL should be set
DDK_IOMUX_PAD_PULL	Pull-up/pull-down/keeper configuration for a pad
DDK_IOMUX_SELECT_INPUT	Functional pad name to be selected and involved in Daisy Chain

7.6.3.2 DDK_IOMUX Functions

7.6.3.2.1 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

7.6.3.2.2 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

7.6.3.2.3 DDKIomuxSetPadConfig

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

```
BOOL DDKIomuxSetPadConfig(
         DDK_IOMUX_PAD pad,
         DDK_IOMUX_PAD_SLEW slew,
         DDK_IOMUX_PAD_DRIVE drive,
         DDK_IOMUX_PAD_OPENDRAIN openDrain,
         DDK_IOMUX_PAD_PULL pull,
         DDK_IOMUX_PAD_HYSTERESIS hysteresis,
         DDK_IOMUX_PAD_INMODE inputMode,

         DDK_IOMUX_PAD_OUTVOLT outputVol)
```

Parameters

pad	[in] Functional pad name used to select the pad that is configured
slew	[in] Slew rate configuration

drive	[in] Drive strength configuration
openDrain	[in] Open drain configuration
pull	[in] Pull-up/pull-down/keeper configuration
hysteresis	[in] Hysteresis configuration
inputMode	[in] Input mode (CMOS/DDR) configuration
outputVolt	[in] Output voltage configuration

Return Values Returns TRUE if successful, otherwise returns FALSE.

7.6.3.2.4 DDKIomuxGetPadConfig

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

```
BOOL DDKIomuxSetPadConfig(
         DDK_IOMUX_PAD pad,
         DDK_IOMUX_PAD_SLEW *pSlew,
         DDK_IOMUX_PAD_DRIVE *pDrive,
         DDK_IOMUX_PAD_OPENDRAIN *pOpenDrain,
         DDK_IOMUX_PAD_PULL *pPull,
         DDK_IOMUX_PAD_HYSTERESIS *pHysteresis,
         DDK_IOMUX_PAD_INMODE *pInputMode,

         DDK_IOMUX_PAD_OUTVOLT *pOutputVol)
```

Parameters

pad	[in] Functional pad name used to select the pad that is configured
pSlew	[out] Slew rate configuration
pDrive	[out] Drive strength configuration
pOpenDrain	[out] Open drain configuration
pPull	[out] Pull-up/pull-down/keeper configuration
pHysteresis	[out] Hysteresis configuration
pInputMode	[out] Input mode (CMOS/DDR) configuration
pOutputVolt	[out] Output voltage configuration

Return Values Returns TRUE if successful, otherwise returns FALSE.

7.6.3.2.5 DDKIomuxSetGpr0

This function writes a value into IOMUX GPR0.

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

Parameters

data [in] Data to be written

Return Values Returns TRUE if successful, otherwise returns FALSE

7.6.3.2.6 DDKIomuxGetGpr0

This function read a value from IOMUX GPR0.

```
UINT32 DDKIomuxGetGpr0(VOID)
```

Return Values Returns IOMUX GPR0 value

7.6.3.2.7 DDKIomuxSetGpr1

This function writes a value into IOMUX GPR1.

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

Parameters

data [in] Data to be written

Return Values Returns TRUE if successful, otherwise returns FALSE

7.6.3.2.8 DDKIomuxGetGpr1

This function read a value from IOMUX GPR1.

```
UINT32 DDKIomuxGetGpr1(VOID)
```

Return Values Returns IOMUX GPR1 value

7.6.3.2.9 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

7.6.3.3 DDK_IOMUX Examples

Example 7-4. 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 7-5. CSPDDK IOMUX Pad Configuration

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

// Configure the GPIO1_5 pad for the following configuration: fast slew rate,
// high drive strength, and remainder fields are invalid for GPIO1_5.
DDKIomuxSetPadConfig(DDK_IOMUX_PIN_GPIO1_5, DDK_IOMUX_PAD_SLEW_FAST,
DDK_IOMUX_PAD_DRIVE_HIGH, DDK_IOMUX_PAD_OPENDRAIN_NULL, DDK_IOMUX_PAD_PULL_NULL,
DDK_IOMUX_PAD_HYSTERESIS_NULL, DDK_IOMUX_PAD_INMODE_NULL,
DDK_IOMUX_PAD_OUTPUT_NULL);
```

7.6.4 CSPDDK DLL SDMA (DDK_SDMA) Reference

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

7.6.4.1 DDK_SDMA Enumerations

Table 7-9. 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

7.6.4.2 DDK_SDMA Functions

7.6.4.2.1 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

7.6.4.2.2 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

7.6.4.2.3 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

7.6.4.2.4 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

7.6.4.2.5 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

7.6.4.2.6 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

7.6.4.2.7 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

7.6.4.2.8 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

7.6.4.2.9 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

7.6.4.2.10 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

7.6.4.2.11 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

7.6.4.2.12 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

7.6.4.2.13 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 8
Display Driver for IPUv3

The Windows Embedded CE 6.0 BSP display driver is based on the Microsoft DirectDraw Graphics Primitive Engine (DDGPE) classes and supports the Microsoft DirectDraw interface. This driver combines the functionality of a standard LCD display with DirectDraw support. The display driver interfaces with the Image Processing Unit v3 (IPUv3).

The i.MX51 EVK supports the following display types:

- DVI digital output
- Mitsubisihi SVGA LVDS panel
- Chunghwa CLAA070VC01 WVGA LCD panel
- VGA analog output
- D1 TV Output following the NTSC or PAL television standard
- 720p TV Output following the 720p60 or 720p50 television standard
- 1080i TV Output following the 1080i30 television standard

8.1 Display Driver Summary

Table 8-1 identifies the source code location, library dependencies and other BSP information.

Table 8-1. Display Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\IPUV3\DISPLAY
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SoC*>\IPUV3\DISPLAY
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\IPUV3\DISPLAY
Driver DLL	ddraw_ipu.dll
SDK Library	N/A

Table 8-1. Display Driver Summary (continued)

Catalog Items	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Display > Display Port0 > IPU Support for DVI Output Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Display > Display Port0 > IPU Support for the Mitsubishi LVDS Panel Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Display > Display Port1 > IPU Support for the Chunghwa WVGA Panel Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Display > Display Port1 > IPU Support for VGA output Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > Display > Display Port1 > TVE Output Support
SYSGEN Dependency	SYSGEN_DDRAW=1
BSP Environment Variables	BSP_NODISPLAY= BSP_DISPLAY_DVI_TFP410 = 1 for DVI Output BSP_DISPLAY_LVDS_MITSUBISHI_AA084XA03 = 1 for Mitsubishi LVDS Panel BSP_DISPLAY_CHUNGHWA_CLAA070VC01 = 1 for Chunghwa WVGA Panel BSP_DISPLAY_VGA = 1 for VGA Output BSP_TVE = 1 for TV Encoder support

8.2 Supported Functionality

The display driver provides the following software and hardware support:

1. Variety of display types and resolutions (see Section 8.3.2, “Display Configurations.”)
2. Dual simultaneous output for two display devices (see Section 8.4.3.2, “Dual Display Support.”)
3. RGB565 and RGB8888 frame buffer pixel format
4. DirectDraw Hardware Abstraction Layer (DDHAL)
5. Up to five overlay surfaces
6. One overlay surface on each display when two displays are on (two active overlay surfaces total)
7. Video overlays containing image data in any of the following FOURCC pixel formats:
   - RGB565
   - UYVY
   - YV12
   - NV12
8. Hardware-accelerated color space conversion in video overlays
9. Hardware-accelerated image resizing in video overlays, resizing ratios ranging from 1:8 to 1000:1
10. Overlay surface color keying
11. Alpha blending with an overlay surface, through use of a global alpha value
12. Alpha blending with an overlay surface containing per-pixel alpha data (only ARGB8888 format)
13. Cropping of an overlay surface
14. Screen rotation of 0°, 90°, 180°, or 270°
15. Gamma correction support for dumb display device (Epson 2.8”, TV)
16. De-interlacing of a video overlay

The following limitations apply to the display driver overlay support:

1. The dimensions of the overlay surface may not exceed 1280×720
2. The width of the overlay surface must conform to an 8-pixel alignment restriction
3. The minimum width (or height if screen is rotated) of an overlay surface is 8 pixels
4. The minimum height (or width if screen is rotated) of an overlay surface is 8 pixels
5. Overlays are not supported when using a rotated screen with a resolution larger than 1024×768
6. When using the cropping feature, the x coordinate position must conform to the 8-pixel alignment restriction
7. When using the cropping feature with a surface using the YV12 pixel format, the x coordinate position must conform to 16-pixel alignment restriction and the y coordinate position must conform to 4-pixel alignment restriction
8. For a display using interlaced output (for example NTSC/PAL TV), the target overlay surface must have an even surface height
9. The area of overlay surface must be divisible by 32 for YV12 format
10. The Giantplus display panel exhibits tearing artifacts when playing dynamic video streams, due to a hardware limitation
11. When the display driver is configured for TV output, video de-interlacing is not supported

While the display driver is in 720p or 1080i TV output mode, the following supported features become unavailable due to the limited bandwidth and increased system loading associated with these modes:

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

8.3 Hardware Operation

For operation and programming information, see the chapter on the IPUv3 in the i.*MX51 Applications Processor Reference Manual*.

8.3.1 IPUv3 Overview

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

- Control Module (CM)—Provides control and synchronization for the entire IPUv3
- Image DMA Controller (IDMAC)—Transfers data to and from system memory
- Display Processor (DP)—Performs the processing required for data sent to display, including color space conversion and image combining
- Image Converter (IC)—Performs resizing, color conversion, combining with graphics, and horizontal inversion
- Image Rotator (IRT)—Performs rotation (90° or 180°) and inversion (vertical or horizontal)

- Video De-Interlacer (VDI)—Performs de-interlacing of interlaced video content
- Display Interface 0 and 1 (DI0/DI1)—Provides interface to displays, display controllers, and related devices
- Display Controller (DC)—Controls the display ports
- Display Multi-FIFO Controller (DMFC)—Controls FIFOs for IDMAC channels related to the display system

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

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

8.3.2 Display Configurations

The IPUv3 features two display ports each capable of generating output for one display. The platform catalog allows for the selection of only one display type for each display port—Display Port 0 and Display Port 1. Choosing a configuration that includes a display for both Display Port 0 and Display Port 1 allows the use of dual display mode.When a display is selected for both display ports, the display device on Display Port 0 is the default display device and is the only display that will be active when the system boots up (the display device on Display Port 1 will be turned off by default). See Section 8.4.2.1.2, “Changing To Dual Display Mode,” and Section 8.4.3.2, “Dual Display Support,” for details on configuring and changing to dual display mode. The catalog, and thus the OS image, may also be configured to select a display from only one of the display ports. Choose this configuration to switch between different display types or supporting multiple simultaneous displays.

8.3.2.1 i.MX51 EVK

The following displays and resolutions may be selected for the i. MX51 EVK:

- Display Port 0
  - DVI digital output—800×600, 1024×768, 1280×1024, 1280x720
  - Mitsubishi 8.4” XGA LVDS display (AA084XA03)—1024×768
- Display Port 1
  - Chunghwa 7” WVGA Display (CLAA070VC01)—800×480
  - VGA analog output—800×600, 1024×768, 1280×1024
  - TV Output support for NTSC and PAL standard televisions and 720p and 1080i HDTVs—720×480, 720×576, 1280×720, 1920×1080

8.3.3 Conflicts with Other Peripherals and Catalog Items

8.3.3.1 Conflicts with SoC Peripherals

No conflicts.

8.3.3.2 i.MX51 EVK Peripheral Conflicts

No conflicts.

8.4 Software Operation

8.4.1 Software Architecture

8.4.1.1 Software Driver Components

Figure 8-1 shows the relationship between software components in the display driver architecture.

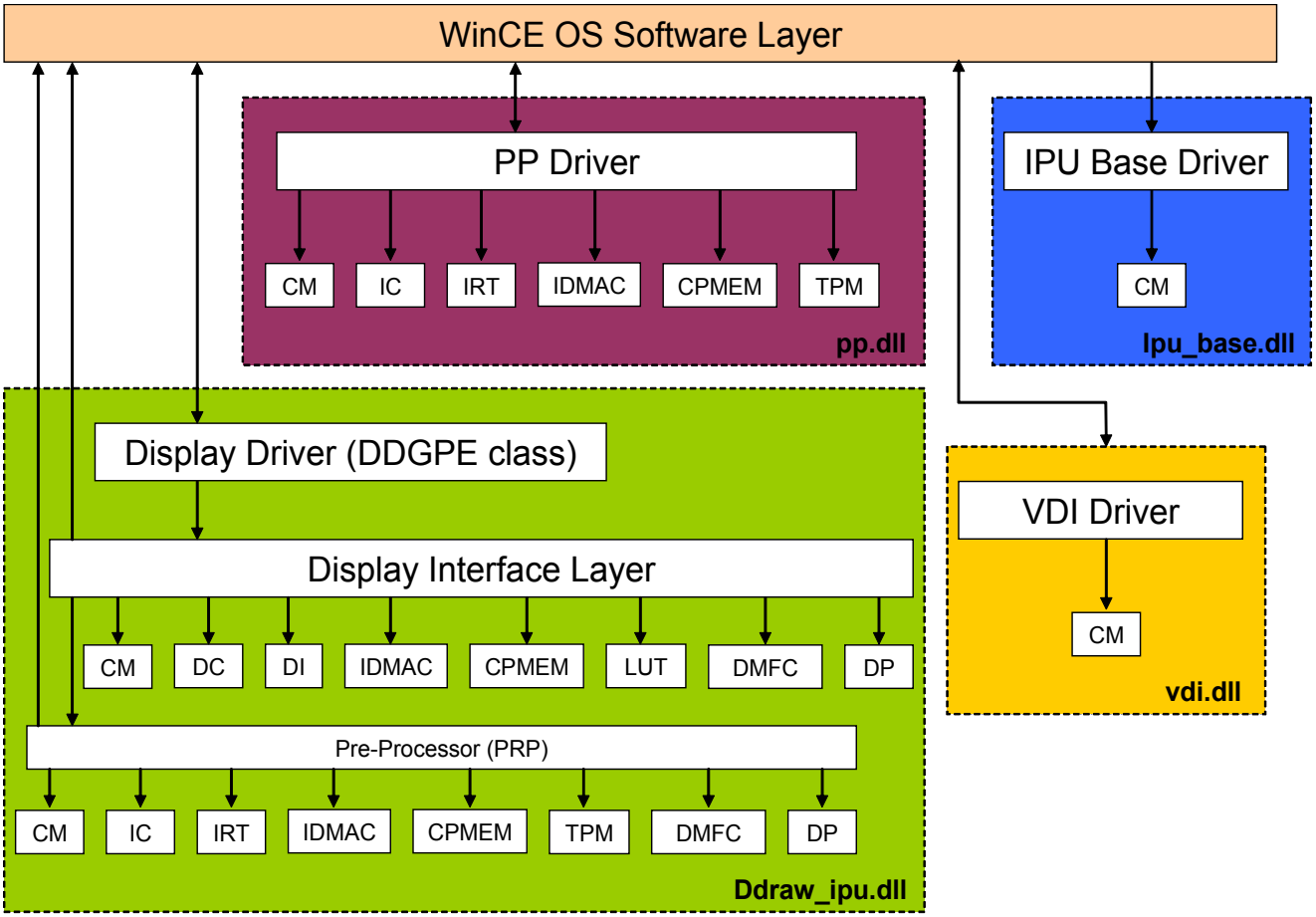


Figure 8-1. Software Architecture

Figure 8-1 shows the main elements of the display driver architecture:

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

8.4.1.1.1 Display Driver

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

8.4.1.1.2 Display Interface Layer

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

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

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

8.4.1.1.3 Pre-Processor Driver

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

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

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

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

8.4.1.1.4 Post-Processor Driver

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

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

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

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

8.4.1.1.5 Video De-Interlacer Driver

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

8.4.1.1.6 IPUv3 Base Driver

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

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

8.4.1.1.7 Low Level APIs

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

8.4.1.2 Video Memory Requirements

Memory must be reserved for the following types of surfaces:

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

Figure 8-2 shows the amount of memory required for each of the surfaces based on assumptions about how many surfaces are needed and the maximum resolutions for LCD and video content that are used in a hypothetical embedded system. For the application surfaces, an estimate has been made based on the size and number of surfaces needed to decode worst-case video content, and some additional buffering has been added to that to ensure space for additional surfaces. It is also important to note that the number of video processing surfaces multiplies with the number of simultaneous overlays that are used. Therefore, when developing a system that uses three simultaneous overlay surfaces, the number of video processing buffers increases proportionally.

Table 8-2. Surface Memory Requirements

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

8.4.2 Communicating with the Display

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

8.4.2.1 Using the Graphics Device Interface

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

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

8.4.2.1.1 Changing the Display Mode

The GDI function ChangeDisplaySettingsEx is used to change the display mode. For information and syntax on this function, see the Windows CE 6.0 documentation:

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

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

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

NOTE

There may be multiple display modes supported by the display driver that support the same resolution. To distinguish between these modes, the calling application should use the display frequency (no two display modes have the same resolution and frequency). The display frequency is set using the SET_DISPLAY_FREQUENCY DrvEscape code (see Section 8.4.2.3.1, “Setting the Display Frequency,”), and must be set before calling ChangeDisplaySettingsEx to change the mode.

8.4.2.1.2 Changing To Dual Display Mode

Display mode transitions may also trigger the enabling of dual display mode. In order for the display driver to allow a transition to dual display mode, the display driver must be configured and built with dual display supported (see Section 8.4.3.2, “Dual Display Support,”). Once a device with dual display support transitions from a display mode associated with Display Port 0 to a display mode associated with Display Port 1, dual display mode becomes active. At this point, the secondary primary surface is shown on the secondary display (the LCD transitioned from), and may be accessed through the steps described in Section 8.6.3, “Dual Display API.”

8.4.2.2 Using DirectDraw

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

Windows Embedded CE Features > Graphics > DirectDraw

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

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

The IPUv3 contains multiple image processing hardware blocks, which are used within the display driver to accelerate the following operations:

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

8.4.2.3 Using Display Driver Escape Codes

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

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

8.4.2.3.1 Setting the Display Frequency

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

- DISPLAY_SET_OUTPUT_FREQUENCY
- DISPLAY_GET_OUTPUT_FREQUENCY

The display frequency must be set in order to disambiguate between display modes that use the same resolution (for example 720p50 and 720p60). The display frequency should be set before calling ChangeDisplaySettingsEx to set the display mode. See Section 8.6.2.1, "DISPLAY_SET_DISPLAY_FREQUENCY Escape Code," for information about how to use these APIs.

8.4.2.3.2 Video De-Interlacing

Since there is no other way to pass information about whether an overlay surface is interlaced through the DirectDraw API, video de-interlacing is enabled through the DISPLAY_IS_VIDEO_INTERLACED driver escape code. Video de-interlacing is primarily used when decoding and playing back interlaced video content, so the video playback application must use the driver escape code to request that the display driver enable interlaced video mode. Refer to Section 8.6.2.1, "DISPLAY_SET_DISPLAY_FREQUENCY Escape Code," for information about how to use the API to enable video de-interlacing.

NOTE

Video de-interlacing is currently supported only for LCD mode. When TV mode is active, video de-interlacing should be disabled.

8.4.2.4 Using The Display Driver Control Panel Application

A control panel application provides access to additional display driver functionality. Look for the icon shown in Figure 8-2 in Windows CE control panel.

Figure 8-2. Display Driver Icon

The control panel application supports the following display driver features:

- Rotation between 0°, 90°, 180°, and 270°
- Gamma correction configuration for a synchronous display device, The gamma value may be set between 0.5 and 3.5. The default gamma value is 1.0.
- Display mode configuration with a drop-down box listing all of the display modes supported by the display driver. Each display mode is listed as a combination of the mode width, height, and frequency. For example, 480×640@60Hz represents the Epson panel LCD mode, and 720×480@50Hz represents NTSC TV mode. The resolution of the current mode is displayed in the box.

NOTE

This may includes display modes for panels that are not currently connected to the device, and which therefore do not work correctly.

The GUI of the display driver control panel application is shown in Figure 8-3.

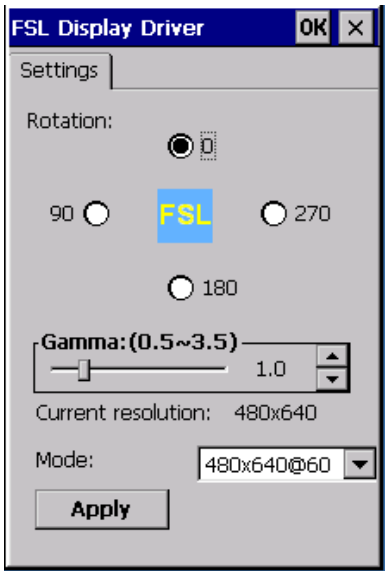


Figure 8-3. Display Driver GUI

NOTE

Because Windows CE 6.0 only identifies display modes from the width, height, and frequency, the least significant digit of the frequency is varied to distinguish otherwise identical modes. For example, both the VGA and DVI display types support a 1024×768@60Hz mode, so the DVI mode is represented as 1024×768@60Hz and the VGA mode is represented as 1024×768@61Hz.

8.4.3 Configuring the Display

The primary means for configuring the display is through the selection of a display panel type in the Platform Builder catalog. The selection of a panel in the catalog causes a BSP environment variable to be selected, which ultimately leads to the inclusion in the OS image of a **PanelType** registry key. The **PanelType** registry key, which is described in Section 8.4.3.5, "Display Registry Settings," specifies the display panel that is being used to the display driver. The PanelType provides the display driver an index into an array containing all of the main display configuration information for the panel—panel resolution, timings, pixel mappings, and additional information.

8.4.3.1 Rotation Support

The DirectDraw display driver may be configured to allow screen rotation through a parameter in the bspdisplay.h file. If the BSP_DIRECTDRAW_SUPPORT_ROTATION parameter is set to TRUE, the DirectDraw display driver supports rotation. If it is set to FALSE, it does not.

NOTE

The rotation feature is disabled when the panel resolution is larger than 1024×768.

8.4.3.2 Dual Display Support

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

NOTE

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

8.4.3.3 DispPerf Support

The DispPerf utility is provided by Microsoft to facilitate display driver performance profiling. When enabled in the display driver, the DispPerf utility can be executed and will generate a table providing data for all of the raster operations (ROP) that have been performed. The number of executions, the total execution time and the average execution time is listed for each ROP.

In order to enable DispPerf support in the display driver, the BSP must be built with the the environment variable DO_DISPPERF equal to 1. Since several public-side display driver libraries must be rebuilt to include DispPerf support, a "Clean Sysgen" is required to properly rebuild the BSP once the DO_DISPPERF environment variable has been set.

Once the display driver has been built with DispPerf support enabled, the DispPerf utility must be executed in order to generate the display performance output data. The DispPerf executable can be generated by building the source code found in `\WINCE600\SUPPORT\APP\DISPPERF`.

For more information about the DispPerf utility, including the command line options and interpreting the output data, please refer to the following section in the WinCE 6.0 Help documentation:

Developing a Device Driver > **Windows Embedded CE Drivers** > **Display Drivers** > **Display Driver Development Concepts** > **Display Driver Performance** > **Display Driver Performance Profiling**

8.4.3.4 Display Driver Blit Acceleration

Two on-chip Graphics Processing Unit (GPU) cores, GPU2D and GPU3D, may be accessed through the display driver to accelerate a subset of the GDI graphical blit operations. The subsequent sections provide details on the features offered by these two GPU cores, and how to configure the BSP to enable acceleration through these GPU cores.

8.4.3.4.1 GPU2D Graphics Acceleration

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

1. Enable the GPU base by setting the BSP_GPU_BASE environment variable. This may be achieved by selecting at least one GPU catalog item from the Third Party Catalog.

2. Enable the GPU2D component by setting the platform environment variable BSP_DISPLAY_Z160=1. This may be achieved by navigating to the project properties, and adding the environment variable in the Configuration Properties->Environment window.

8.4.3.4.1.3 Supported Acceleration Features

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

   The Y,U,V data range is:

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

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

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

8.4.3.4.1.4 Hardware Restrictions

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

8.4.3.4.2 GPU3D Graphics Acceleration

GPU3D core graphics acceleration may be enabled through the following steps:

1. Enable the GPU base by setting the BSP_GPU_BASE environment variable. This may be achieved by selecting at least one GPU catalog item from the Third Party Catalog..
2. Enable the GPU3D component by setting the platform environment variable BSP_DISPLAY_Z430=1. This may be achieved by navigating to the project properties, and adding the environment variable in the Configuration Properties->Environment window.

8.4.3.4.2.5 Supported Acceleration Features

1. Solid color fills.
2. BitBlt() - Simple operations not requiring rotation or resizing.
3. StretchBlt() - Support for COLORONCOLOR and BILINEAR stretch modes. For a DDraw blt, the default stretch mode is BILINEAR.
4. PatBlt() - Pattern copy blits are accelerated.
5. The following accelerated ROP operations: BLACKNESS, PATCOPY, SRCCOPY, WHITENESS.
6. 16BPP and 32BPP are supported.

8.4.3.5 Display Registry Settings

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

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

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

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

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

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

- Bit 0 - Enable/Disable solid color fill acceleration
- Bit 1 - Enable/Disable pattern fill acceleration
- Bit 2 - Enable/Disable simple bitblt (without rotation, stretchblt) acceleration

- Bit 3 - Enable/Disable line draw acceleration
- Bit 4 - Enable/Disable maskblt acceleration
- Bit 5 - Enable/Disable stretchblt acceleration
- Bit 8 - Enable/Disable acceleration for rotated screen cases

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

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

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

8.4.4 Power Management

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

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

The display driver also controls the operation of TVE module, calling to enable or disable the TVE and its clocks. To facilitate management of these modules, the display driver implements the power management I/O Control (IOCTL) codes, such as IOCTL_POWER_CAPABILITIES, IOCTL_POWER_QUERY, IOCTL_POWER_GET and IOCTL_POWER_SET.

8.4.4.1 PowerUp

This function is not implemented for the display driver.

8.4.4.2 PowerDown

This function is not implemented for the display driver.

8.4.4.3 IOCTL_POWER_SET

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

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

- D4—The DI and DC submodules of the IPUv3 are disabled. The display panel is disabled. If TV output mode was enabled, disable the TVE and its clocks.

8.5 Unit Test

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

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

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

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.

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

8.5.1 Unit Test Hardware

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

8.5.2 Unit Test Software

8.5.2.1 GDI Tests

Table 8-6 lists the software required to run the GDI tests.

Table 8-6. 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.

8.5.2.2 DirectDraw Tests

Table 8-7 lists the software required to run the DirectDraw tests.

Table 8-7. Direct Draw Software Requirements

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

8.5.2.3 Windows Media Player Tests

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

Table 8-8. Windows Media Player Software Requirements

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

8.5.2.4 Multiple Overlay Custom Test

A custom test is provided to test the display driver support for multiple overlays. Table 8-9 lists the software required to run the MultipleOverlay test.

Table 8-9. Multiple Overlay Software Requirement

Requirement	Description
MultipleOverlay.exe	Multiple overlay sample test application

8.5.2.5 Video De-Interlacing Custom CETK Test

A custom test is provided to test the de-interlacing functionality of the display driver. Table 8-10 lists the software required to run the VDI test.

Table 8-10. Video De-Interlacing Software Requirement

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
Tooltalk.dll	Library required by Tux.exe and Kato.dll. Handles the transport between the target device and the development workstation
Vdi_test.dll	VDI test .dll file
stefan_interlaced_320x240_30frames.yv12	Test input file containing Interlaced video input

8.5.3 Building the Unit Tests

8.5.3.1 GDI/DirectDraw Tests

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

Windows Embedded CE Test Kit > **Running the CETK**

8.5.3.2 Windows Media Player Tests

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

8.5.3.3 Multiple Overlay Custom Test

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

1. Open the BSP sample solution in Microsoft Visual Studio
2. Click **Build OS** > **Open Release Directory** to open the command prompt
3. Navigate to the test directory: `\WINCE600\SUPPORT\APP\MultipleOverlay`
4. Build the application with the command **build -c**
5. The binary `MultipleOverlay.exe` is automatically copied into the release directory

8.5.3.4 Video De-Interlacing Custom CETK Test

To build the VDI test, build an OS image for the desired configuration using the following steps:

1. Within Platform Builder, choose **Build OS** > **Open Release Directory**.
   A DOS prompt is displayed.
2. Navigate to the VDI test directory: `\WINCE600\SUPPORT\TESTS\VDI`
3. Enter **set WINCEREL=1** on the command prompt and press return.
   This copies the DLL to the flat release directory.
4. Execute the **build -c** to build the VDI test.

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

8.5.4 Running the Unit Tests

8.5.4.1 Running the GDI Tests

The command line for running the GDI tests is:

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

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

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

8.5.4.2 Running the DirectDraw Tests

The command line for running the DirectDraw tests is:

```
tux –o –d ddrawtk
```

NOTE

The display driver fails or aborts in the following DirectDraw CETK testcases: 400, 410, 420, 430, 500, 502, 504, 506, 508, 510, 512, 514, 516, 518, and 520. The failure occurs because flag DDSCAPS_VIDEOPORT is used for secondary panel support. DDSCAPS_VIDEOPORT is enabled; however, a real videoport feature is not implemented. This causes all CETK test cases involving videoport to fail.

8.5.4.3 Running the Windows Media Player tests

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

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

For example, `ceplayer motocross_208x160_30fps.wmv`

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

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

8.5.4.4 Running the Multiple Overlay Custom Test

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

```
s MultipleOverlay.exe
```

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

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

8.5.4.5 Running the Video De-Interlacing CETK Custom Test

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

The command line for running the VDI test is

```
tux –o –d vdi_test
```

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

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

8.6 Display Driver API Reference

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

8.6.1 GDI and DirectDraw APIs

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

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

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

Windows CE Features > **Graphics and Multimedia Technologies** > **Graphics** > **DirectDraw**

8.6.2 Driver Escape Code Extensions

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

8.6.2.1 DISPLAY_SET_DISPLAY_FREQUENCY Escape Code

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

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

Parameters

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

8.6.2.2 DISPLAY_GET_DISPLAY_FREQUENCY Escape Code

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

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

Parameters

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

8.6.2.3 DISPLAY_IS_VIDEO_INTERLACED Escape Code

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

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

Parameters

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

8.6.2.4 DISPLAY_SETCRRECT Escape Code

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

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

Parameters

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

8.6.3 Dual Display API

Microsoft does not provide native support for dual independent displays in Windows CE 6.0. The Windows CE 6.0 documentation describes support for Multiple Screens and for a Secondary Display Driver, but both of these features are critically limited and insufficient to provide dual independent display support. As a result, a custom extension to the DirectDraw APIs is required to allow an application to access a secondary primary surface for the secondary display.

8.6.3.1 Dual Display Interface

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

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

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

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

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

// Create back buffer with size equal to primary to blit from
memset(&ddsd, 0, sizeof(ddsd));
ddsd.dwSize = sizeof(ddsd);
ddsd.ddsCaps.dwCaps = DDSCAPS_SYSTEMMEMORY;
ddsd.dwFlags = DDSD_CAPS | DDSD_HEIGHT | DDSD_WIDTH | DDSD_PIXELFORMAT;
ddsd.dwWidth = 480;
ddsd.dwHeight = 640;
ddsd.ddpfPixelFormat = ddpfOverlayFormats[0];
hRet = g_pDD->CreateSurface(&ddsd, &g_pDDSBack[0], NULL);

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

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

8.6.3.2 Dual Display API Limitations

The dual display API limitations are as follows:

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

Chapter 9
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.

9.1 DVFC Driver Summary

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

Table 9-1. DVFC Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVKiMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\DVFC
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\DVFC
Driver DLL	dvfc_mc13892.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > DVFC driver support using the MC13892
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOPMIC = BSP_DVFC_MC13892 = 1

9.2 Supported Functionality

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

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

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

9.2.1 i.MX51 Supported Functionality

1. Supports DVFS for CPU (GP) and peripheral (LP) power domains
2. Exposes separate Power Manager stream interfaces for CPU and peripheral domains to provide individual control of setpoint for each domains
3. Supports reactive CPU load tracking to control setpoint based on system performance requirements. Current release uses software load tracking algorithm
4. Provides voltage control using MC13892 PMIC

9.3 Hardware Operation

This section describes about the DVFC hardware operation.

9.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

9.3.2 i.MX51 EVK Configuration

The DVFC driver is dependent upon the MC13892 PMIC interface for dynamic voltage control through the eCSPI1 port. The MC13892 SDK import library is used by the DVFC driver to access the PMIC interface.

9.4 Software Operation

This section describes about the registry settings.

9.4.1 i.MX51 Registry Settings

The following registry keys are required to properly load the i.MX51 DVFC module.

```
IF BSP_DVFC_MC13892
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\DVFC1]
   "Prefix" = "DVF"
   "Index" = dword:1
   "Dll"="dvfc_mc13892.dll"
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"          ; PMCLASS_GENERIC_DEVICE

[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\DVFC2]
   "Prefix" = "DVF"
   "Index" = dword:2
   "Dll"="dvfc_mc13892.dll"
```

```
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"          ; PMCLASS_GENERIC_DEVICE
ENDIF            ; BSP_DVFC_MC13783
```

9.4.2 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.

9.4.3 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.

9.4.3.1 i.MX51 Voltage/Frequency Setpoints

The i.MX51 DVFC driver supports mutually exclusive voltage and frequency setpoints for the CPU and peripheral power domains. Table 9-2 and Table 9-3 provide the voltage/frequency characteristics for these setpoints.

Table 9-2. i.MX51 DVFC Setpoints for CPU Domain

Setpoint Name	CPU Frequency (MHz)	Core Voltage
DDK_DVFC_SETPOINT_HIGH	800	1.1 V
DDK_DVFC_SETPOINT_MEDIUM	160 (pll1_sw_clk sourced from PLL1) 166 (pll1_sw_clk migration to PLL2)[1]	0.85 V
DDK_DVFC_SETPOINT_LOW	Unused	Unused

[1] Refer to Section 9.4.3.2, “i.MX51 PLL Migration,” for more details

Table 9-3. i.MX51 DVFC Setpoints for Peripheral Domain

Setpoint Name	AXI/AHB/DDR Frequency (MHz)	Core Voltage
DDK_DVFC_SETPOINT_HIGH	166/133/200	1.225 V
DDK_DVFC_SETPOINT_MEDIUM	166/83/133	1.225 V
DDK_DVFC_SETPOINT_LOW	24/24/133	1.225 V

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

9.4.3.2 i.MX51 PLL Migration

By default, pll1_sw_clk (CPU and high-speed DDR clock source) is sourced from PLL1. The i.MX51 CCM allows pll1_sw_clk to be sourced from an alternate PLL. Under certain conditions, it is possible to migrate pll1_sw_clk from PLL1 to PLL2. Such migration allows PLL1 to be shut down for power savings. The feature can be enabled and disabled using the BSP_DVFS_PLL1_SW_CLK_MIGRATION macro in `\PLATFORM\<Target Platform>\SRC\INC\dvfs.h`. The BSP currently restricts pll1_sw_clk migration unless the following conditions are met:

- ARM_CLK_ROOT frequency provided from divided PLL1 can be replaced with nearly equivalent divided PLL2
- DDR_CLK_ROOT frequency provided from divided PLL1 can be replaced with nearly equivalent divided PLL2, or DDR_CLK_ROOT is not sourced from PLL1
- main_bus_clk is being sourced from periph_apm_clk (peripheral domain is at LOW setpoint)
- PLL1 is not configured as source for other peripheral clock roots

9.4.3.3 i.MX51 Setpoint Mapping

The peripherals may not be able to operate properly in all of the supported setpoints due to minimum frequency/voltage requirements. Therefore, drivers that support these peripherals need a method of communicating setpoint requirements. The setpoint requirements for drivers are expressed in terms of the the following parameters:

- Internal bus (AXI/AHB) frequency requirement
- External DDR bus frequency requirement
- Peripheral domain (LP) voltage requirement

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

WARNING

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

The DVFC driver advertises support for IOCTL_POWER requests from the Power Manager. A IOCTL_POWER_SET request is mapped to a setpoint by the DVFC driver. This mapping allows applications to use the Power Manager APIs to request changes in the DVFC setpoint. The mapping of device power states (D0–D4) to DVFC setpoints is located in **DvfcMapDevPwrStateToSetpoint** (found

in `\PLATFORM \<Target Platform>\SRC\DRIVERS\DVFC\COMMON\dvfc.c`). The DVFC driver exposes two separate stream interfaces to allow individual control of the CPU and peripheral power domain setpoints. Stream DVF1 is mapped to the CPU domain and DVF2 is mapped to the peripheral domain. To change the setpoint mapping for a specific device power state (D0–D4), modify the code in **DvfcMapDevPwrStateToSetpoint**.

9.4.4 DDK Interface

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

- DDKClockSetpointRequest, Section 6.5.1.2.6, "DDKClockSetpointRequest."
- DDKClockSetpointRelease, Section 6.5.1.2.7, "DDKClockSetpointRelease."

9.4.5 Power Management

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

9.4.5.1 PowerUp

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

9.4.5.2 PowerDown

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

9.4.5.3 IOCTL_POWER_CAPABILITIES

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

9.4.5.4 IOCTL_POWER_SET

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

9.4.5.5 IOCTL_POWER_GET

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

9.5 Unit Test

This section explains about the unit testing.

9.5.1 i.MX51 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 and peripheral DVFS domains. Follow these steps to run this unit test:

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

NOTE

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

2. Build the DVFC stress test located in `\SUPPORT_PDK1_7\TEST\APP\PWRMGMT`. The resulting application pwrmgmt.exe is generated in the flat release directory.
3. Boot the OS image and launch application code such as media player that can perform continuous playback. WMA audio playback is a good use case since audio playback can be performed across all supported setpoints.
4. Launch the stress test application. From the CE shell, the stress test can be launched with the following command line:

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

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

Chapter 10
Enhanced Configurable Serial Peripheral Interface (eCSPI) Driver

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

10.1 eCSPI Driver Summary

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

Table 10-1. eCSPI Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\ECSPI
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\ECSPI
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\ECSPI
SDK Library	ecspisdk.lib
Driver DLL	ecspi.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>:ARMV4I > Device Drivers > CSPI Bus
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_ECSPI1=1

10.2 Supported Functionality

The eCSPI driver supports the following features:

1. eCSPI master mode of operation
2. eCSPI configurable bus feature
3. eCSPI multiple channel method
4. DMA exchange mode
5. Configurable access method of interrupt mode and DMA mode
6. Buffering exchange for asychronous SPI access
7. Stream interface
8. Two power management modes, full on and full off

10.2.1 Conflicts with Other Peripherals and Catalog Items

10.2.1.1 Conflicts with SoC Peripherals

10.2.1.1.1 i.MX51 Peripheral Conflicts

The i.MX51 contains two eCSPI controllers. The first eCSPI conflicts with IIC1.

10.2.1.2 Conflicts with Board Peripherals

No conflicts.

10.3 Software Operation

10.3.1 Registry Settings

10.3.1.1 i.MX51 Registry Settings

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

```
;------------------------------------------------------------------------------
; ECSPI Bus Driver
;
IF BSP_ECSPI1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\ECSPI1]
   "Prefix"="SPI"
   "Dll"="ecspi.dll"
   "Index"=dword:1
   "Order"=dword:1
   "IClass"=multi_sz:"{A32942B7-920C-486b-B0E6-92A702A99B35}"
ENDIF    ; BSP_ECSPI1
;------------------------------------------------------------------------------
```

10.3.2 Communicating with the eCSPI

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

10.3.3 Creating a Handle to the eCSPI

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

To open a handle to the eCSPI:

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

The following code example shows how to open a eCSPI port:

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

10.3.4 Data Transfer Operations

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

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

To perform an eCSPI transfer:

1. Create a CSPI_XCH_PKT object and initialize the fields of the packet as follows:
   a) Initialize a CSPI_BUSCONFIG datum to specify the bus parameters CHANNEL SELECT, DATA RATE, BURST LENGTH, SSPOL, POL, DRCTL, and specify the method parameters for using or not using the DMA. The BURST LENGTH unit is bit.
   b) Set the *pTxBuf* field to the user buffer with the transmit data.
   c) Set the *pRxBuf* field to the user buffer which receives data. If there is no receive data, set the field to NULL.
   d) Set the *xchCnt* field. The *xchCnt* unit is 32-bit. *xchCnt* must equal BurstLength/32 or BurstLength/32 +1 (if BurstLength is not multiple of 32-bits).
   e) Specify the *xchEvent* parameter and the *xchEventlength* including the tail zero character. Otherwise, set *xchEvent* to NULL and *xchEventlength* to 0. When using *xchEvent*, the XCH data is queued inside the driver.
2. Set the *hDevice* parameter to the previously acquired eCSPI port handle.

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

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

```
CSPI_BUSCONFIG_T buscnfg =
{       0, //use channel 0
        16000000, //XCH speed 16M
        32*64,   //Burstlength 64 DWORDS
        FALSE, // SSPOL: Active LOW
        FALSE, // POL: Active high polarity
        0,     // DRCTL: Don't care SPI_RDY
        FALSE}; //Don't use DMA

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

CSPI_XCH_PKT_T xchPkt =
{       &buscnfg,
        TxData,
        RxData,
        64,                                          // DWORD, Equal Burstlength/32
        NULL,
        0};
// optional asynchronous event, recommended
hEvent = CreateEvent(0, FALSE, FALSE, L"RX_EVENT");
xchpkt.xchEvent = L"RX_EVENT";
xchpkt.xchEventLength = sizeof(L"RX_EVENT");

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

10.3.5 Closing the Handle to the eCSPI

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

10.3.6 Power Management

The primary method for limiting power consumption in the eCSPI module is to gate off the input clock to the module when the input eCSPI clock is not needed. This is accomplished through the **DDKClockSetGatingMode** function call. In the all Windows CE 6.0 BSP use cases, the eCSPI controller acts as a master device. As a result, the eCSPI clock can be turned off, whenever the module is not processing eCSPI packets.

10.3.6.1 PowerUp

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

10.3.6.2 PowerDown

This function is not implemented for the eCSPI driver.

10.3.6.3 IOCTL_POWER_SET

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

10.4 Unit Test

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

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

10.5 eCSPI Driver API Reference

10.5.1 eCSPI Driver IOCTLS

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

10.5.1.1 CSPI_IOCTL_EXCHANGE

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

Parameters

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

10.5.1.2 CSPI_IOCTL_ENABLE_LOOPBACK

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

10.5.1.3 CSPI_IOCTL_DISABLE_LOOPBACK

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

10.5.2 eCSPI Driver SDK Wrapper

10.5.2.1 CSPIOpenHandle

This function retrieves the eCSPI device handle.

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

Parameters

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

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

10.5.2.2 CSPICloseHandle

This function closes a handle of the eCSPI stream driver.

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

Parameters

hDev — eCSPI device handle retrieved from CreateFile()

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

10.5.2.3 CSPIEnableLoopback

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

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

Parameters

hDev — eCSPI device handle retrieved from CreateFile()

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

10.5.2.4 CSPIExchange

This function performs XCH operations.

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

Parameters

hDev — eCSPI device handle retrieved from CreateFile()

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

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

10.5.3 eCSPI Driver Structures

10.5.3.1 CSPI_BUSCONFIG_T

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

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

Table 10-2. CSPI_BUSCONFIG_T Structure Members

Member	Description
ChannelSelect	Select XCH channel, range 0–3
Freq	Data bandrate
BurstLength	Define bits used in a single XCH, range 1–32×64
SSPOL	SPI SS Polarity Select FALSE: active low TRUE: active high
SCLKPOL	SPI Clock Polarity Control FALSE: active high polarity (0 = Idle) TRUE: active low polarity (1 = Idle)
SCLKPHA	SPI Clock/Data Phase Control FALSE: phase 0 operation TRUE: phase 1 operation
DRCTL	DRCTL of eCSPI XCH operation 00: Do not care SPI_RDY 01: Burst triggered by failing edge of SPI_RDY 10: Burst triggered by low level of SPI_RDY 11: Reserved
usedma	True: use DMA mode

10.5.3.2 CSPI_XCH_PKT_T

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

```
// eCSPI exchange packet
typedef struct
{
    PCSPI_BUSCONFIG_T pBusCnfg;
    LPVOID pTxBuf;
    LPVOID pRxBuf;
    UINT32 xchCnt;
```

```
    LPWSTR xchEvent;
    UINT32 xchEventLength;
} CSPI_XCH_PKT_T, *PCSPI_XCH_PKT_T;
```

Table 10-3. CSPI_XCH_PKT_T Structure Members

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

Chapter 11
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.

11.1 eSDHC Driver Summary

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

Table 11-1. eSDHC Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\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_v2.lib, esdhcbase_<*Target SOC*>.lib, sdcardlib.lib, sdhclib.lib, sdbus.lib
Catalog Item	Third Party > BSP > Freescale i.MX51 EVK: ARMV4I > Device Drivers > SD Host Controller > Enhanced SD Host Controller 1 (ESDHC1) Support Third Party > BSP > Freescale i.MX51 EVK: ARMV4I > Device Drivers > SD Host Controllers > Enhanced SD Host Controller 2 (ESDHC2) Support
SYSGEN Dependency	SYSGEN_SD_MEMORY=1
BSP Environment Variables	BSP_NOSDHC= BSP_ESDHC1=1 BSP_ESDHC2=1 IMGSDBUS2

11.2 Supported Functionality

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

1. Enhanced Secure Digital Host Controllers
2. Designed and implemented as close as possible to Standard Host Controller Driver in CE 6.0 R2
3. Compliant with the SDBUS2 driver provided in CE 6.0 R2
4. Fast Path

5. DMA or PIO modes of data transfers based on value of eSDHC driver registry key, DisableDMA
6. SD, SD High Capacity (up to spec v2.1), MMC (up to spec v4.3), and SDIO cards (up to spec v2.0). High capacity MMC cards are not supported because SDBUS2 in CE 6.0 R2 does not support these cards.
7. One host supports only one card connected to it
8. DLL supports multiple instances of the eSDHC controller
9. Configuration of the block sizes from 1–4096 bytes in single and multi-block modes
10. Insertion and removal of card, even when system is suspended; when the system resumes, the card (if present) is remounted
11. Power states on(D0) and off (D4), D1–D3 states are treated as D4
12. Clocks are gated in D4 state, and ungated in D0 state, except for SDIO cards for which clocks are never gated. Clocks are never gated if BSP_CLK_GATING_BETWEEN_CMDS_SDHC macro is FALSE or undefined in the bsp_cfg.h file
13. Power supply (VGEN2) to SD socket is turned off when no card is present
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
17. MMC cards at 15.8 MHz and SD/SDIO cards at 23.75 MHz, the clocks closest to the limits in SDBUS2 in CE 6.0 R2 (20 MHz and 25 MHz for MMC and SD cards, respectively) using the 47.5 MHz eSDHC master clock
18. Reads at 7.0 Mbps and writes at 4.8 Mbps measured by running the CETK Storage Device Block Driver Performance Test on SanDisk Extreme III SDHC card
19. Reads at 1.75 Mbps and writes at 1.25 Mbps measured by running the CETK Storage Device Block Driver Performance Test on Transcend MMCplus card

11.3 Hardware Operation

Refer to the chapter on the eSDHC in the *i.MX51 Applications Processor Reference Manual* for detailed operation and programming information.

11.3.1 Conflicts with Other Peripherals and Catalog Options

11.3.1.1 Conflicts with SoC Peripherals

eSDHC1 CMD, CLK, and DATA[0–3] pads can be configured for their primary function in Alternate Mode 0. eSDHC2 DATA[0–3] can be configured as eSDHC1 DATA[4–7] in Alternate Mode 1. eSDHC2 CMD, CLK, and DATA[0–3] can be configured for their primary function in Alternate Mode 0. SD2_WP pad (GPIO1_7 in Alternate Mode 6) can also be configured as SPDIF_OUT signal in Alternate Mode 2.

eSDHC3 and eSDHC4 signals can be connected through various NANDF pads configured in Alternate Mode 2 or 5.

11.3.1.2 Conflicts with Other Board Peripherals

No conflicts.

11.4 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

11.4.1 Required Catalog Items

11.4.1.1 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.

11.4.2 eSDHC Registry Settings

11.4.2.1 i.MX51 SDHC Registry Settings

```
; @CESYSGEN IF CE_MODULES_SDBUS

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

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

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

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

; @CESYSGEN ENDIF CE_MODULES_SDBUS
```

11.4.3 DMA Support

11.4.3.1 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.

11.4.3.1.1 i.MX51 DMA Support

For the i.MX51, both DMA and non-DMA mode are supported by the driver. DMA mode is used by default, and user can change the *DisableDMA* value in registry file `esdhc_mx51.reg` to enable non-DMA mode. Internal DMA on eSDHC is used. Two internal DMA modes are supported by the eSDHC hardware: Simple DMA and Advanced DMA (ADMA). The driver supports only ADMA mode. TO1 supports ADMA1 protocol, while TO2 supports the improved ADMA2 protocol.

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

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

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

11.4.4 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.

11.4.4.1 i.MX51 Power Management

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

The power supply for the SD ports on the i.MX51-EVK is the VGEN2 output from the MC13892 PMIC. This supply is shared with other peripherals. The PMIC driver determines when it is safe to grant the eSDHC driver request to turn off the supply. The eSDHC driver requests to turn off this supply whenever there is no card plugged in or when the system is suspended. When a card is inserted or the system resumes from suspend, the eSDHC driver requests to turn on this supply. The SHC_PowerDown and SHC_PowerUp APIs calls down to the BspESDHCSetSlotVoltage function, which actually handles the communication with the PMIC driver.

11.5 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

11.5.1 Unit Test Hardware

Table 11-2 lists the required hardware to run the unit tests.

Table 11-2. 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)

11.5.2 Unit Test Software

Table 11-3 lists the required software to run the unit tests.

Table 11-3. 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

11.5.3 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
```

11.5.4 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

11.5.4.1 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.

11.5.4.2 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.

11.5.4.3 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.

11.5.4.4 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.

11.5.4.5 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.

11.5.5 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.

11.6 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 12
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.

12.1 Fast Ethernet Driver Summary

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

Table 12-1. FEC Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\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_NOETHER=

12.2 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

12.3 Hardware Operations

The Fast Ethernet Controller connects with the external transceivers using standard MII (Media Independent Interface) connection. All the registers in the external transceivers can be accessed by the MII

compatible management frames. The interrupt signal from the external transceiver is connected to the processor through the PBC (Peripheral Bus Controller). Refer to the Peripheral Bus Controller document for detailed operation and programming information. The attached transceiver for the Fast Ethernet Controller can detect the speed of the ethernet network automatically by the auto-negotiation process. The software accesses the status register of attached transceiver to determine the speed of the ethernet network (10 Mbps or 100 Mbps).

12.3.1 Conflicts with Other Peripherals and Catalog Items

12.3.1.1 Conflicts with SoC Peripherals

12.3.1.2 Conflicts with i.MX51 EVK Peripherals

No conflicts.

12.4 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.

12.4.1 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"
```

12.5 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.

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

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
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

12.5.3 Building the Unit Tests

12.5.3.1 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
```

12.5.3.2 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
```

12.5.3.3 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
```

12.5.3.4 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
```

12.5.3.5 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
```

12.5.4 Running the Unit Tests

12.5.4.1 Network Utilities Related Tests

12.5.4.1.1 Ping Tests

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

12.5.4.1.2 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.

12.5.4.1.3 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:
```

12.5.4.2 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).

12.5.4.3 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.

12.5.4.4 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.

12.5.4.5 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**.

12.6 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 13
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.

13.1 GPT Driver Summary

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

Table 13-1. GPT Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\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 > Timers > General-purpose Timer Support
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_GPT=1

13.2 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.

13.3 Hardware Operation

Refer to the chapter on GPT in the *i.MX51 Applications Processor Reference Manual* for detailed hardware operation and programming information.

13.3.1 Conflicts with Other Peripherals and Catalog Items

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

13.4 Software Operation

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

13.4.1 GPT Registry Settings

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

13.4.2 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 13.7, "GPT SDK API Reference." To use this API, it is necessary to include the `gptsdk.lib` library.

13.4.2.1 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:");
```

13.4.2.2 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);
```

13.4.2.3 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)
{
}
```

13.4.2.4 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.
```

13.4.3 DMA Support

The GPT driver does not use the DMA.

13.5 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()**.

13.5.1 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.

13.5.2 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.

13.5.3 IOCTL_POWER_SET

This function is not implemented for the GPT driver.

13.6 Unit Test

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

13.6.1 Unit Test Hardware

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

Table 13-2. Hardware Requirements

Requirement	Description
No additional hardware required	

13.6.2 Unit Test Software

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

Table 13-3. Software Requirements

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

13.6.3 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\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.

13.6.4 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 13-4 describes the test cases contained in the GPT tests.

Table 13-4. GPT Test Cases

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

13.7 GPT SDK API Reference

13.7.1 GPT SDK Functions

13.7.1.1 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()

13.7.1.2 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.

13.7.1.3 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

13.7.1.4 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

13.7.1.5 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

13.7.1.6 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

13.7.1.7 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

13.7.1.8 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

13.7.2 GPT Driver Structures

13.7.2.1 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

13.7.2.2 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 14
Graphics Processing Unit (GPU)

The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D/3D graphics applications. The GPU3D (3D graphics processing unit) is based on the AMD Z430 core, which is an embedded engine capable of DirectX9 Shader Model 3.0+ program execution and accelerates user level graphics APIs such as OpenGL ES 1.1 and 2.0, and Direct3D Mobile. The GPU2D (2D graphics processing unit) is based on the AMD Z160 core, which is an embedded 2D and vector graphics accelerator targeting the OpenVG 1.1 graphics API and feature set. The GPU driver is delivered only as binary code.

14.1 GPU Driver Summary

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

Table 14-1. GPU Driver Summary

Driver Attribute	Definition			
Target Platform	iMX51-EVK			
Target SOC				
SOC Common Path	N/A			
SOC Specific Path	N/A			
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\GPU			
Driver DLL	amdgsllddl.dll lib2dz160k.dll libGLES_CM.dll ibgslmemcfg.dll libOpenVG.dll	d3dm_ati.dll lib2dz430k.dll libGLESv1_CM.dll libgsluser.dll libos.dll	essc.dll lib2d-z430.dll libGLESv2.dll libgslWrapperk.dll libpanel.dll	lib2d-z160.dll libEGL.dll libgsl.dll libkos.dll librenderboy.dll
SDK Library	N/A			
Catalog Item	Third Party > BSP > Freescale <*Target Platform*> > Device Drivers > GPU > Graphics Processing Unit > OpenGL ES support Third Party > BSP > Freescale <*Target Platform*> > Device Drivers >GPU > Graphics Processing Unit > Direct3D Mobile support Third Party > BSP > Freescale <*Target Platform*> > Device Drivers > GPU > Graphics Processing Unit > OpenVG support			
SYSGEN Dependency	SYSGEN_D3DM for Direct3D Mobile support			
BSP Environment Variables	BSP_GPU_BASE=1 BSP_GPU_OPENGLES=1 BSP_GPU_D3DM=1 BSP_GPU_OPENVG=1			

14.2 Supported Functionality

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

1. EGL™ (interface between Khronos rendering APIs such as OpenGL ES or OpenVG and the underlying native platform window system) 1.3 API defined by Khronos Group
2. OpenGL® ES (royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems) 1.1 API defined by Khronos Group
3. OpenGL ES 2.0 API defined by Khronos Group
4. Direct3D® Mobile (API that provides support for 3D graphics applications on Windows Embedded CE-based platforms) API defined by Microsoft
5. OpenVG™ (royalty-free, cross-platform API that provides a low-level hardware acceleration interface for vector graphics libraries such as Flash and SVG) 1.1 API defined by Khronos Group
6. D0 (Full On) and D4 (Off) power states

14.3 Hardware Operation

Refer to the GPU chapter in the *i.MX51 Applications Processor Reference Manual* for detailed hardware operation and programming information.

14.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

14.4 Software Operation

14.4.1 Communicating with the GPU

The GPU driver is divided into two layers. The first layer is running in kernel mode, acting as the base driver for the whole stack and providing the essential hardware access, device management, memory management, command stream management, context management and power management. The second layer is running in user mode, implementing the stack logic and providing following APIs to the upper layer applications such as:

-
- EGL 1.3 API
- OpenVG 1.1 API
- OpenGL ES 1.1 and 2.0 API
- Direct3D Mobile API

14.4.2 GPU Driver Files

Listed below is a brief introduction to the GPU driver files. This list is not complete. The platform.bib file contains the complete list.

- Files that reside in kernel space:

amdgslldd.dll—base GPU driver and the standard stream interface driver, provides essential access to GPU hardware

  - libkos.dll—contains OS helper functions
  - libgsl.dll—contains common Graphics System Layer (GSL) logic
  - libgslmemcfg.dll—contains memory configuration helper functions
  - lib2dz430k.dll—contains Z430 c2d helper functions
- Files that reside in user space
  - libos.dll—contains OS helper functions
  - libgsluser.dll—contains common Graphics System Layer (GSL) logic
  - lib2d-z160.dll—contains Z160 c2d helper functions
  - libpanel.dll—contains GPU configuration helper functions so that some configurations could be customized during runtime, instead of hard-built images
  - libEGL.dll—contains EGL implementation
  - libOpenVG.dll—contains OpenVG 1.1 implementation
  - essc.dll—contains shader compiler logic
  - librenderboy.dll—contains the logic of rendering framework
  - lib2d-z430.dll—contains Z430 c2d helper functions
  - libGLES_CM.dll—contains OpenGL ES 1.1 implementation
  - libGLESv1_CM.dll—contains OpenGL ES 1.1 implementation, different wrapper
  - libGLESv2.dll—contains OpenGL ES 2.0 implementation
  - d3dm_ati.dll—contains Direct3D Mobile implementation

14.4.3 Power Management

The GPU driver implements the PowerUp and PowerDown APIs with support for the D0 (Full On) and D4 (Off) power states. These states are handled in the following manner:

- D0—GPU clocks are not enabled until the GPU driver is required to enable the clocks, for example, when an OpenGL ES application is launched. The GPU driver disabled the clocks when applications exit. Additionally, the graphics core has integrated power management design that supports gated clock branches used to turn off idle blocks within the core. This block-level clock gating is managed automatically in the core and GPU driver enables this capability when configure the core at the initialization time.
- D4—GPU clocks are disabled and power supplies are also disabled when possible.

14.4.4 GPU Registry Settings

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

[HKEY_LOCAL_MACHINE\Drivers\GPU]
;"MemSize"=dword:2000000  ; 32MB

; @CESYSGEN IF DIRECTX_MODULES_D3DM
;
IF BSP_GPU_D3DM
[HKEY_LOCAL_MACHINE\System\D3DM\Drivers]
"LocalHook"="d3dm_ati.dll"
ENDIF BSP_GPU_D3DM
;
; @CESYSGEN ENDIF DIRECTX_MODULES_D3DM

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

14.4.5 Graphics Device Interface (GDI) Acceleration

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

Currently reference codes are added into the display driver to accelerate following GDI operations using C2D Z430 or C2D Z160. An environment variable bsp_display_c2d is used to toggle on and off the following support:

- Solid color fills
- Pattern fills
- SRCCOPY blit operations

14.5 Float Pointing Acceleration using the ARM Vector Floating Point (VFP) Library

As this SOC includes a VFP module, graphics applications or drivers can use VFP to accelerate the mathematical algorithm. You can download the ARM VFP library release from the ARM website(http://www.arm.com/products/os/windowsce.html) and use the information in the release notes to enable the OEM floating point library support.

14.6 Unit Test

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

14.6.1 Unit Test Hardware

No special requirements.

14.6.2 Unit Test Software

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

14.6.2.1 GSLTest

This internal unit test application verifies the basic functionality of GSL driver layer. It is included in the release image and is located under `\Windows\gsl_test.exe`. Click to launch this test and read the log messages to verify the test completed without any error or warning messages.

NOTE

The log messages is outputted on the console connected to the debug serial port or output window of Visual studio if KITL connection is available.

14.6.2.2 EGLTest

This internal unit test application verifies the basic functionality of EGL driver layer. It is included in the release image and is located under `\Windows\egltest.exe`. Click to launch this test and read the log messages to verify the test completed without any error or warning messages.

14.6.2.3 Tiger Test

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

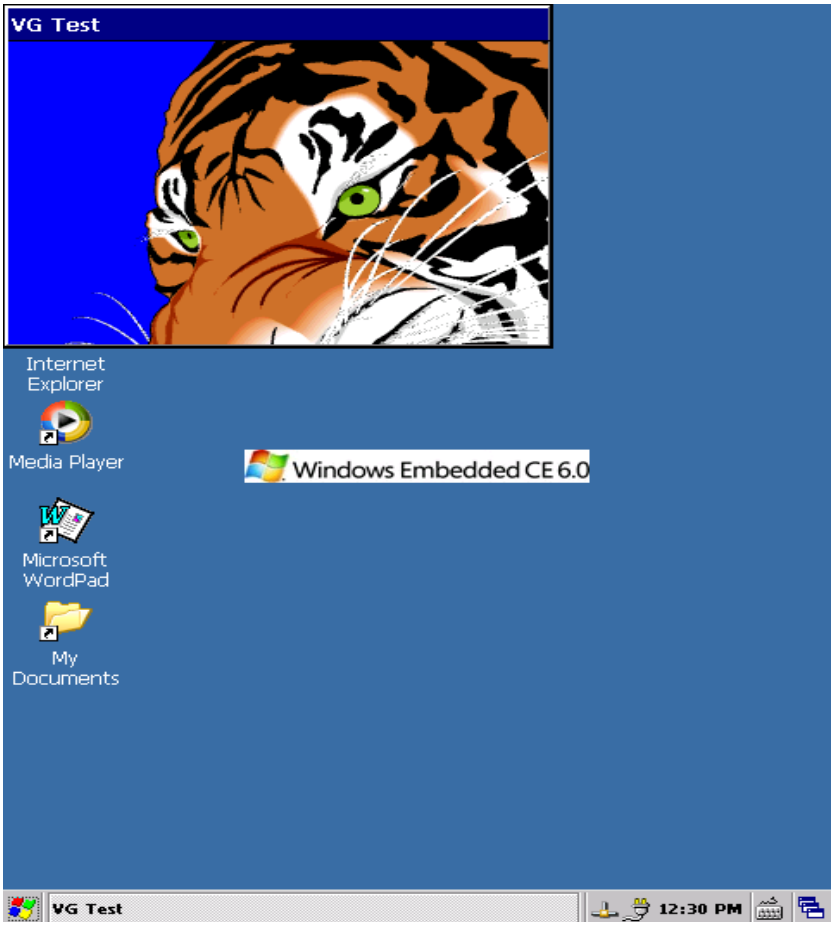


Figure 14-1. Tiger Test

14.6.2.4 OpenVG 1.1 Conformance Test

The OpenVG 1.1 conformance test is standard OpenVG conformance test designed by the Khronos Group. Visit the Khronos Group website at http://www.khronos.org/opengles/adopters/login/ for detailed information about how to download the source code, build the test binaries and run this tests.

14.6.2.5 Cube Test

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

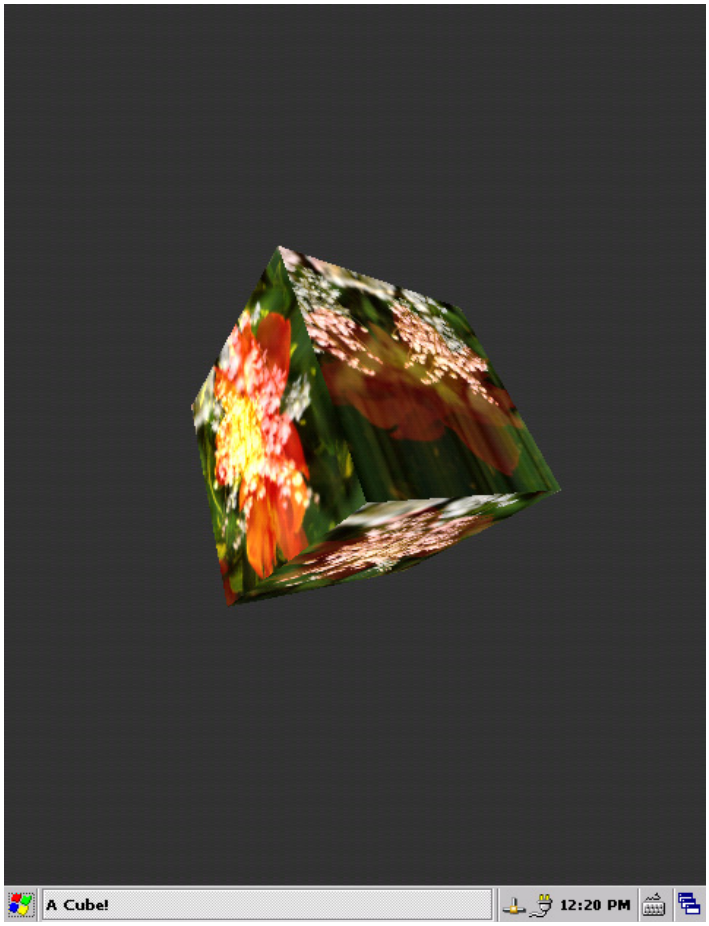


Figure 14-2. Cube Test

14.6.2.6 Triangle Test

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

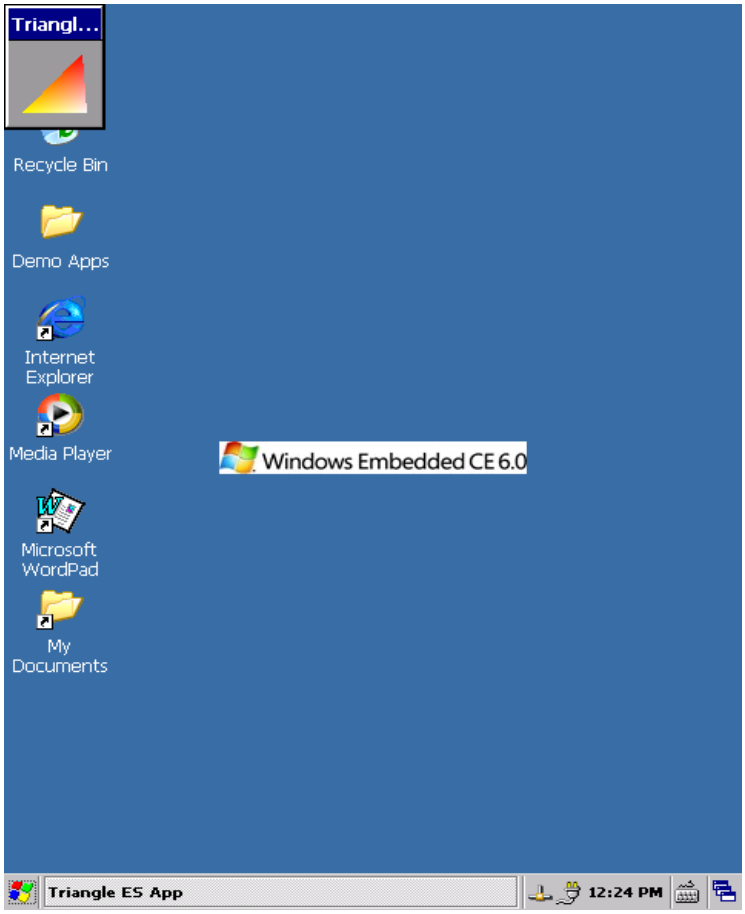


Figure 14-3. Triangle Test

14.6.2.7 Direct3D Mobile CETK Tests

The following tests are standard Direct3D Mobile CETK tests designed by Microsoft:

- Direct3D Mobile Interface Test
- Direct3D Mobile Driver Verification Test
- Direct3D Mobile Driver Comparison Test
- Direct3D Mobile Driver Performance Test

For detailed information about how to run these CETK tests, refer to the MSDN documentation library topic: **Windows Embedded CE Test Kit** > **CETK Tests and Test Tools** > **CETK Tests** > **Display Tests**. The corresponding subtopics describe these Direct3D Mobile CETK tests.

14.6.2.8 OpenGL ES 1.1/2.0 Conformance Test

The OpenGL ES 1.1 and 2.0 conformance tests are standard OpenGL ES conformance test designed by the Khronos Group. Visit the Khronos Group website at http://www.khronos.org/opengles/adopters/login/ for detailed information about how to download the source code, build the test binaries and run these tests.

14.6.2.9 Known Issues

- The Direct3D Mobile Driver Comparison Test #2805 case fails due to hardware limitation
- Refer to the release notes for up-to-date known issue list

14.7 GPU Driver API Reference

- For OpenGL ES 1.1 and 2.0 API refer to http://www.khronos.org/opengles/ for detailed specifications
- For EGL 1.3 API refer to http://www.khronos.org/egl/ for detailed specifications
- For Direct3D Mobile API refer to the MSDN documentation library topic: **Windows Embedded CE Features** > **Graphics** > **Direct3D Mobile**
- For OpenVG 1.1 API refer to http://www.khronos.org/openvg/ for detailed specifications

Chapter 15
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.

15.1 I^2C Driver Summary

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

Table 15-1. I^2C Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\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 or BSP_I2CBUS2=1

15.2 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

15.3 Hardware Operation

15.3.1 Conflicts with Other Peripherals and Catalog Items

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

15.3.1.1 Conflicts with SoC Peripherals

No conflicts.The i.MX51 platform contains two I^2C modules, I2C1 and I2C2. The I2C1 module shares pins with the eCSPI1 module in the i.MX51 hardware. Therefore, both the I2C1 and eCSPI1 drivers cannot be included in the BSP workspace at the same time. There is no conflict with the I2C2 module.

15.3.1.2 Conflicts with Board Peripherals

No conflicts.

15.4 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.

15.4.1 Registry Settings

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

15.4.1.1 i.MX51 Registry Settings

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

```
; I2C Driver
;
IF BSP_I2CBUS1
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\I2C1]
   "Prefix"="I2C"
   "Dll"="i2c.dll"
   "Index"=dword:1
   "Order"=dword:2
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"          ; PMCLASS_GENERIC_DEVICE
ENDIF    ; BSP_I2CBUS1

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

```
ENDIF   ; BSP_I2CBUS2
```

The following is the registry key to load the I2C.

15.4.2 Communicating with the I2C

The I2C is a stream interface driver, and is thus accessed through the file system APIs. To communicate using the I2C, a handle to the device must first be created using the **CreateFile** function. Subsequent commands to the device are issued using the **DeviceIoControl** function with IOCTL codes specifying the desired operation. The following are the basic steps. The I2C driver is provided to hide all the IOCTL calls from the calling application.

15.4.3 Creating a Handle

Call the **CreateFile** function to open a connection to the I2C device. An I2C port must be specified in this call. The format is `I2CX:`, with `X` being the number indicating the I2C port. This number should not exceed the number of I2C instances on the platform. If an I2C port does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the I2C:

1. Insert a colon after the I2C port for the first parameter, *lpFileName*. For example, specify `I2C1:`.
2. Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to an I2C port are supported by the driver.
3. Specify OPEN_EXISTING in the *dwCreationDisposition* parameter. This flag is required.
4. Specify FILE_FLAG_RANDOM_ACCESS in the *dwFlagsAndAttributes* parameter.

Example 15-1shows how to open an I2C port.

Example 15-1. Code to Open I2C 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 I2C port, configure the port. When an application opens an I2C port, it uses the default configuration settings, which might not be suitable for the device at the other end of the connection.

15.4.4 Configuring the I2C

Configuring the I2C port for communications involves two main operations:

- Setting the master or slave mode
- Setting the I2C clock rate

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

Example 15-2 shows the code to configure an I^2C port:

Example 15-2. Code to Configure I^2C Port

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

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

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

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

15.4.5 Data Transfer Operations

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

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

To perform an I^2C transfer:

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

NOTE

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
        }
```

15.4.5.1 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.

15.4.6 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.

15.4.7 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.

15.4.7.1 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.

15.4.7.2 PowerDown

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

15.4.7.3 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.

15.5 Unit Test

The following section explains about the hardware and software requirements for unit tests.

15.5.1 Unit Test Hardware

The unit tests are not supported for this release.

15.5.2 Unit Test Software

The unit tests are not supported for this release.

15.5.3 Building the Unit Tests

The unit tests are not supported for this release.

15.5.4 Running the Unit Tests

The unit tests are not supported for this release.

15.6 Hardware Limitations

The following is the hardware limitation:

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

15.7 I^2C Driver API Reference

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

15.7.1 I^2C Driver IOCTLS

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

15.7.1.1 I2C_IOCTL_GET_CLOCK_RATE

This **DeviceIoControl** request retrieves the clock rate

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

Parameters

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

15.7.1.2 I2C_IOCTL_GET_SELF_ADDR

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

Parameters

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

15.7.1.3 I2C_IOCTL_IS_MASTER

This **DeviceIoControl** request determines whether the I^2C is currently in Master mode.

Parameters

lpOutBuffer	Pointer to a BYTE that contains the return value from the Master mode inquiry: TRUE if currently in Master mode; FALSE if currently in Slave mode
nOutBufferSize	Size in bytes of the return value, should be one byte

15.7.1.4 I2C_IOCTL_IS_SLAVE

This **DeviceIoControl** request determines whether the I^2C is currently in Slave mode.

Parameters

lpOutBuffer	Pointer to a BYTE that contains the return value from the Slave mode inquiry: TRUE if currently in Slave mode; FALSE if currently in Master mode
nOutBufferSize	Size in bytes of the return value, should be one byte

15.7.1.5 I2C_IOCTL_RESET

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

15.7.1.6 I2C_IOCTL_SET_CLOCK_RATE

This **DeviceIoControl** request initializes the I^2C device with the given clock rate. This IOCTL does not expect to receive the absolute peripheral clock frequency. Rather, it expects the clock rate divisor index stated in the I^2C specification. If absolute clock frequency must be used, use the macro I2C_MACRO_SET_FREQUENCY.

Parameters

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

15.7.1.7 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, see the macro I2C_MACRO_GET_CLOCK_RATE to determine the estimated index.

Parameters

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

15.7.1.8 I2C_IOCTL_SET_MASTER_MODE

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

15.7.1.9 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.

15.7.1.10 I2C_IOCTL_SET_SLAVE_MODE

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

15.7.1.11 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

15.7.1.12 I2C_IOCTL_ENABLE_SLAVE

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

15.7.1.13 I2C_IOCTL_DISABLE_SLAVE

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

15.7.1.14 I2C_IOCTL_GET_SLAVESIZE

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

15.7.1.15 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.

15.7.1.16 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.

15.7.1.17 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.

15.7.2 I^2C Driver SDK Encapsulation

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

15.7.2.1 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

15.7.2.2 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

15.7.2.3 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

15.7.2.4 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

15.7.2.5 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

15.7.2.6 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

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

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

15.7.2.9 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, see the macro I2CGetClockRate to determine the estimated index.

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

Parameters

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

15.7.2.10 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

15.7.2.11 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

15.7.2.12 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

15.7.2.13 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 I2C device handle retrieved from CreateFile()

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

15.7.2.14 I2CEnableSlave

This function enables a I2C slave access from the bus. After the I2C slave interface is enabled, the I2C slave driver waits for an external master access.

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

Parameters

hDev I2C device handle retrieved from CreateFile()

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

15.7.2.15 I2CDisableSlave

This function disables I2C slave access from the bus. Note that after the I2C slave interface disabled, I2C slave module can be turned off.

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

Parameters

hDev I2C device handle retrieved from CreateFile()

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

15.7.2.16 I2CGetSlaveSize

This function returns the I2C slave interface buffer length. The I2C slave driver directly returns data to the master from the interface buffer. The interface buffer can be set at any time, even when the I2C slave module has been turned off.

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

Parameters

hDev I2C 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

15.7.2.17 I2CSetSlaveSize

This function sets the I2C 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 I2C slave driver directly returns data to the master from the

interface buffer. The interface buffer can be set at any time, even when the I^2C slave module has been turned off.

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

Parameters

hDev	I^2C device handle retrieved from CreateFile()
dwSize	DWORD variable that sets interface buffer length
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

15.7.2.18 I2CGetSlaveText

This function returns the I^2C slave interface buffer text. The I^2C slave driver directly returns data to the master from the interface buffer. The interface buffer can be accessed at any time, even when the I^2C slave module has been turned off.

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

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pbyTextBuf	User buffer to store text returned from interface buffer
pdwBufSize	User buffer size
pdwTextLen	Actual data bytes returned
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

15.7.2.19 I2CSetSlaveText

This function returns the I^2C slave interface buffer text. The I^2C slave driver directly returns data to the master from the interface buffer. The interface buffer can be accessed at any time, even when the I^2C slave module has been turned off.

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

Parameters

hDev	I^2C device handle retrieved from CreateFile()
pbyTextBuf	User buffer to store text to interface buffer
dwTextLen	Text length in user buffer
Return Values	Returns TRUE or FALSE, if the result is TRUE, the operation is successful

15.7.3 I^2C Driver Structures

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

15.7.3.1 I2C_PACKET

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

```
typedef struct {
        BYTE byAddr;
        BYTE byRW;
        PBYTE pbyBuf;
        WORD wLen;
        LPINT lpiResult;
} I2C_PACKET, *PI2C_PACKET;
```

Parameters

byAddr 7-bit slave address that specifies the target I^2C device to or from which data is read or written

byRW Determines whether the packet is a read or a write packet. Set to I2C_RW_READ for reading and I2C_RW_WRITE for writing.

Set to I2C_POLLING_MODE to force polling mode for transfer.

pbyBuf Pointer to a buffer of bytes. For a read operation, this is the buffer into which data is read. For a write operation, this buffer contains the data to write to the target device.

wLen If the operation is a read, wLen specifies the number of bytes to read into pbyBuf. If the operation is a write, wLen specifies the number of bytes to write from pbyBuf.

lpiResult Pointer to an int that contains the return code from the transfer operation

15.7.3.2 I2C_TRANSFER_BLOCK

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

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

Parameters

pI2CPackets Pointer to an array of I2C_PACKET objects

iNumPackets Number of I2C_PACKET objects pointed to by pI2CPackets

Chapter 16
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).

16.1 Keypad Driver Summary

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

Table 16-1. Keypad Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	KEYBD
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\KEYBD
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\KEYBD
Driver DLL	kbdmouse.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale i.MX51 EVK : ARMV4I > Device Drivers > Input Devices > Keyboard/Mouse > Freescale iMX51-EVK 16-key keypadThird Party > BSP > Freescale i.MX51 EVK : ARMV4I > Device Drivers > Input Devices > Keyboard/Mouse > Freescale iMX51-EVK 16-key keypad
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_NOKEYPAD=

16.2 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. Supports the 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.

16.3 Hardware Operation

16.3.1 Conflicts with Other Peripherals and Catalog Items

16.3.2 Keypad

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

16.3.2.1 i.MX51 EVK Keypad Mapping

The 16-key keypad is located on the accessory card and the mapping is shown in Table 16-2.

Table 16-2. Keypad Mapping

Label	Key
SW40	DOWN
SW36	UP
SW34	ESC
SW32	TAB
SW39	LEFT
SW31	RIGHT
SW18	ENTER
SW17	ALT
SW38	ENTER
SW29	ENTER
SW14	ENTER
SW13	ENTER
SW37	BACKSPACE
SW30	BACKSPACE
SW10	BACKSPACE
SW9	BACKSPACE

16.4 Software Operation

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

16.4.1 Keypad Scan Codes and Virtual Keys

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

16.4.1.1 i.MX51 EVK Scan Code Mapping Table

Table 16-3 shows the scan code mapping.

Table 16-3. Scan Code Mapping

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

16.4.2 Power Management

The following power management functions are used by the keypad driver.

16.4.2.1 BSPKppPowerOn

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

16.4.2.2 BSPKppPowerOff

This function powers down the keypad.

16.4.2.3 IOCTL_POWER_CAPABILITIES

This function is not implemented for the keypad driver.

16.4.2.4 IOCTL_POWER_SET

This function is not implemented for the keypad driver.

16.4.2.5 IOCTL_POWER_GET

This function is not implemented for the keypad driver.

16.4.3 Keypad Registry Settings

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

16.5 Unit Test

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

.

16.5.1 Unit Test Hardware

- i.MX51 EVK board
- Accessory card for i.MX51 EVK board
-

16.5.2 Unit Test Software

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

16.5.3 Building the Unit Tests

No additional steps are required to build the keypad tests.

16.5.4 Running the Unit Tests

The procedure of keyboard tests is as follows:

1. Run Microsoft WordPad application
2. Input Tab
3. Input Alt to open the menu bar
4. Input Up Down Left and Right
5. Run the Internet Explorer application
6. Open the help document by click the question mark on Internet Explorer application
7. Input the ESC to quit from help document
8. Input Alt + Tab to call the Task Manager, input Enter
9. Input Enter into the content and Input Backsapce to delete
10. Quit Microsoft WordPad, there is a pop up dialog box, click the Yes button

NOTE

Befoe running this test, ensure that the WordPad items are included in the project (SYSGEN_PWORD).

11.

16.6 Keypad Driver API Reference

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

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

16.6.1 Keypad PDD Functions

Table 16-4 shows a mapping of the keyboard PDD functions to the functions used in the keypad driver:

Table 16-4. 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 17
Notification LED Driver

The notification LED (NLED) is used to notify the user about the occurrence of an event.

17.1 Notification LED Driver Summary

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

Table 17-1. Notification LED Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	N/A
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\NLED
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\NLEDDRVR
Import Library	N/A
Driver DLL	nleddrvr.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > NLED
SYSGEN Dependency	SYSGEN_NLED=1
BSP Environment Variables	Remove BSP_NONLED

17.2 Supported Functionality

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

1. One Notification LED (STAT0)

17.3 Hardware Operation

Refer to the Peripheral Bus Controller document for hardware implementation details for the NLED.

17.3.1 Conflicts with Other SoC peripherals

17.3.1.1 i.MX51 EVK Peripheral Conflicts

No conflicts.

17.4 Software Operation

17.4.1 Communicating with the Notification LED

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

17.4.2 Creating a Handle to the Notification LED

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

To open a handle to the NLED:

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

   For example, specify `NLD1:` for STAT0.
2. Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to an NLED are supported by the driver.
3. Specify OPEN_EXISTING in the *dwCreationDisposition* parameter. This flag is required.
4. Specify FILE_FLAG_RANDOM_ACCESS in the *dwFlagsAndAttributes* parameter.

The following code example shows how to open an NLED.

```
// Open the NLED.
    HANDLE hNLeddrvr;
    hNLeddrvr = CreateFile (TEXT ("NLD1:"),                  // name of device
                        GENERIC_READ|GENERIC_WRITE,          // desired access
                        FILE_SHARE_READ|FILE_SHARE_WRITE,    // sharing mode
                        NULL,                                // security attributes (ignored)
                        OPEN_EXISTING,                       // creation disposition
                        FILE_FLAG_RANDOM_ACCESS,             // flags/attributes
                        NULL);
```

17.4.3 Configuring the Notification LED

Once the handle for the driver is obtained, it can be used for configuring parameters used by the NLED driver by calling the **DeviceIoControl** function with appropriate IOCTL. The configuration parameters are defined in the structure **NLED_SETTINGS_INFO**. The following parameters can be set:

- Time cycle of a blink
- On time of the cycle
- Off time of the cycle

- Number of on blink cycles
- Number of off blink cycles

Refer to Platform Builder documentation for more details about the structure:

Developing a Device Driver > Windows Embedded CE Drivers > Notification LED Drivers > Notification LED Driver Reference > Notification LED Driver Structures

Refer to Section 17.6, "NLED Driver API Reference," for more details on the NLED driver IOCTLs.

17.4.4 Closing the Handle of the Notification LED

Call the **CloseHandle()** function to close a handle to the NLED when an application is done using it. **CloseHandle()** has one parameter, the handle returned by the CreateFile function that opened the NLED.

```
CloseHandle (hNLeddrvr);
```

17.4.5 Power Management

There is no clock that needs to be enabled for this module.

17.4.5.1 PowerUp

This function brings back the state of the NLED to its original state (the state just before the power down sequence happened).

17.4.5.2 PowerDown

This function turns off the NLED to save the power and is called while entering the suspend state.

17.4.5.3 IOCTL_POWER_CAPABILITIES

This function is not implemented for the NLED driver.

17.4.5.4 IOCTL_POWER_SET

This function is not implemented for the NLED driver.

17.4.5.5 IOCTL_POWER_GET

This function is not implemented for the NLED driver.

17.4.6 Notification LED Registry Settings

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

17.4.6.1 i.MX51 EVK NLED Registry Settings

```
; HIVE BOOT SECTION
```

```
[HKEY_LOCAL_MACHINE\System\Events]
    "SYSTEM/NLedAPIsReady"="Notification LED APIs"

; END HIVE BOOT SECTION

; These registry entries load the NLed driver. The IClass value must match
; the NLED_DRIVER_CLASS definition in nled.h -- this is how the system
; knows which device is the battery driver. Note that we are using
; DEVFLAGS_NAKEDENTRIES with this driver. This tells the device manager
; to instantiate the device with the prefix named in the registry but to look
; for DLL entry points without the prefix. For example, it looks for Init
; instead of NLD_Init. This allows the prefix to be changed in the registry (if
; desired) without editing the driver code.
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\NLed]
   "Prefix"="NLD"
   "Dll"="nleddrvr.dll"
   "Flags"=dword:8                  ; DEVFLAGS_NAKEDENTRIES
   "Order"=dword:0
   "Index"=dword:1
   "IClass"="{CBB4F234-F35F-485b-A490-ADC7804A4EF3}"
```

17.5 Unit Test

The NLED CETK test cases verify the functionality of NLED driver.

17.5.1 Unit Test Hardware

Table 17-2 lists the required hardware to run the unit tests.

Table 17-2. Hardware Requirements

Requirement	Description
No additional hardware required	

17.5.2 Unit Test Software

Table 17-3 lists the required software to run the unit tests.

Table 17-3. Software Requirements

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

17.5.3 Building the NLED Tests

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

Windows Embedded CE Test Kit > **Running the CETK**

17.5.4 Running the NLED Tests

The command line for running the NLED tests is:

```
tux -o -d nledtest
```

NOTE

Test cases 2000 and 2100 (Get Invalid Parameters Test and Set NLED_INFO Invalid Parameters) pass an invalid pointer to the driver to verify that the return code is FALSE. When this invalid pointer is not NULL, the driver causes an Exception Data Abort when trying to visit the invalid address. The system goes into kernel debug mode if the kernel debugger is enabled. It is strongly suggested that the kernel debugger should not be enabled when running these two NLED CETK tests.

To disable the kernel debugger, deselect the kernel debugger module:

Project > **Properties** > **Configuration Properties** > **Build Options** > **Enable kernel debugger (no IMGNODEBUGGER=1)**

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

Windows EmbeddedCE Test Kit > **CETK Tests and Test Tools** > **CETK Tests** > **Nled Tests**

17.6 NLED Driver API Reference

17.6.1 NLED Driver IOCTLS

This section contains the descriptions for NLED I/O control codes (IOCTLs). These IOCTLs are used in calls to DeviceIoControl to issue commands to the NLED. Only descriptions for relevant IOCTL parameters are provided.

17.6.1.1 IOCTL_NLED_GETDEVICEINFO

This **DeviceIoControl** request can retrieve NLED device information. This information includes the number of NLEDs supported by the driver, NLED support information and NLED settings information. The kind of information required needs to be specified in the lpInBuffer parameter of the **DeviceIoControl** request. Refer to Platform Builder documentation for more details on using the NLED structures: **Developing a Device Driver** > **Windows CE Drivers** > **Notification LED Drivers** > **Notification LED Driver Reference**

Parameters

lpInBuffer	Pointer to a buffer that contains the information request. This request can be anyone of the following: NLED_COUNT_INFO_ID NLED_SUPPORTS_INFO_ID NLED_SETTINGS_INFO_ID
nInBufferSize	Buffer to have a size of UINT
lpOutBuffer	Pointer to NLED_SUPPORTS_INFO structure where the queried information is stored
nOutBufferSize	Size in bytes of the structure NLED_SUPPORTS_INFO

17.6.1.2 IOCTL_NLED_SETDEVICE

This **DeviceIoControl** is used to change the configuration settings of the LED. The data to be updated is stored in the input buffer which is pointed to by lpInBuffer. Refer to Platform Builder documentation for detailed information about the structure: **Developing a Device Driver** > **Windows CE Drivers** > **Notification LED Drivers** > **Notification LED Driver Reference** > **Notification LED Driver Structures**

Parameters

lpInBuffer	Pointer to a buffer that contains the data to be updated. This data is the NLED_SETTINGS_INFO structure which contains the new configuration details.
nInBufferSize	Size in bytes of the structure NLED_SETTINGS_INFO

Chapter 18
One-Wire (OWIRE) Driver

The One-Wire driver provides a communication channel through One-Wire interface with the DS2438 Smart Battery Monitor by writing or reading one bit data at a time.

18.1 One-Wire Driver Summary

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

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\OWIRE
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\OWIRE
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\OWIRE
Driver DLL	owire.dll
SDK Library	owiresdk_COMMON_FSL_V2.lib
Catalog Item	Third Party —> BSP —> Freescale <*Target Platform*>: ARMV4I —> Device Drivers —> One-Wire Interface
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_OWIRE=1

18.2 Supported Functionality

The One-Wire driver enables the board to provide the following software and hardware support:

1. Support the Windows CE streams interface.
2. Support communicating with 1-Wire devices through One-Wire Interface.
3. Support the DS2438 for i.MX51.

18.3 Hardware Operation

Refer to the chapter on One-Wire Interface in the hardware specification document for detailed operation and programming information.

18.3.1 Conflicts with other Peripherals and Catalog Items

No Conflicts.

18.3.1.1 Conflicts with SoC Peripherals

18.3.1.1.1 i.MX51 Peripheral Conflicts

No Conflicts.

18.3.1.2 Conflicts with Board Peripherals

18.3.1.2.1 i.MX51 EVK Peripheral Conflicts

The OWIRE pin conflicts with the SPDIF_OUT pin.

18.4 Software Operation

18.4.1 Communicating with the One-Wire Interface

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

18.4.2 Creating a Handle to the One-Wire Interface

Call the **OwireOpenHandle( )** function that is defined in the driver file. This function will in turn call the **CreateFile** function to open a connection to the One-wire Interface. If a One-wire port does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the One-wire Interface:

Call the **OwireOpenHandle( )** function which would return a handle to the One-wire Interface.

The following code example shows how to open a One-wire device.

```
// Open handle to the OWIRE device
 hOwire = OwireOpenHandle ();
 if (hOwire == NULL)
 {
     g_pKato->Log(1, (TEXT("InitializeTests: OwireOpenHandle failed!\r\n")));
     return TPR_FAIL;
 }
```

18.4.3 Configuring the One-Wire Interface

N/A

18.4.4 Bus Lock / Unlock

One-Wire driver could support multiple applications to communicate with one or more devices on the bus. To support this case, each application should use **OwireBusLock()** / **OwireBusUnlock()** functions properly to avoid any interference.

To lock the One-Wire bus:

Call the **OwireBusLock()** function before any Read / Write operations.

To unlock the One-Wire bus:

Call the **OwireBusUnlock()** function after the Read / Write operations are completed or any error occurs.

18.4.5 Write Operations

Before initiating the write operation on the One-wire Interface bus, it is required to reset the One-wire device and wait for the Device Presence Pulse to ensure that a One-Wire device is connected to the interface. This can be accomplished by calling the driver function **OwireResetPresencePulse( ),** along with a handle to the One-wire Interface passed as a parameter. This **OwireResetPresencePulse( )** will in turn issue an OWIRE_IOCTL_RESET_PRESENCE_PULSE IOCTL which would be handled by the driver.

The following code example shows this presence detect operation.

```
if (!OwireResetPresencePulse(hOwire))
      {
           OwireBusUnLock(hOwire);
           g_pKato—>Log(OWIRE_ZONE_ERROR,
                (TEXT("OwireWriteMemTest: OwireResetPresencePulse() failed!\r\n")));
           return TPR_FAIL;
      }
```

Once a One-Wire device is detected, we can continue with the write operation. The One-wire Interface write operation involves exchange of commands and responses, before the data is finally written into the EEPROM. **OwireWrite()** is used to send the protocol command or data to the One-Wire device.

To write command / data into the One-Wire device using One-wire Interface:

1. Create a write buffer of bytes to hold the commands and the actual data.
2. Call the driver function **OwireWrite( )** along with the following set of parameters:

- Handle to the One-wire interface which was previously acquired using the **OwireOpenHandle( )** function.
- Address of the write buffer that was created.
- The number of bytes that were written into the write buffer.

This function will in turn call the **WriteFile** API to write the command and the data into the One-wire Interface bus.

The following code example shows Write Scratchpad command being issued.

```
BYTE WriteSPCMD[] = {0xCC, 0x4E, 0x05};
if (!OwireWrite(hOwire, WriteSPCMD, 3))
```

```
    {
        OwireBusUnLock(hOwire);
        g_pKato->Log(OWIRE_ZONE_ERROR,
            (TEXT("OwireReadMemTest: OwireWrite failed!\r\n")));
        return TPR_FAIL;
    }
```

18.4.6 Read Operations

Before initiating the read operation on the One-wire Interface bus, it is required to reset the One-wire interface and wait for the Device Presence Pulse to ensure that a One-Wire device is connected to the interface. This can be accomplished by calling the driver function **OwireResetPresencePulse( ),** along with a parameter which is a handle to the One-wire interface. This **OwireResetPresencePulse( )** will in turn issue an IOCTL OWIRE_IOCTL_RESET_PRESENCE_PULSE which would be handled by the driver.

Once the One-Wire device connection is detected, we can continue with the read operation. The One-wire Interface write operation involves exchange of commands and responses, before the data is finally read from the One-Wire device. **OwireWrite()** is used to send the protocol command or data to the One-Wire device. **OwireRead()** is used to read the response or data from the One-Wire device.

Once this READ command is issued, the bus master will start issuing read time slots and receive data from One-Wire device.

To read data from One-Wire device using One-wire Interface:

1. Create a read buffer of bytes to hold the data read from One-Wire device.
2. Call the driver function **OwireRead( )** along with the following set of parameters:

- Handle to the One-wire Interface which was previously acquired using the **OwireOpenHandle( )** function.
- Address of the read buffer that was created.
- The number of bytes that is to be read from the read buffer.

This function will in turn call the **ReadFile( )** API to read the data from the One-Wire device.

The following code example shows how to read 8 bytes of data from the One-Wire device.

```
if (!OwireRead(hOwire, data, 8))
    {
        OwireBusUnLock(hOwire);
        g_pKato->Log(OWIRE_ZONE_ERROR,
            (TEXT("OwireReadMemTest: OwireRead failed!\r\n")));
        return TPR_FAIL;
    }
```

18.4.7 Closing the Handle to the One-Wire Interface

Call the **OwireCloseHandle( )** driver function to close a handle to the One-wire Interface when an application is done using it. This function will in turn call the **CloseHandle** API.

OwireCloseHandle( ) has one parameter, which is the handle returned by the **OwireOpenHandle( )** function call that opened the One-wire Interface.

18.4.8 Power Management

The primary method for limiting power consumption in the One-wire module is to gate off all clocks to the module when those clocks are not needed. This is accomplished through the **DDKClockSetGatingMode** function call. One-wire module clock is enabled only when the module is initialing / reading / writing / setting ResetPresencePulse.

Moreover, the One-wire module goes into low power mode by gating off the clock automatically whenever it is not in use i.e. whenever it is not communicating with One-Wire device. Refer to the chapter on One-wire Interface in the hardware specification document for detailed description of this.

18.4.8.1 PowerUp

This function is not implemented for the One-wire driver.

18.4.8.2 PowerDown

This function is not implemented for the One-wire driver.

18.4.8.3 IOCTL_POWER_CAPABILITIES

N/A

18.4.8.4 IOCTL_POWER_SET

N/A

18.4.8.5 IOCTL_POWER_GET

N/A

18.4.9 Registry Settings

18.4.9.1 i.MX51 Registry Settings

The following registry keys are required to properly load the One-Wire driver.

```
;-------------------------------------------------------------------------------
; One Wire Driver
;
IF BSP_OWIRE
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\OWIRE]
   "Prefix"="WIR"
   "Dll"="owire.dll"
   "Index"=dword:1
   "Order"=dword:1
ENDIF ; BSP_OWIRE
;-------------------------------------------------------------------------------
```

18.5 Unit Test

The One-wire tests verify that the One-wire driver properly initializes the One-wire support.

18.5.1 Unit Test Hardware

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

Requirements	Description
The <TGTPLAT> board with DS2438 as the One-Wire chip.	

18.5.2 Unit Test Software

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

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

18.5.3 Building the One-Wire Tests

In order to build the One-wire tests, complete the following steps:

1. Build an OS image for the desired configuration
2. Within Platform Builder, go to the **Build OS** menu option and select the **Open Release Directory** menu option. This will open a DOS prompt.
3. Change to the OWIRETest directory. (\WINCE600\SUPPORT\APP\OWIRETest).
4. Enter **set WINCEREL=1** on the command prompt and hit return. This will copy the built DLL to the flat release directory.
5. Enter the build command at the prompt and press return.

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

18.5.4 Running the One-Wire Tests

The command line for running the One-wire tests is ***tux –o –d OnewireTest***. The One-wire tests do not contain any test specific command line options.

The following table describes the test cases contained in the One-wire tests.

Test Case	Description	Parameters	Remarks
Owire Soft Reset	This test issues OWIRE_IOCTL_RESET_PRESENCE_PULSE IOCTL and test for the OWIRE module soft reset functionality.	None	The One-Wire device is DS2438 on iMX51 EVK / board.
OwireReadStatus Test	Reads the One-Wire device's family code, unique 48-bit serial number and 8-bit CRC.	None	The One-Wire device is DS2438 on iMX51 EVK board.
Owire Read Memory	Reads data from the One-Wire device and displays.	None	The One-Wire device is DS2438 on iMX51 EVK board. This test reads and displays Temperature / Voltage information.
Owire Write Memory	Writes data into the One-Wire device and reads back the data and verifies whether the data was written correctly into the memory.	None	The One-Wire device is DS2438 on iMX51 EVK board.

18.6 One-Wire Driver API Reference

18.6.1 One-Wire Driver SDK Wrapper Functions

This section consists of descriptions for the One-wire driver SDK wrapper functions. These wrapper functions are used in accessing the stream interface for the One-Wire driver.

18.6.1.1 OwireOpenHandle()

This function creates a handle to the One-Wire driver. It returns the handle to One_wire driver. If failure, the function returns INVALID_HANDLE_VALUE.

Parameters

NULL

18.6.1.2 OwireCloseHandle()

This function closes a handle to the One-Wire driver.

Parameters

hOwire Handle to close

18.6.1.3 OwireResetPresencePulse()

This function performs a One-Wire reset sequence with a reset pulse and presence pulse.

Parameters

hOwire Handle to the One-Wire driver

18.6.1.4 OwireRead()

This function attempts to read data from the One-Wire device.

Parameters

hOwire	Handle to the One-Wire driver
readBuf	Pointer to buffer containing bytes read from the One-Wire device
bytesToRead	Number of bytes to read from the One-Wire device

18.6.1.5 OwireWrite()

This function attempts to writedata to the One-Wire device.

Parameters

hOwire	Handle to the One-Wire driver
writeBuf	Pointer to buffer containing bytes to write to the One-Wire device
bytesToWrite	Number of bytes to write to the One-Wire device

18.6.1.6 OwireBusLock()

This function attempts to lock the One-Wire bus.

Parameters

hOwire	Handle to the One-Wire driver

18.6.1.7 OwireBusUnLock()

This function attempts to unlock the One-Wire bus.

Parameters

hOwire	Handle to the One-Wire driver

18.6.2 One-Wire Driver Structures

N/A

Chapter 19
Power Management IC (PMIC)

19.1 PMIC Summary

This chapter provides information to develop:

- Device drivers that interface directly to the Freescale power management IC (PMIC) hardware components. The PMIC that is specifically referenced in this document is the MC13892.
- Applications that use the special hardware capabilities that are provided by the PMIC (for example, touch I/O, BackLight function.).

This chapter describes the API provided by Freescale which allows complete access to the functionality of the PMICs. This document is intended for device driver and application developers who need to understand and gain access to the functionality provided by the PMICs. Table 19-1 provides a summary of source code location, library dependencies and other BSP information.

Table 19-1. PMIC Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\PMIC\MC13892
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\PMIC\MC13892
Driver DLL	pmicPdk_mc13892.dll
SDK Library	pmicSdk_mc13892.lib
Catalog Item(s)	N/A
SYSGEN Dependency	N/A
BSP Environment Variable(s)	BSP_NOPMIC=

19.2 Supported Functionality

The PMIC device driver framework for Windows CE is a stream interface driver and a SDK DLL. A description of the stream interface driver may be found in the Windows CE Platform Builder documentation at **Developing a Device Driver** > **Windows CE Drivers** > **Stream Interface Drivers**.

The PMIC Stream Interface driver controls the PMIC hardware directly using the SPI or I^2C bus. The Stream Interface driver provides an IOCTL interface for SDK DLLs. The SDK DLLs provide APIs for Windows CE drivers and applications.

The API covers the PMIC functionality of the following areas:

1. Register Access
2. Tri-Color LED
3. Battery
4. Regulators
5. Keys (Power, PTT)
6. ADC /Touch
7. Backlight (Keyboard, LCD)
8. Battery Charger
9. GPO

19.3 Hardware Operation

Refer to the MC13892 document for details on the MC13892 PMIC.

19.3.1 Conflicts with Other On-Chip Peripherals

19.3.1.1 i.MX51 Peripheral Conflicts

No conflicts.

19.3.2 Conflicts with Other EVK Peripherals

No conflicts.

19.4 Software Operation

19.4.1 Configuring the PMIC

The PMIC modules can be used by applications or device drivers. For example, the battery API of the PMIC is used by the battery driver. Configuring the PMIC port for communications involves some basic operations. A handle to the desired PMIC port must be opened prior to accessing the module registers. This handle is required to call the **DeviceIoControl** function. The function parameters include the PMIC port handle, appropriate IOCTL code, and other input and output parameters.

19.4.2 Creating a Handle to the PMIC

Before calling any PMIC API make sure that the PMIC device is attached by calling the **CreateFile** function which opens a file and it returns a handle that can be used to access the MC13892 hardware. If the MC13892 hardware does not exist, **CreateFile** returns ERROR_FILE_NOT_FOUND.

To open a handle to the PMIC:

1. Insert a colon after the PMI1 port for the first parameter, *lpFileName*.
   For example, specify PMI1: as the PMIC port.
2. Specify FILE_SHARE_READ | FILE_SHARE_WRITE in the *dwShareMode* parameter. Multiple handles to a PMIC port are supported by the driver.
3. Specify OPEN_EXISTING in the *dwCreationDisposition* parameter. This flag is required.
4. Specify FILE_FLAG_RANDOM_ACCESS in the *dwFlagsAndAttributes* parameter.

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

```
// Open the PMIC port.
hPMI =   CreateFile(TEXT("PMI1:"),
         GENERIC_READ | GENERIC_WRITE,                // access (read-write) mode
         FILE_SHARE_READ | FILE_SHARE_WRITE,                    // sharing mode
         NULL,                                     // security attributes (ignored)
         OPEN_EXISTING// sharing mode                      // creation disposition
         FILE_FLAG_RANDOM_ACCESS,                          //flags and attributes
         NULL);                                      // template file (ignored)
if ((hPMI == NULL) || (hPMI == INVALID_HANDLE_VALUE))
{
         ERRORMSG(1, (_T("Failed in create File()\r\n")));
}
```

19.4.3 Write Operations

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

19.4.4 Read Operations

Like the write operation, the PMIC driver does not provides for reading through the PMIC_Read function. This is a stub function and always returns success.

19.4.5 Closing the Handle to the PMIC

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

19.4.6 Power Management

The primary method for limiting power consumption in the PMIC module is to gate off all clocks to the module when those clocks are not needed. This is accomplished through the **DDKClockSetGatingMode**

function call. The PMIC module clock is enabled whenever any of the PMIC registers need to be accessed and then disabled once it is done.

19.4.6.1 PowerUp

This function is not implemented for the PMIC driver.

19.4.6.2 PowerDown

This function is not implemented for the PMIC driver.

19.4.6.3 IOCTL_POWER_CAPABILITIES

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

19.4.6.4 IOCTL_POWER_SET

This IOCTL requests a change from one device power state to another. D0 and D4 are the only two supported **CEDEVICE_POWER_STATE** in the PMIC driver. Any request that is not D0 is changed to a D4 request and results in the system entering into suspend state. For a request of value of D0, the system is resumed.

19.4.6.5 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.

19.4.7 PMIC Registry Settings

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\PMI]
   "Prefix"="PMI"
   "Dll"="pmicpdk_mc13892.dll"
   "Index"=dword:1
   "Order"=dword:2
   "IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"     ; PMCLASS_GENERIC_DEVICE
```

19.4.8 DMA Support

No support.

19.5 Unit Test

19.5.1 Unit Test Hardware

The EVK and the MC13892 PMIC boards are required.

19.5.2 Unit Test Software

No software is necessary for this test.

19.5.3 Running the PMIC Tests

The PMIC driver can be tested using the following actions:

- The command line for running the PMIC tests is `s i2cpmic`
- Use the touch driver CEKT to test MC13892 touch function
- Use the battery driver CEKT to test MC13892 battery function
- Use the backlight driver CEKT to test MC13892 backlight function

19.6 PMIC Driver API Reference

19.6.1 PMIC Driver IOCTLS

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

19.6.1.1 PMIC_IOCTL_LLA_READ_REG

This DeviceIoControl request reads the register content.

Parameters

hPMI	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
lpInBuffer	Index of the register
lpOutBuffer	[out] Long pointer to a buffer that receives the output data for the operation. Set to NULL if the dwIoControlCode parameter specifies an operation that does not produce output data

19.6.1.2 PMIC_IOCTL_LLA_WRITE_REG

This DeviceIoControl request writes the data to the said register of the PMIC device.

Parameters

hPMI	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
lpInBuffer	Index of the register
lpOutBuffer	Pointer to data which needs to be written to the register

19.6.1.3 PMIC_IOCTL_LLA_INT_REGISTER

This DeviceIoControl is used to register interrupt.

Parameters

hPMI	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
lpInBuffer	Index of the register
lpOutBuffer	Pointer to event name and interrupt ID

Code example:

```
param.int_id = int_id;
param.event_name = event_name;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_REGISTER, &param,sizeof(param), NULL, 0,
NULL, NULL);
```

19.6.1.4 PMIC_IOCTL_LLA_INT_DEREGISTER

This DeviceIoControl is used to deregister PMIC interrupt.

Parameters

hPMI	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
lpInBuffer	Index of the register
lpOutBuffer	Null

Code example:

```
param.int_id = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_DEREGISTER, &param,sizeof(param), NULL,
0, NULL, NULL)
```

19.6.1.5 PMIC_IOCTL_LLA_INT_COMPLETE

Parameters

hPMI	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
lpInBuffer	Index of the register
lpOutBuffer	Pointer to interrupt ID

Code example:

```
param.int_id = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_COMPLETE, &param,sizeof(param), NULL, 0,
NULL, NULL);
```

19.6.1.6 PMIC_IOCTL_LLA_INT_ENABLE

This IOCTL is used to enable the interrupt.

Parameters

hPMI	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
lpInBuffer	Index of the register
lpOutBuffer	Pointer to interrupt ID

Code example:

```
param.int_id = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_COMPLETE, &param,sizeof(param), NULL, 0,
NULL, NULL);
```

19.6.1.7 PMIC_IOCTL_LLA_INT_DISABLE

This IOCTL is used to disable the interrupt.

Parameters

hPMI	[in] Handle to the device that is to perform the operation. To obtain a device handle, call the CreateFile function
lpInBuffer	Index of the register
lpOutBuffer	Pointer to interrupt ID

Code example:

```
param.int_id = int_id;
ret = DeviceIoControl(hPMI, PMIC_IOCTL_LLA_INT_COMPLETE, &param, sizeof(param), NULL, 0,
NULL, NULL);
```

19.6.2 Interrupt Handling

This section describes the interrupt handling of the PMIC driver.

19.6.2.1 Interrupt Handling Overview

The PMIC has interrupt generation capability to inform the CPU when events occur. This is signaled to the processors driving the SPI or I^2C buses. There is only one interrupt line connected to each processor, so the kernel can only know that there is an interrupt from the PMIC, but without knowing exactly which module generated the interrupt.

There is one PMIC Interrupt Service Thread (IST) to handle all interrupts from the PMIC. The PMIC IST is invoked by the kernel once the kernel receives an interrupt from the PMIC. This IST first queries the PMIC to determine the source of the interrupt. The IST maintains a table to track if an interrupt has been registered by a driver or application. If the interrupt is registered, the IST then sets a predefined event. For any drivers and applications that need notification of an interrupt, they must register the interrupt and wait for the event. They also need to reset the event after handling the event.

19.6.2.2 Interrupt Events

Drivers or applications that wish to monitor an interrupt should create a named event for each interrupt. The event name is passed to PMIC driver when registering the interrupt. The PMIC IST triggers the event when the corresponding interrupt occurs.

19.6.2.3 PMIC Interrupt Events

Table 19-2 shows the events and corresponding MC13892 interrupts.

Table 19-2. PMIC Interrupt Events

PMIC Interrupt	Description
ADCDONEI	ADC has finished requested conversions
ADCBISDONEI	ADCBIS has finished requested conversions
TSI	Touch screen wakeup
VBUSVALIDI	VBUSVALID detect
IDFACTORYI	ID factory mode detect
USBOVI	USB over-voltage detection
CHGDETI	Charger attach
CHGFAULTI	Charger fault detection
CHGREVI	Charger path reverse current
CHGSHORTI	Charger path short circuit
CCCVI	Charger path CC / CV transition detect
CHGCURRI	Charge current below threshold warning
BPONI	BP turn on threshold
LOBATLI	Low battery low threshold warning
LOBATHI	Low battery high threshold warning
IDFLOATI	USB ID float detect
IDGNDI	USB ID ground detect
1HZI	1 Hz time tick
TODAI	Time of day alarm
PWRON3I	PWRON3 event
PWRON1I	PWRON1 event
PWRON2I	PWRON2 event
WDIRESETI	WDI system reset event
SYSRSTI	PWRON system reset event
RTCRSTI	RTC reset event
PCI	Power cut event

Table 19-2. PMIC Interrupt Events (continued)

WARMI	Warm start event
MEMHLDI	Memory hold event
LPBI	Low power USB boot detection
THWARNLI	Thermal warning low threshold
THWARNHI	Thermal warning high threshold
CLKI	Clock source change
SCPI	Short circuit protection trip detection
BATTDETBI	Battery removal detect

19.6.2.4 Interrupt Data Structures

```
typedef enum _PMIC_MC13892_INT_ID {
    PMIC_MC13892_INT_ADCDONEI = 0,
    PMIC_MC13892_INT_ADCBISDONEI = 1,
    PMIC_MC13892_INT_TSI = 2,
    PMIC_MC13892_INT_VBUSVALIDI = 3,
    PMIC_MC13892_INT_IDFACTORYI = 4,
    PMIC_MC13892_INT_USBOVI = 5,
    PMIC_MC13892_INT_CHGDETI = 6,
    PMIC_MC13892_INT_CHGFAULTI = 7,
    PMIC_MC13892_INT_CHGREVI = 8,
    PMIC_MC13892_INT_CHGSHORTI = 9,
    PMIC_MC13892_INT_CCCVI = 10,
    PMIC_MC13892_INT_CHGCURRI = 11,
    PMIC_MC13892_INT_BPONI = 12,
    PMIC_MC13892_INT_LOBATLI = 13,
    PMIC_MC13892_INT_LOBATHI = 14,
    PMIC_MC13892_INT_IDFLOATI = 19,
    PMIC_MC13892_INT_IDGNDI = 20,
    PMIC_MC13892_INT_1HZI = 32,
    PMIC_MC13892_INT_TODAI = 33,
    PMIC_MC13892_INT_PWRON3I = 34,
    PMIC_MC13892_INT_PWRON1I = 35,
    PMIC_MC13892_INT_PWRON2I = 36,
    PMIC_MC13892_INT_WDIRESETI = 37,
    PMIC_MC13892_INT_SYSRSTI = 38,
    PMIC_MC13892_INT_RTCRSTI = 39,
    PMIC_MC13892_INT_PCI = 40,
    PMIC_MC13892_INT_WARMI = 41,
    PMIC_MC13892_INT_MEMHLDI = 42,
    PMIC_MC13892_INT_LPBI = 43,
    PMIC_MC13892_INT_THWARNLI = 44,
    PMIC_MC13892_INT_THWARNHI = 45,
    PMIC_MC13892_INT_CLKI = 46,
    PMIC_MC13892_INT_SCPI = 48,
    PMIC_MC13892_INT_BATTDETBI = 54,
    PMIC_INT_MAX_ID
} PMIC_INT_ID;
```

19.6.2.5 Interrupt Functions

Table 19-3 shows the interrupt functions.

Table 19-3. Interrupt Functions

Function	Description
PmicInterruptRegister	Register the interrupt if the interrupt is to be enabled
PmicInterruptDeregister	Deregisters an interrupt
PmicInterruptHandlingComplete	Completion of a interrupt handling, enable an interrupt
PmicInterruptDisable	Disables an interrupt
PmicInterruptEnable	Reenable an interrupt

19.6.3 Register Access API

The PMIC Low Level Access API allows drivers and applications to read and write PMIC registers. There are some restrictions to prohibit drivers and application from accessing some registers. Interrupt registers is one example. The interrupt library functions are in this Low Level Access DLL.

19.6.3.1 Read Register

This function reads a PMIC register.

Prototype

PMIC_STATUS PmicRegisterRead(unsigned char index, UINT32* reg);

Parameters

index [in] register index

reg [out] The contents of the register

Return Value Status code

19.6.3.2 Write Register

This function writes a PMIC register.

Prototype

PMIC_STATUS PmicRegisterWrite(unsigned char index, UINT32 reg, UINT32 mask);

Parameters

index [in] register index

reg [in] data to be written

mask [in] bitmap mask to indicate which bits in parameter reg should be written to PMIC register

Return Value Status code

19.6.4 Power Control Reference

19.6.4.1 Power Control Function

This section provides information about MC13892 power control module. The API MC13892 Power control module can be accessed using the functions shown in Table 19-4.

Table 19-4. Power Control Functions

Function	Usage
PmicPwrctrlSetPowerCutTimer	Set the power cut timer duration
PmicPwrctrlGetPowerCutTimer	Get the power cut timer duration
PmicPwrctrlEnablePowerCut	Enable the power cut
PmicPwrctrlDisablePowerCut	Disable the power cut
PmicPwrctrlSetPowerCutCounter	Set the power cut counter
PmicPwrctrlGetPowerCutCounter	Get the power cut counter
PmicPwrctrlSetPowerCutMaxCounter	Set the maximum number of power cut counter
PmicPwrctrlGetPowerCutMaxCounter	Get the setting of maximum power cut counter
PmicPwrctrlEnableCounter	Enable the power counter
PmicPwrctrlDisableCounter	Disable the power counter
PmicPwrctrlEnableClk32kMCU	Enable the CLK32KMCU
PmicPwrctrlDisableClk32kMCU	Disable the CLK32KMCU
PmicPwrctrlEnableDRM	Set Keeps VSRTC and CLK32KMCU on for all states
PmicPwrctrlDisableDRM	Disable Keeps VSRTC and CLK32KMCU on for all states
PmicPwrctrlEnableUSEROFFCLK	Enable Keeps VSRTC and CLK32KMCU during user off
PmicPwrctrlDisableUSEROFFCLK	Disable VSRTC and CLK32KMCU during user off
PmicPwrctrlEnablePCUTEXPB	E nable PCUTEXPB=1 at a startup event
PmicPwrctrlDisablePCUTEXPB	Disable PCUTEXPB=1 at a startup event
PmicPwrctrlEnableUserOffModeWhenDelay	Place the phone in User Off Mode after a delay
PmicPwrctrlDisableUserOffModeWhenDelay	Set not to place the phone in User Off Mode after a delay
PmicPwrctrlEnableWarmStart	Warm start enable
PmicPwrctrlDisableWarmStart	Warm start disable
PmicPwrctrlEnablePWRONRESET	System reset on PWRON pin
PmicPwrctrlDisablePWRONRESET	Disable system reset on PWRON pin
PmicPwrctrlSetDebtime	Set debounce time on PWRON pin
PmicPwrctrlEnableSTANDBYINV	Set STANDBY is interpreted as active low
PmicPwrctrlDisableSTANDBYINV	Set disable STANDBY is interpreted as active not low
PmicPwrctrlEnableSTANDBYSECINV	Set disable STANDBYSEC is interpreted as active low

Table 19-4. Power Control Functions (continued)

Function	Usage
PmicPwrctrlDisableSTANDBYSECINV	Disable STANDBYSEC is interpreted as active not low
PmicPwrctrlEnableWDIRESET	Enable system reset through WDI
PmicPwrctrlDisableWDIRESET	Disable system reset through WDI
PmicPwrctrlSetSPIDRV	Set SPI drive strength
PmicPwrctrlGetSPIDRV	Get SPI drive strength
PmicPwrctrlSetCLK32KDRV	Set CLK32K and CLK32KMCU drive strength
PmicPwrctrlGetCLK32KDRV	Get the CLK32K and CLK32KMCU drive strength
PmicPwrctrlSetSTBYDLY	Set Standby delay
PmicPwrctrlGetSTBYDLY	Get the Standby delay
PmicPwrctrlGetMODES	Get the MODE sense
PmicPwrctrlGetI2CS	Get the I2CS mode
PmicPwrctrlGetPUMSS	Get the PUMSS mode

19.6.4.2 Power Control Data Structures

```
typedef enum _MC13892_PWRCTRL_PWRON{
    PWRON1=0,
    PWRON2,
    PWRON3,
} MC13892_PWRCTRL_PWRON;

typedef enum _MC13892_PWRCTRL_MODES{
    MODES_GROUNDED=0,
    MODES_RESEVED,
    MODES_VCOREDIG,
    MODES_VCORE,
} MC13892_PWRCTRL_MODES;

typedef enum _MC13892_PWRCTRL_I2CS{
    SPI=0,
    I2C,
} MC13892_PWRCTRL_I2CS;

typedef enum _MC13892_PWRCTRL_PUMSS{
    PUMSS_GROUNDED=0,
    PUMSS_OPEN,
    PUMSS_VCOREDIG,
    PUMSS_VCORE,
} MC13892_PWRCTRL_PUMSS;
```

19.6.5 Buck Switchers and Linear Regulators

This section provides information about control MC13892 buck switchers and linear regulators.

19.6.5.1 Functions

```
PMIC_STATUS PmicSwitchModeRegulatorOn (PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorOff (PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorGetVoltageLevel (PMIC_REGULATOR_SREG regulator,
            PMIC_REGULATOR_SREG_VOLTAGE_TYPE voltageType,PMIC_REGULATOR_SREG_VOLTAGE*voltage);
PMIC_STATUS PmicSwitchModeRegulatorSetMode (PMIC_REGULATOR_SREG
            regulator,PMIC_REGULATOR_SREG_STBY standby,PMIC_REGULATOR_SREG_MODE mode);
PMIC_STATUS PmicSwitchModeRegulatorGetMode (PMIC_REGULATOR_SREG regulator,
            PMIC_REGULATOR_SREG_STBY standby,PMIC_REGULATOR_SREG_MODE* mode);
PMIC_STATUS PmicSwitchModeRegulatorEnableSTBYDVFS (PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorDisableSTBYDVFS (PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorSetDVSSpeed (PMIC_REGULATOR_SREG regulator, UINT8dvsspeed);
PMIC_STATUS PmicSwitchModeRegulatorSetSidLevel (PMIC_REGULATOR_SREG regulator, UINT8
            hilevel,UINT8 lowlevel);
PMIC_STATUS PmicSwitchModeRegulatorGetSidLevel(PMIC_REGULATOR_SREG regulator, UINT8*
            hilevel,UINT8* lowlevel);
PMIC_STATUS PmicSwitchModeRegulatorSetPLLMF (UINT8 mf);
PMIC_STATUS PmicSwitchModeRegulatorEnableHIRANGE(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorDisableHIRANGE(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorEnableMemoryHoldMode(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorDisableMemoryHoldMode(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorEnableUserOffMode(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorDisableUserOffMode(PMIC_REGULATOR_SREG regulator);
PMIC_STATUS PmicSwitchModeRegulatorEnableSIDMode();
PMIC_STATUS PmicSwitchModeRegulatorDisableSIDMode();
PMIC_STATUS PmicSwitchModeRegulatorEnablePLL();
PMIC_STATUS PmicSwitchModeRegulatorDisablePLL();
PMIC_STATUS PmicSwitchModeRegulatorEnableSWBST();
PMIC_STATUS PmicSwitchModeRegulatorDisableSWBST();
PMIC_STATUS PmicVoltageRegulatorOn (PMIC_REGULATOR_VREG regulator);
PMIC_STATUS PmicVoltageRegulatorOff (PMIC_REGULATOR_VREG regulator);
PMIC_STATUS PmicVoltageRegulatorSetVoltageLevel (PMIC_REGULATOR_VREG regulator,
            PMIC_REGULATOR_VREG_VOLTAGE voltage);
PMIC_STATUS PmicVoltageRegulatorGetVoltageLevel (PMIC_REGULATOR_VREG regulator,
            PMIC_REGULATOR_VREG_VOLTAGE* voltage);
PMIC_STATUS PmicVoltageRegulatorSetPowerMode (PMIC_REGULATOR_VREG regulator,
            PMIC_REGULATOR_VREG_POWER_MODE powerMode);
PMIC_STATUS PmicVoltageRegulatorGetPowerMode (PMIC_REGULATOR_VREG regulator,
            PMIC_REGULATOR_VREG_POWER_MODE* powerMode);
PMIC_STATUS PmicVoltageGPOOn (MC13892_GPO_SREG gpo);
PMIC_STATUS PmicVoltageGPOOff (MC13892_GPO_SREG gpo);
```

19.6.6 Backlight and Led

This section provides information about control MC13892 backlight system and signaling LEDs.

19.6.6.1 Backlight and LED Functions

```
PMIC_STATUS PmicBacklightEnableHIMode(BACKLIGHT_CHANNEL channel);
PMIC_STATUS PmicBacklightDisableHIMode(BACKLIGHT_CHANNEL channel);
```

```
PMIC_STATUS PmicBacklightEnableRamp(BACKLIGHT_CHANNEL channel);
PMIC_STATUS PmicBacklightDisableRamp(BACKLIGHT_CHANNEL channel);
PMIC_STATUS PmicBacklightSetCurrentLevel(BACKLIGHT_CHANNEL channel, UINT8 level);
PMIC_STATUS PmicBacklightGetCurrentLevel(BACKLIGHT_CHANNEL channel, UINT8* level);
PMIC_STATUS PmicBacklightSetDutyCycle(BACKLIGHT_CHANNEL channel, UINT8 cycle);
PMIC_STATUS PmicBacklightGetDutyCycle(BACKLIGHT_CHANNEL channel, UINT8* cycle);
PMIC_STATUS PmicLEDIndicatorEnableRamp(LED_CHANNEL channel);
PMIC_STATUS PmicLEDIndicatorDisableRamp(LED_CHANNEL channel);
PMIC_STATUS PmicLEDIndicatorSetCurrentLevel(LED_CHANNEL channel, unsigned char level);
PMIC_STATUS PmicLEDIndicatorGetCurrentLevel(LED_CHANNEL channel, unsigned char* level);
PMIC_STATUS PmicLEDIndicatorSetDutyCycle(LED_CHANNEL channel, unsigned char dc);
PMIC_STATUS PmicLEDIndicatorGetDutyCycle(LED_CHANNEL channel, unsigned char* dc);
PMIC_STATUS PmicLEDIndicatorSetBlinkPeriod(LED_CHANNEL channel, unsigned char bp);
PMIC_STATUS PmicLEDIndicatorGetBlinkPeriod(LED_CHANNEL channel, unsigned char* bp);
PMIC_STATUS PmicLEDIndicatorEnableSWBST();
PMIC_STATUS PmicLEDIndicatorDisableSWBST();
```

19.6.6.2 Backlight and Led Data Structures

```
typedef enum _BACKLIGHT_CHANNEL {
    BACKLIGHT_MAIN_DISPLAY,
    BACKLIGHT_AUX_DISPLAY,
    BACKLIGHT_KEYPAD
} BACKLIGHT_CHANNEL;
typedef enum _LED_CHANNEL {
    TCLED_RED,
    TCLED_GREEN,
    TCLED_BLUE
} LED_CHANNEL;
```

19.6.7 ADC and Touch Controller

19.6.7.1 ADC and Touch Controller Function

```
PMIC_STATUS PmicADCInit(void);
PMIC_STATUS PmicADCGetSingleChannelOneSample(UINT16 channel, UINT16 * pResult);
PMIC_STATUS PmicADCGetSingleChannelEightSamples(UINT16 channel, UINT16 * pResult);
PMIC_STATUS PmicADCGetMultipleChannelsSamples(UINT16 channels, UINT16 * pResult);
PMIC_STATUS PmicADCGetHandsetCurrent(PMIC_ADC_CONVERTOR_MODE mode, UINT16 *pResult);
PMIC_STATUS PmicADCTouchRead(UINT16* x, UINT16* y);
PMIC_STATUS PmicADCTouchStandby(BOOL intEna);
void        PmicADCDeinit(void);
```

19.6.7.2 ADC and Touch Controller Data Structures

```
typedef enum _MC13892_TOUCH_MODE {
    TM_INACTIVE = 0,
    TM_INTERRUPT,
    TM_TOUCHSCREEM
} MC13892_TOUCH_MODE;
typedef MC13892_TOUCH_MODE PMIC_TOUCH_MODE;

typedef enum _PMIC_ADC_CONVERTOR_MODE
{
    ADC_8CHAN_1X = 0, // RAND = 0, 8 channels, 1 sample
```

```
    ADC_1CHAN_8X      // RAND = 1, 1 channel, reads 8 sequential values
} PMIC_ADC_CONVERTOR_MODE;
```

19.6.8 Battery Charger

This section provides information about control MC13892 battery charger system.

19.6.8.1 Battery Charger Functions

```
PMIC_STATUS PmicBatterEnableCharger(BATT_CHARGER chgr, UINT8 c_voltage, UINT8 c_current);
PMIC_STATUS PmicBatterDisableCharger(BATT_CHARGER chgr);
PMIC_STATUS PmicBatterSetCharger(BATT_CHARGER chgr, UINT8 c_voltage, UINT8 c_current);
PMIC_STATUS PmicBatterGetChargerSetting(BATT_CHARGER chgr, UINT8* c_voltage, UINT8*c_current);
PMIC_STATUS PmicBatterGetChargeCurrent(UINT16* c_current);
PMIC_STATUS PmicBatterLedControl(BOOL on);
PMIC_STATUS PmicBatterSetReverseSupply(BOOL enable);
PMIC_STATUS PmicBatterSetUnregulated(BOOL enable);
```

19.6.8.2 Battery Charger Data Structures

```
typedef enum {
    BATT_MAIN_CHGR = 0,         // Main battery charger
    BATT_CELL_CHGR,             // CoinCell battery charger
    BATT_TRCKLE_CHGR            // Trickle charger
} BATT_CHARGER;
typedef enum {
    DUAL_PATH = 0,
    SINGLE_PATH,
    SERIAL_PATH,
    DUAL_INPUT_SINGLE_PATH,
    DUAL_INPUT_SERIAL_PATH,
    DUAL_INPUT_DUAL_PATH,
    INVALID_CHARGER_MODE
}CHARGER_MODE;
```

Chapter 20
Serial Driver

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

20.1 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 20-1 provides a summary of source code location, library dependencies and other BSP information.

Table 20-1. Serial Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\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 serial port support Third Party –> BSP –> Freescale <*Target Platform*>: ARMV4I –> Device Drivers –> Serial –> UART3 serial port support
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_SERIAL_UART2 =1 BSP_SERIAL_UART3 =1

20.2 Supported Functionality

The serial port driver enables the hardware system to provide the following support:

1. Conforms to RS232 protocol standard
2. Supports RTS/CTS hardware flow control function
3. Supports parity check and optional stop bit
4. Supports power management mode full on/full off

5. Supports DMA transfer
6. Supports baud rate up to 4 Mbps

NOTE

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

20.3 Hardware Operation

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

20.3.1 Conflicts with Other Peripherals and Catalog Items

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

20.3.1.1 Conflicts with SoC Peripherals

All the pins of UART can be configured for alternate functionality (USBOTG, EIM, GPIO) using the i.MX51 IOMUX. The configuration is specified by BSP serial driver. Changing this configuration would result in a conflict and prevent proper operation of the UART.

20.3.1.2 Conflicts with Board Peripherals

20.4 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 documentation at the following location:

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

20.4.1 Registry Settings

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

20.4.2 Power Management

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

20.5 Unit Test

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

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

Note: n is COM number

The Serial Port Test assesses if the driver supports configurable device parameters such as baud rate and data bits. The test also assesses additional functionality such as COM port events, escape functions, and time-outs.

20.5.1 Unit Test Hardware

The following hardware is used for the unit test:

- i.MX51 EVK board

20.5.2 Unit Test Software

Table 20-2 lists the required software to run the unit tests.

Table 20-2. 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

20.5.3 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
```

20.5.4 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 20-3 describes the Serial Port driver test cases.

Table 20-3. 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.

Table 20-3. Serial Port Driver Test Cases (continued)

Test Case	Description
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.

20.6 Serial Driver API Reference

The 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

20.6.1 Serial PDD Functions

Table 20-4 shows a mapping of Serial PDD functions to the functions used in the serial driver.

Table 20-4. Serial PDD Functions

PDD Function Pointer	Serial Driver Function
HWInit	SerSerialInit
HWPostInit	SerPostInit
HWDeinit	SerDeinit
HWOpen	SerOpen
HWClose	SerClose
HWGetIntrType	SL_GetIntrType
HWRxIntrHandler	SL_RxIntrHandler
HWTxIntrHandler	SL_TxIntrHandler
HWModemIntrHandler	SL_ModemIntrHandler
HWLineIntrHandler	SL_LineIntrHandler
HWGetRxBufferSize	SL_GetRxBufferSize
HWPowerOff	SerPowerOff
HWPowerOn	SerPowerOn
HWClearDTR	SL_ClearDTR

Table 20-4. Serial PDD Functions (continued)

PDD Function Pointer	Serial Driver Function
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

20.6.2 Serial Driver Structures

This section explains the serial driver structures.

20.6.2.1 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;
```

Parameters

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

20.6.2.2 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;
```

Parameters

uart_info	This structure contains information about UART
fIRMode	This boolean value determines the module is FIR or serial
dwDevIndex	This static value contains the device index value which is read from registry
dwIOBase	This static value contains the I/O Base address of UART module which is read from registry
dwIOLen	This static value contains the I/O length of UART Module which is read from registry
pBaseAddress	Pointer to the start address of the UART registers mapped
cOpenCount	Contains count of the concurrent open
CommProp	Pointer to CommProp structure
pHWObj	Pointer to PDDs HWObj structure
useDMA	This boolean flag indicates if SDMA is to be used for transfers through this UART
SerialDmaReqTx	SDMA request line for Tx
SerialDmaReqRx	SDMA request line for Rx
SerialPhysTxDMABufferAddr	Physical address of Tx SDMA address
SerialPhysRxDMABufferAddr	Physical address of Rx SDMA address

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

Chapter 21
Sony/Philips Digital Interface (SPDIF) Driver

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

21.1 SPDIF Driver Summary

The SPDIF driver module (`spdifdev.dll`) provides transmitter (TX) functions as a waveform audio driver. For more information about the waveform audio driver, see the Platform Builder Help topic:

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

Table 21-1 provides the source code location, library dependencies, and other BSP information.

Table 21-1. SPDIF Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\SPDIFDEV
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\SPDIF
Driver DLL	spdifdev.dll
SDK Library	N/A
Catalog Item	Third Party > BSP > Freescale<*Target Platform*>:ARMV4I > Device Drivers > SPDIF > SPDIF
SYSGEN Dependency	SYSGEN_AUDIO
BSP Environment Variables	BSP_NOAUDIO= BSP_SPDIF=1

21.2 Supported Functionality

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

1. Conforms to the Microsoft audio driver architecture as defined for Windows Embedded CE 6.0 and all related operating systems
2. Supports Freescale hardware platforms that include the SPDIF module
3. Double-buffered DMA operations to transfer audio data between memory and the SPDIF TX FIFO
4. Two power management modes, full on and full off

5. PCM data and compressed data transmission according with IEC958 spec
6. TX function with 44.1Kbyte sample rate

21.2.1 Conflicts with Other Peripherals and Catalog Items

21.2.1.1 Conflicts with SoC Peripherals

No conflicts

21.2.1.2 Conflicts with board Peripherals

21.2.1.2.1 i.MX51 EVK Peripheral Conflicts

The SPDIF_OUT pin conflicts with the 1-Wire pin.

21.2.2 Known Issues

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

The SPDIF TX driver does not operate with 32Kbyte or 48Kbyte sample rates.

21.3 Software Operation

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

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

21.3.1 SPDIF Transmitter (TX)

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

21.3.2 Compile-Time Configuration Options

Table 21-2 shows the compile-time configuration options.

Table 21-2. SPDIF Driver Configuration Options (hwctxt.cpp)

Configuration Setting Name	Description
AUDIO_DMA_PAGE_SIZE	The size in bytes of each DMA buffer. Default is 6144 bytes.
SPDIF_SFCSR_TX_WATERMARK	The transmitter watermarks that are to be used with SPDIF TX FIFO. The default is 16.

21.3.3 Registry Settings

At least one registry key must be properly defined so that the Device Manager loads the SPDIF driver when the system is booted. The following registry keys are required in order for the Device Manager to properly load the SPDIF device driver during the normal device boot process. These registry settings should typically not be modified. If they are missing or incorrectly defined, then the SPDIF driver may not be loaded and all SPDIF functions are disabled.

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\SPDIF]
"Prefix"="WAV"
"Dll"="spdifdev.dll"
"Index"=dword:2
"Order"=dword:6
;"Priority256"=dword:99
"IClass"="{A32942B7-920C-486b-B0E6-92A702A99B35}"       ; PMCLASS_GENERIC_DEVICE
```

21.3.4 DMA Support

As indicated previously, the SPDIF driver uses the SDMA controller to transfer the digital audio data between the audio application and the TX FIFOs. This minimizes the processing required by the core and can also reduce the power consumption during SPDIF transmitting and receiving operations. This section describes the SPDIF driver DMA implementation issues and trade-offs, and the available compile-time DMA-related configuration options.

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

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

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

- Internal memory region:
  - Allows the external memory to be placed in a low power mode while the DMA data buffers are being processed to reduce system power consumption (as long as nothing else on the system requires access to external memory). Also, less power is required to access the internal RAM than to access.

  - Total size of the internal memory region is limited.
  - The limited amount of internal memory may have to be shared by multiple device drivers.
  - The entire internal memory region must be manually managed with predefined addressed ranges being reserved for each specific use.
- External memory region:
  - The total size of the external memory is typically much greater than the size of the internal memory. This provides much greater flexibility in selecting the size of the DMA data buffers.
  - There is typically no need to worry about the possible impact and memory requirements of any other device driver.
  - Memory allocation is handled using the standard Windows Embedded CE 6.0 system calls.
  - The external memory cannot be placed into a low power mode while the DMA is active.

The build configuration options such that the SPDIF driver allocates its DMA data buffers from either internal or external memory are as follows:

- Internal memory region—Set the BSP_SPDIF_DMA_BUF_ADDR macro in `bsp_cfg.h` to an address within the internal memory region. Also set BSP_SPDIF_DMA_BUF_SIZE to the total size (in bytes) for all DMA data buffers that are allocated.
- External memory region—Comment out the BSP_SPDIF_DMA_BUF_ADDR macro in `bsp_cfg.h`

The DMA buffer descriptors can also be allocated from either internal or external memory. However, in this case, the choice is made automatically through the use of the CSPDDK API, specifically DDKSdmaAllocChain(). See theChapter : Chip Support Package Driver Driver Development Kit(CSPDDK), for additional information about the DDKSdmaAllocChain() API.

21.4 Power Management

The primary method for limiting power consumption in the SPDIF driver is to gate off all clocks to the SPDIF when those clocks are not needed and set SPDIF to lower power mode. This is accomplished through the **DDKClockSetGatingMode** function call and the SPDIF related register setting. The clock gating and the disabling of the SPDIF is handled automatically within the SPDIF module and requires no additional configuration or code changes. The SPDIF driver operates correctly after resuming from the power down mode.

21.4.1 PowerUp

This function resumes an SPDIF I/O operation that was previously terminated by calling the PowerDown() API. It begins by restoring power and then it restarts the DMA transfers to complete the powerup process for the SPDIF driver. This function is intended to be called only by the Power Manager and must not block or depend on any hardware interrupts. Therefore, all required timed delays must be handled by using a polling loop instead of any of the normal wait for an event to be signalled functions. This functionality is currently handled by IOCTL_POWER_SET and the function is just a stub.

21.4.1.1 i.MX51 PowerUp Support

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

21.4.2 PowerDown

This function suspends all currently active SPDIF I/O operations just before the entire system enters the low power state. This function is intended to be called only by the Power Manager and must not block or depend on any hardware interrupts. This functionality is currently handled by IOCTL_POWER_SET and the function is just a stub.

21.4.2.1 i.MX51 Power Down Support

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

21.4.2.2 IOCTL_POWER_SET

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

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

21.5 Unit Test

21.5.1 Unit Test Hardware

Table 21-3 lists the required hardware to run the unit tests.

Table 21-3. Hardware Requirements

Requirement	Description
M-Audio Card on PC	M-Audio Card to send/receive SPDIF digital data
Audio Daughter Card	SPDIF output interface on the card

21.5.2 Unit Test Software

Table 21-4 lists the required software to run the unit tests.

Table 21-4. 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

Table 21-4. Software Requirements (continued)

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

21.5.3 Building the Unit Tests

To build the SPDIF tests, build an OS image for the desired configuration using the following steps:

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

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

21.5.4 Running the Unit Tests

The command line for running the SPDIF tests is:

```
tux –o –n –d spdiftest.dll
```

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

21.6 System Testing

In addition to running the SPDIF driver tests in the CETK, simple applications can be developed to perform various system-level tests that involve the use of the SPDIF driver. For example, a small modification can be made to WAVPLAY and WAVEREC to test the SPDIF TX and RX functions (Windows CE sample application source code located in `WINCE600\PUBLIC\COMMON\SDK\SAMPLES\AUDIO`).

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

For perform this testing, a SPDIF transmitter device which can be used to send audio data to the i.MX51 board is required, such as an M-Audio USB card (which can be connected to the PC by the USB port).

The TX path should be connected as follows:

M-Audio optical port [in] <—> line dual-optical interface <—> converter optical port [out] <—> converter coaxial port [in] <—> i.MX51 SPDIF TX coaxial port

Then Spectralab can be used capture audio data from the EVK SPDIF device.

21.7 SPDIF Driver API Reference

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

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

Chapter 22
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.

22.1 Touch Panel Driver Summary

Table 22-1 provides a summary of source code location, library dependencies, and other BSP information.

Table 22-1. Touch Panel Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	N/A
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\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.MX51 EVK :ARMV4I > Device Drivers > TOUCH > MC13892 TOUCH > for MC13892 PMIC Touch
SYSGEN Dependency	SYSGEN_TOUCH = 1
BSP Environment Variables	BSP_NOTOUCH=

22.2 Supported Functionality

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

Developing a Device Driver > Windows Embedded CE Drivers > Touch Screen Drivers

22.3 Hardware Operations

The hardware consists of a LCD Panel with a touch screen and a TI TSC2007 touch controller. The I^2C module sends control information to the TSC2007 and reads back the touch samples. More details about the I^2C can be found in Chapter 15, "Inter-Integrated Circuit (I2C) Driver."

The hardware also consists of a LCD panel with a touch screen and the MC34708. The MC34708 touch screen driver sends control commands and reads back the touch samples. More details about MC34708 driver can be found in Chapter 19, "Power Management IC (PMIC)."

22.3.1 Conflicts with SOC Peripherals

The touch driver requires a timer to provide the necessary timings between different touch samples. Therefore, EPIT2 is dedicated for the touch panel and cannot be used by any other module.

22.4 Software Operations

The touch screen driver reads user input from the touch screen hardware and converts the input to touch events. The touch screen events are then sent to the Graphics, Windowing, and Events Subsystem (GWES). The driver also converts un-calibrated coordinates to calibrated coordinates. Calibrated coordinates compensate for any hardware anomalies, such as skew or nonlinear sequences.

For the touch screen driver to work properly, it has to submit points while the user's finger or stylus is touching the touch screen. When the user's finger or stylus is removed from the screen, the driver must submit at least one final event indicating that the user's finger or stylus tip is removed. The calibrated coordinates must be reported to the nearest one-quarter of a pixel.

The 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` function 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 the calibration data.
3. Call the `TouchPanelSetCalibration` function to calculate the calibration coefficients.

22.4.1 Touch Driver Registry Settings

```
IF BSP_NOTOUCH !
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]
    "DriverName"="touch.dll"

; For double-tap default setting
[HKEY_CURRENT_USER\ControlPanel\Pen]
    "DblTapDist"=dword:18
    "DblTapTime"=dword:637
```

```
[HKEY_LOCAL_MACHINE\HARDWARE\DEVICEMAP\TOUCH]
    "MaxCalError"=dword:7
"CalibrationData"="524,523 796,244 796,808 252,809 258,233"

; For Touch Panel calibration. Note that the Windows Mobile PocketPC touch panel
; calibration is handled automatically by the welcome.exe application so a
; separate "Launch" registry key is not required. Also, Windows Mobile
; SmartPhone does not support a touch panel at all which means that this is
; not required for SmartPhone either.

[HKEY_LOCAL_MACHINE\init]
    "Launch80"="touchc.exe"
    "Depend80"=hex:14,00, 1e,00
ENDIF    ; BSP_NOTOUCH !
```

22.5 Unit Tests

This section explains the unit tests.

22.5.1 Unit Test Hardware

Table 22-2 lists the hardware required to run the unit tests.

Table 22-2. Hardware Requirements

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

22.5.2 Unit Test Software

Table 22-3 lists the software required to run the unit tests.

Table 22-3. Software Requirements

Requirement	Description
Tux.exe	Tux test harness, which is needed for executing the test
Kato.dll	Kato logging engine, which is required for logging test data
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 works after the CETK Touch Panel Test. This is a known MSFT CETK issue. In the MSFT online help it is mentioned that when the test is complete, the OS does not regain control of the touch panel. The touch panel should be reset 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 a arc is drawn. This is also a known MSFT CETK issue. All these points are captured.

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.

22.5.3 Running the Touch Panel Tests

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

```
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

22.6 Touch Panel API Reference

The complete API reference is available in the documentation at:

Developing a Device Driver > Windows Embedded CE Drivers > Touch Screen Drivers > Touch Screen Driver Reference

Chapter 23
TV Encoder (TVE)

The TV Encoder (TVE) is designed to provide direct connection between an Application Processor (AP) and a TV set via analog interfaces. The TVEv2 supports both Standard Definition and High Definition (HD) television standards. The TVE module receives display content input from the Image Processing Unit v3 (IPUv3). The TVEv2 functionality includes support for the NTSC, PAL, 720P60, 720P50, 1080I30 and 1080I25 standards, and various output formats.

23.1 TVE Summary

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

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\TVEV2
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SoC*>\TVEV2
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\TVE
Driver DLL	tve.dll
SDK Library	tvesdk_mx51_fsl_v2.lib
Catalog Items	Third Party –> BSP –> Freescale <*Target Platform*>: ARMV4I –> Device Drivers–> Display –> Display Port1–> TVE Output Support
SYSGEN Dependency	SYSGEN_DDRAW=1
BSP Environment Variables	BSP_TVE = 1 for TV Encoder

23.2 Supported Functionality

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

1. NTSC, PAL, 720P60, 720P50, 1080I30 and 1080I25 television standards in TVEv2.
2. YCrCb 4:2:2 input format support.
3. Dynamic switching of the LCD device (Dumb LCD/Smart LCD) and the TV device (NTSC/ PAL/ 720P/1080I). (Note: Directly dynamic switching between a TV mode to a TV mode will be supported in the next IPUV3/TVEv2 drivers release)
4. Composite video (CVBS) TV output mode.
5. Component (YPrPb) TV output mode.

6. S-Video TV output mode.(i.MX51)

NOTE:

- When using the TVE driver to drive display data to a TV, some WinCE icons on the home screen may appear partly outside of the TV overscan. TV overscan is dependent on the TV, so the adjustment to the screen position must be made using TV controls.
- For the i.MX51 TO2 board with the component output, if it shows some noise signals when playing a video clip, please try to lose a little bit the green cable connection of the component cables.

23.3 Hardware Operation

23.3.1 Conflicts with other On-Chip Peripherals

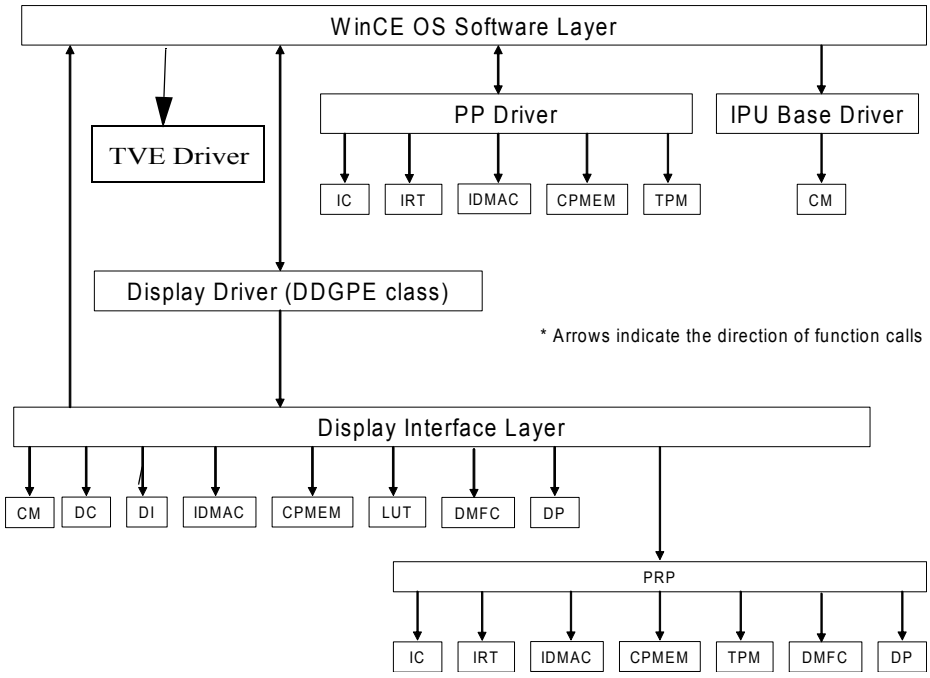
23.3.1.1 i.MX51 Peripheral Conflicts

No conflicts.

23.4 Software Operation

23.4.1 Software Architecture

The following block diagram shows the relationship between the TVE driver and the Display driver:



The TVE driver is controlled exclusively by the display driver. The Display Interface Layer makes stream interface calls to access the TVE stream driver and control TVE processing functionality.

23.4.2 Communicating with the TVE

The TVE is a stream interface driver, and is accessed through SDK APIs provided by the TVE driver. To communicate using the TVE, a handle to the device must first be obtained using the **TVEOpenHandle** function. Subsequent commands to the device are issued using various APIs supported by this driver. For more information on these APIs, please see the TVE driver API reference section.

The following code example shows how to use the TVE driver.

```
HANDLE hTVE = NULL;
PTVEOutputStdInfo pOutputStdData = (PTVEOutputStdInfo)malloc(sizeof(PTVEOutputStdInfo));
hTVE = TVEOpenHandle();
if (pOutputStdData == NULL)
{
   ERRORMSG(1, (TEXT("%s: Alloc memory for TVE parameter settings failed\r\n"),
   __WFUNCTION__));
}
// Set NTSC as a TVE output standard type
pOutputStdData->iTVOutputStd = (UINT16) TV_STAND_NTSC;
TVESetOutputStdType(hTVE, pOutputStdData);
TVECloseHandle(hTVE);
free(pOutputStdData);
```

In addition, we have extended the display driver to support TV. When switching to a TV mode, a framework for accessing the display driver through the Graphics Device Interface (GDI), DirectDraw and

an escape code mechanism are still supported. For accessing these Microsoft API interfaces, please refer to the display chapter for details.

23.4.3 Configuring the TVE

The TVE configuration is based on the **PanelType** registry key, which is described in the **TVE Registry Settings** section below. The **PanelType** registry key indicates the TV panel that is being used. The following TV output modes are supported by the TVE driver:

- TVE NTSC SD 480i at 60MHz panel.
- TVE PAL SD 576i at 50MHz panel.
- TVE HD 720P60 at 60MHz panel
- TVE HD 720P50 at 50MHz panel
- TVE HD 1080I30 at 30MHz panel
- TVE HD 1080I25 at 25MHz panel
- Composite output mode.
- Componment (YPrPb) output mode.
- S-Video output mode (i.MX51 only).

23.4.3.1 Rotation Support

The DirectDraw display driver and the TVE driver may be configured to allow screen rotation, through a parameter in the bsp_cfg.h file. If the BSP_DIRECTDRAW_SUPPORT_ROTATION parameter is set to TRUE, the DirectDraw display driver and the TVE driver will support rotation. If it is set to FALSE, it will not support rotation in the DirectDraw display driver and the TVE driver. If support for screen rotation is disabled, the TVE driver will not function properly.

23.4.3.2 TVE Registry Settings

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

If the TVE Output Support catalog item is included in the OS image, the following default registry keys are included:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\TVE]
        "Prefix"="TVE"
        "Dll"="tve.dll"
        "Order"=dword:9
        "Index"=dword:1

[HKEY_LOCAL_MACHINE\Drivers\Display\DDIPU\DI1]
        "PanelType"=dword:4        ; DISPLAY_PANEL_TV_NTSC_SD_480I60
        "TVOutputMode"=dword:6     ; TVE Component YPrPb for both SDTV and HDTV
        "DualDevice"=dword:1       ; 1 - TRUE; 0 - FALSE
        "EnableOnBoot"=dword:0     ; FALSE
```

The PanelType settings might be configured as follows:

- 4 is for DISPLAY_PANEL_TV_NTSC_SD_480I60
- 5 is for DISPLAY_PANEL_TV_PAL_SD_576I50
- 6 is for DISPLAY_PANEL_TV_HD_720P60
- 7 is for DISPLAY_PANEL_TV_HD_720P50
- 8 is for DISPLAY_PANEL_TV_HD_1080I30
- 9 is for DISPLAY_PANEL_TV_HD_1080I25

The TVOutputMode settings might be configured as follows:

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

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

23.4.3.3 i.MX51-Specific Configuration Settings

N/A

23.4.4 Power Management

The TVE driver consumes power primarily through the operation of the TVE module. The TVE power management implementation consists of the TVE clock gating, the PMIC power gating and IPUv3 sub-modules power management.

When the display output mode is switched to the TV mode, a TVE SDK API TVEEnable() function is called by the display driver to enable the TVE module and clocks. When the display output mode is switched to the LCD mode, a TVE SDK API TVEDisable() function is called by the display driver to disable the TVE module and clocks. Please refer to the TVE Driver API Reference section for details.

The TVE power management could be configured to emerge from suspend in LCD mode or in TV mode. The default TVE power management emerges from suspend in TV mode.

The IPUv3 sub-modules power management related TVE is implemented in the Display driver. Please refer to the display chapter for more information.

23.5 Unit Test

A TVOut application is provided to test that TVE driver and TV out support work correctly.

The functionality of the display functionality when in TV mode may be tested using two of the same applications that are used to test the display driver: the DirectDraw Test and Windows Media Player video playback.

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.

23.5.1 Unit Test Hardware

An SD/HD TV set is needed to run the unit tests.

23.5.2 Unit Test Software

23.5.2.1 TVOut Application

The TVOut application provides a means for testing the TVE driver and TV Out mode. This application switches the display output mode among dumb LCD (DLCD) or smart LCD (SLCD), TV NTSC output mode, and TV PAL output mode, TV 720P60 output mode, TV 720P50 output mode, TV 1080I30 output mode and TV 1080I25 output mode.

23.5.2.2 DirectDraw Tests

The following table lists the software required to run the DirectDraw tests:

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

23.5.2.3 Windows Media Player Tests

The following table lists the software required to perform WMV playback with Windows Media Player:

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

23.5.3 Building the TVE Tests

The TVOut application comes with the i.MX51 BSP release. To build the application:

- Open the i.MX51 BSP sample solution.
- Click Build OS -> Open Release Directory to open the command prompt.
- Change the current directory to \WINCE600\SUPPORT\APP\TVOUT.
- Build the application with command “build -c”.
- The binary TVOut.exe will now be automatically copied into the release directory.

The DirectDraw tests come pre-built as part of the CETK. Ensure that you have the latest CETK suite. No steps are required to build these tests. For informaton 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.

23.5.4 Running the TVE Tests

23.5.4.1 Running the TVOut Application

In the CE target shell window, execute the following commands to dynamically switch the output display between DLCD or SLCD and NTSC or PAL or 720P60 or 720P50 or 1080I30 or 1080I25, or between NTSC and PAL directly:

- “s TVOut.exe NTSC” for switching to TV NTSC
- “s TVOut.exe PAL” for switching to TV PAL
- “s TVOut.exe DLCD” for switching to Dumb LCD
- “s TVOut.exe SLCD” for switching to Smart LCD

Please execute “s TVOut.exe“ in the CE target shell window to know detailed command usage.

23.5.4.2 Running the DirectDraw Tests

The command line for running the DirectDraw tests is:

tux –o –d ddrawtk

23.5.4.3 Running the Windows Media Player Tests

On the CE shell prompt, the command line for starting playback of a WMV test video clip in Windows Media Player is:

“s ceplayer [wmv test file]”

For example, “s ceplayer motocross_208x160_30fps.wmv”

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

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

23.6 TVE Driver API Reference

23.6.1 TVE Driver Functions

23.6.1.1 TVEOpenHandle

This API creates a handle to the TVE stream driver:

```
HANDLE TVEOpenHandle(
        void
);
```

Parameters

This API accepts no parameters.

Return Values

An open handle to the specified file indicates success. INVALID_HANDLE_VALUE indicates failure.

Remarks

A handle returned successfully from this function call is required in all subsequent calls to other TVE API functions. Use the **TVECloseHandle** function to close the handle returned by **TVEOpenHandle**.

23.6.1.2 TVECloseHandle

This API function closes a handle to the TVE driver:

```
HANDLE TVECloseHandle(
        HANDLE hTVE
);
```

Parameters

hTVE

[in] Handle to the TVE driver returned by **TVEOpenHandle** API.

Return Values

Nonzero indicates success.

Zero indicates failure.

To get extended error information, call GetLastError.

Remarks

None.

23.6.1.3 TVEEnable

This API enables the TVE driver:

```
BOOL TVEEnable(
        HANDLE hTVE
);
```

Parameters

hTVE

[in] Handle to the TVE driver returned by **TVEOpenHandle** API.

Return Values

returns TRUE if sucessful.

returns FALSE if failure.

Remarks

None.

23.6.1.4 TVEDisable

This API disables the TVE driver:

```
BOOL TVEDisable(
        HANDLE hTVE
);
```

Parameters

hTVE

[in] Handle to the TVE driver returned by **TVEOpenHandle** API.

Return Values

returns TRUE if sucessful.

returns FALSE if failure.

Remarks

None.

23.6.1.5 TVESetOutputMode

This API function sets the TVE's output mode.

```
BOOL TVESetOutputMode(
        HANDLE hTVE,
        TVEOutputModeInfo *pOutputModeInfo
);
```

Parameters

hTVE

[in] Handle to the TVE driver returned by **TVEOpenHandle** API.

**pOutputModeInfo*

[in] Pointer to a TVEOutputModeInfo struct that stores the TVE output mode parameter.

Return Values

returns TRUE if successful.

returns FALSE if failure.

Remarks

None.

23.6.1.6 TVESetOutputStdType

This API function sets the TVE's output standard type.

```
BOOL TVESetOutputStdType(
        HANDLE hTVE,
        TVEOutputStdInfo *pOutputStdInfo
);
```

Parameters

hTVE

[in] Handle to the TVE driver returned by **TVEOpenHandle** API.

**pOutputStdInfo*

[in] Pointer to a TVEOutputStdInfo struct that stores the TVE output standard type parameter (TV_STAND_NTSC, TV_STAND_PAL, TV_STAND_720P60, TV_STAND_720P50, TV_STAND_1080I30, TV_STAND_1080I25).

Return Values

returns TRUE if successful.

returns FALSE if failure.

Remarks

None.

23.6.1.7 TVESetOutputResSizeType

This API function sets the TVE's output resolution size type.

```
BOOL TVESetOutputResSizeType(
        HANDLE hTVE,
        TVEOutputResSizeType *pOutputResSizeInfo
);
```

Parameters

hTVE

[in] Handle to the TVE driver returned by **TVEOpenHandle** API.

**pOutputResSizeInfo*

[in] Pointer to a TVEOutputResSizeInfo struct that stores the TVE output resolution size type parameter (TV_720X480_D1, TV_720X576_D1, TV_1280X720_720P, TV_1920X1080_1080PI).

Return Values

returns TRUE if successful.

returns FALSE if failure.

Remarks

None.

23.6.1.8 TVEGetOutputMode

This API function gets the TVE's output mode.

```
TVE_TV_OUT_MODE TVEGetOutputMode(
        HANDLE hTVE
);
```

Parameters

hTVE

[in] Handle to the TVE driver returned by **TVEOpenHandle** API.

Return Values

returns a TVE output mode.

Remarks

None.

23.6.1.9 TVEGetOutputStdType

This API function gets the TVE's output standard type.

```
TVE_TV_STAND TVEGetOutputStdType(
        HANDLE hTVE
);
```

Parameters

hTVE

[in] Handle to the TVE driver returned by **TVEOpenHandle** API.

Return Values

returns a TVE output standard type.

Remarks

None.

23.6.1.10 TVEGetOutputResSizeType

This API function gets the TVE's output resolution size type.

```
TVE_TV_RES_SIZE TVEGetOutputResSizeType(
        HANDLE hTVE
);
```

Parameters

hTVE

[in] Handle to the TVE driver returned by **TVEOpenHandle** API.

Return Values

returns a TVE output resolution size type.

Remarks

None.

23.6.2 TVE Driver Enumerations

23.6.2.1 TVE_TV_STAND

Enumeration of TVE output standard type.

```
typedef enum
{
    TV_STAND_NTSC = 0,
    TV_STAND_PALM,
    TV_STAND_PALN, // Combination PLAN
    TV_STAND_PAL, // Normal PAL (B,D,G,H,I)
    TV_STAND_720P60,
    TV_STAND_720P50,
    TV_STAND_720P30,
    TV_STAND_720P25,
    TV_STAND_720P24,
    TV_STAND_1080I30,
    TV_STAND_1080I25,
    TV_STAND_1035I30,
    TV_STAND_1080P30,
    TV_STAND_1080P25,
    TV_STAND_1080P24
} TVE_TV_STAND;
```

23.6.2.2 TVE_TV_RES_SIZE

Enumeration of TVE output resolution size.

```
typedef enum
{
    TV_720X480_D1 = 0, // resolution type for SD 720x480 D1
    TV_720X576_D1 = 1, // resolution type for SD 720x576 D1
    TV_1280X720_720P = 2, // resolution type for HD 1280x720 Progress/Interlace
    TV_1920X1080_1080PI = 3, // resolution type for HD 1920x1080 Progress/Interlace
    TV_1920X1035_1035I = 4, // resolution type for HD 1920x1035 Interlace
} TVE_TV_RES_SIZE;
```

23.6.2.3 TVE_TV_OUT_MODE

Enumeration of TVE output mode.

```
typedef enum
{
    TV_OUT_DISABLE = 0,                    // 0: TVE Standby
    TV_OUT_COMPOSITE_CH0,                  // 1: TVE Composite on Channel #0
    TV_OUT_COMPOSITE_CH2,                  // 2: TVE Composite on Channel #2
    TV_OUT_COMPOSITE_CH0_CH2,              // 3: TVE Composite on Channel #0 and #2
    TV_OUT_SVIDEO_CH0_CH1,                 // 4: TVE SVideo on Channel #0 and #1
    TV_OUT_SVIDEO_CH0_CH1_COMPOSITE_CH2,   // 5: TVE S-video on Ch#0 and #1 & Composite on Ch#2
    TV_OUT_COMPONENT_YPRPB,                // 6: TVE Component YPbPb on Channel #0, #1, and #2
    TV_OUT_COMPONENT_RGB,                  // 7: TVE Component RGB on Channel #0, #1, and #2
} TVE_TV_OUT_MODE;
```

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	iMX51-EVK
Target SOC	MX51_FSL_V2
Common SOC	COMMON_FSL_V2
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\USBD ..\PLATFORM\COMMON\SRC\SOC\<*Common Soc*>\ms\USBFN
CSP Static Library	usb_usbfn_<*Target SOC*>.lib usb_usbfn_os_<*Target SOC*>.lib usb_ufnmddbase_<*Common Soc*>.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBD
Import Library	N/A
Driver DLL	usbfn.dll

Table 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)	iMX51-EVK
Target SOC (TGTSOC)	MX51_FSL_V2PDK1_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*>.lib usbh_ehcdpdd_<*Common SOC*>.lib usbh_usb2com_<*Common SOC*>.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBH\HSOTG
Import Library	N/A
Driver DLL	hcd_hsotg.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: 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)	iMX51-EVK
Target SOC (TGTSOC)	MX51_FSL_V2_
CSP Driver Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\USBXVR
CSP Static Library	usb_xvc_<*Target SOC*>.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

24.2.1 HS Host1 Driver Summary

Table 24-4 provides a summary of source code location, library dependencies and other BSP information for the HS host driver.

Table 24-4. HS Host1 Driver Summary

Driver Attribute	Definition
Target Platform (TGTPLAT)	iMX51-EVK
Target SOC (TGTSOC)	MX51_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*>.lib usbh_ehcdpdd_<*Common SOC*>.lib usbh_usb2com_<*Common SOC*>.lib
Platform Driver Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\USBH\HSH1
Import Library	N/A

Table 24-4. HS Host1 Driver Summary (continued)

Driver DLL	hcd_hsh1.dll
Catalog Item	Third Party > BSP > Freescale <*Target Platform*>: ARMV4I > Device Drivers > USB Devices > USB High Speed Host1 To support high speed host, choose .. >High Speed Host1
SYSGEN Dependency	SYSGEN_USB=1
BSP Environment Variable	BSP_NOUSB= BSP_USB_HSH1=1

The host driver requires a set of class drivers to be loaded. See Section 24.5.6, "Class Drivers," for more information.

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 USBOH3 module provides high performance USB OTG functionality, compliant with the USB 2.0 specification, the OTG supplement and the ULPI specification. The module consists of four independent USB cores, each with Serial and ULPI USB ports. 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

No conflicts.

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.

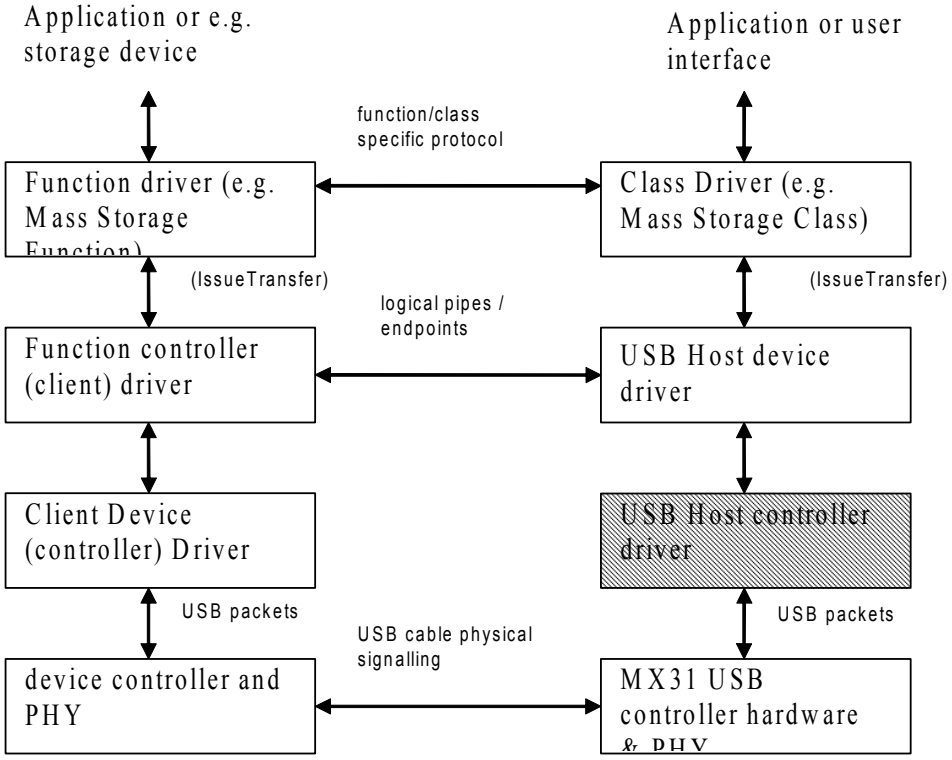


Figure 24-1. Windows USB Driver Architecture

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.MX51 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.

Table 24-5. hsotg.reg Default Values (continued)

Value	Type	Content	Description
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.

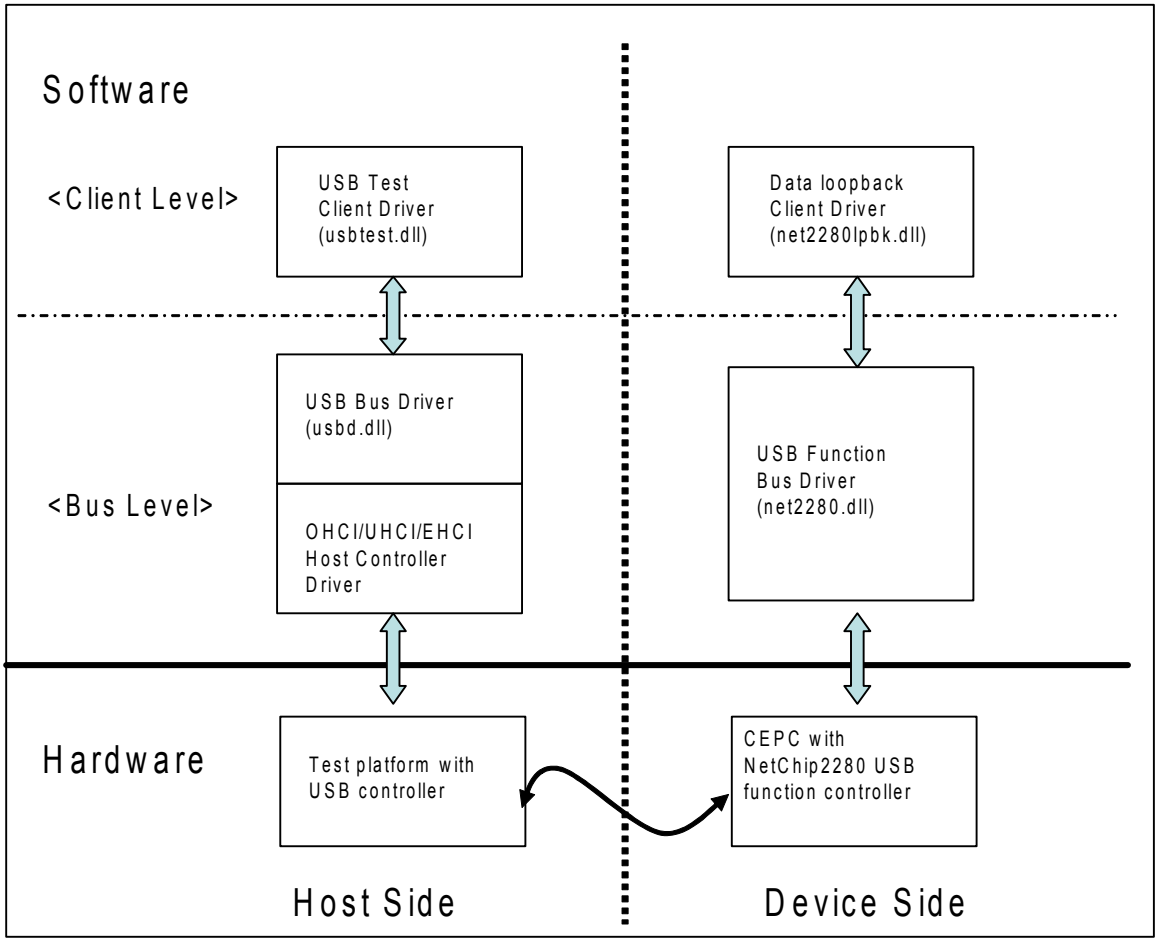


Figure 24-2. Test Setup

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 `MX51_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 - USBCoreClk OFF]
        void (*pfnUSBPowerDown)(CSP_USB_REGS *regs, BOOL *pUSBCoreClk);
        void (*pfnUSBPowerUp)(CSP_USB_REGS *regs, BOOL *pUSBCoreClk);
        // pfnUSBSetPhyPowerMode - function pointer for platform to call when they want to
suspend/resume the PHY
        // Parameter: 1) regs - USB registers
        //            2) bResume - TRUE - request to resume, FALSE - request suspend
        void (*pfnUSBSetPhyPowerMode)(CSP_USB_REGS *regs, BOOL bResume);
} BSP_USB_CALLBACK_FNS;
```

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

Table 24-22. Mutual Exclusive Driver Summary (continued)

High Speed OTG Port Pure Host only	—	—	yes	—	—	yes
High Speed OTG Port Pure Client only	—	—	yes	—	yes	—
Full Speed Host (H1)	no	no	—	—	—	—
High Speed Host (H2)	no	no	—	—	—	—

[1] yes = Required, no = Not permitted, — = Do not care

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

24.7 Application Tools for USB

An application tool is provided for the test mode and USB device class selection.

24.7.1 Application Tool for Test Mode

An application is provided in the control panel for the USB test mode. The source code is in `SUPPORT\APP\USBControlPanel`. A GUI is provided to easily select the proper USB port and corresponding test mode to complete the test.

The application interface for the test mode is shown in Figure 24-3. The left side is used for the OTG test mode and right side is used for the H1 test mode. At any time, only one port can be in test mode, either OTG or H1, but not both.

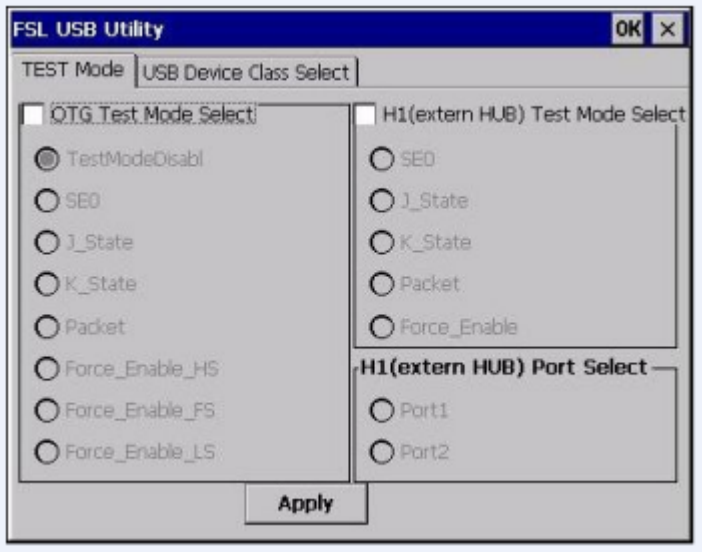


Figure 24-3. Test Mode User Interface

The test mode for the OTG port causes the DP/DM signal to change which causes undefined behavior if the USB driver is loaded. Therefore, the test mode first unloads all of the USB OTG related drivers. After the test mode, to use the USB OTG port (both host and device modes) the i.MX51 EVK must be reset.

Unlike the OTG test mode, the test mode for the H1 port includes an external hub attached to the H1 controller. The test mode for the H1 port sets the port on the external hub to test mode by sending SetPortFeature(Test_Mode) to the hub. Refer to the USB Specification 2.0 chapter 11.24.2.13 for detailed information. At any time, only one port on the external hub can be in test mode. The other port should be disabled, disconnected or in a suspended state, so make sure all of the USB devices are removed from the external hub. Only port 1 and port 2 can be used—select either one to complete the test.

24.7.2 Application Tool for USB Device Class Select

There are three types of USB device classes: ActiveSync, MSC and RNDIS. An application with a GUI is provided to switch between the three classes.

Figure 24-4 shows the tool to switch the USB device class. Make sure the OTG port is operating under the USB device mode (by connecting the mini-B connector of the USB OTG cable to the OTG port in the board) before pressing the Apply button to switch USB device class.

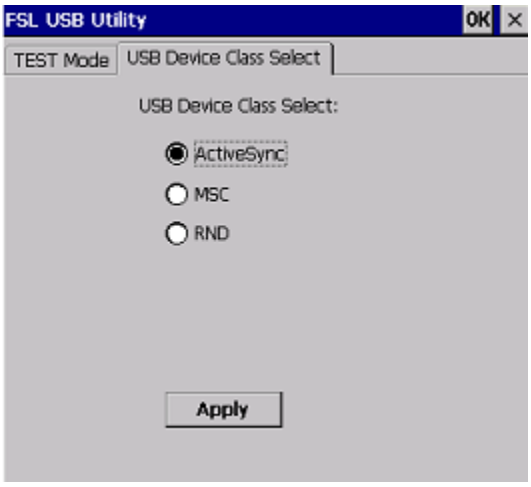


Figure 24-4. USB Device Class Switch User Interface

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	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\MS\RNE_MDD WINCE600\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\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.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 User's 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

This section explains about the software requirements for USB OTG.

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.

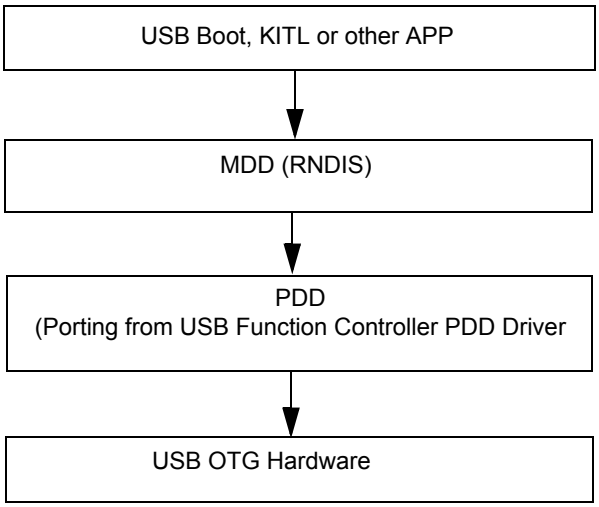


Figure 25-1. USB Boot and KITL Software Architecture Block Diagram

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 see the Platform Builder Help in the following location:

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\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

The following section explains how to perform unit tests.

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.MX51

The steps to test USB Boot and KITL are as follows:

1. Connect the target board to a PC with a USB cable and power on the board.
2. At the EBOOT menu, change the boot configuration to match the following:

```
0) IP address: 192.168.0.2
1) Subnet Mask: 255.255.255.0
3) DHCP: Disabled
6) Set MAC Address : 0-12-34-56-78-12
I) KITL Work mode: Polling
K) KITL Enable Mode: Enable
P) KITL Passive Mode: Disable
E) Select Ether Device: USB RNDIS
```

3. Press **d** to download image over USB. If this is the first time running USB Boot or KITL with the PC, the PC pops up a Found New Hardware Wizard dialog box and prompts the user to install the driver for Microsoft Windows CE RNDIS virtual adapter on the Windows PC. Refer to `WINCE600\PUBLIC\COMMON\OAK\DRIVERS\ETHDBG\RNDISMINI\HOST\howto.txt` for how to install the driver.
4. After the driver is installed successfully, the Microsoft Windows CE RNDIS virtual adapter should be displayed in Network Connections on the PC. Configure this network connection properly. Use a static IP address (such as 192.168.0.3) in the same subnet as the target board.
5. Check **Platform Builder Target** > **Connectivity** options to make sure the target device is selected. The image should be able to be download EBOOT.
6. To test USB KITL, press **r** in the EBOOT menu to enable USB KITL. After the NK starts up, the KITL operates over the USB.

Chapter 26
UUT Driver

The UUT driver is a part of manufacturing tool to provide image burning functionality in mass production stage. Greatly different with typical BSP driver, UUT has no any dedicated hardware to drive. UUT is more likely an application which uses a number of drivers like USB, SD or NAND driver. We call UUT a driver just because it is packaged as a driver format and developed by BSP team.

UUT is so complicated and unique that we create an independent package to contain it. All the docs related to UUT usage, structure and mechanism are listed in UUT package which is released along with BSP package.We only describe BSP related feature and resource here.

26.1 UUT Driver Summary

. Table 26-1 provides a summary of source code location, library dependencies and other BSP information.

Table 26-1. Serial Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	..\PLATFORM\COMMON\SRC\SOC\COMMON_FSL_V2\UUT
SOC Specific Path	N/A
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\UUT
Driver DLL	uut.dll
SDK Library	N\A
Catalog Item	N/A
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_UUT

Different with typical BSP driver, UUT need a unique project to do building work. One can find the project named iMX51-EVK-UUT in ..\OSDesigns

26.2 Supported Functionality

The UUT driver surpport below functionalities:

1. Burning images to a storage device, including: SD card.
2. Writing files to a specified folder.

26.3 Hardware Operation

None.

26.4 Test operation

Please refer to docs in UUT package.

Chapter 27
Video Processing Unit (VPU)

The Video Processing Unit (VPU) is a multi-media video processing module. The multi-instance use case is supported by VPU API. This chapter describes the following topics:

- Brief information of VPU DLL
- API provided by Freescale which allow complete access to the full functionality of the VPU
- VPU control scheme based on the API with some practical programming issues

This document is intended for application developers who use the VPU to implement a high performance video codec and need to understand and gain access to the functionality provided by the VPU.

27.1 VPU Driver Summary

Table 27-1 provides a summary of source code location, library dependencies and other BSP information.

Table 27-1. VPU Driver Summary

Driver Attribute	Definition
Target Platform	iMX51-EVK
Target SOC	MX51_FSL_V2
SOC Common Path	N/A
SOC Specific Path	..\PLATFORM\COMMON\SRC\SOC\<*Target SOC*>\VPU
Platform Specific Path	..\PLATFORM\<*Target Platform*>\SRC\DRIVERS\CSPDDK
Driver DLL	vpu.dll
SDK Library	vpusdk_<*Target SOC*>.lib
Catalog Item	Third Party > BSP > Freescale <*Target Platform*> > Device Drivers > VPU > Video Processing Unit Support
SYSGEN Dependency	N/A
BSP Environment Variables	BSP_VPU=1

27.2 Supported Functionality

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

1. All APIs defined by Freescale
2. Interrupt mode
3. Multi-task function provided by the hardware
4. Power management using chip stop mode to power down the VPU while system suspends

5. Gates off VPU clock at any time when VPU is idle
6. Uses on chip RAM for performance-sensitive buffers, such as encode search RAM
7. Support decoding for:
   - H.264 BP/MP/HP
   - VC-1 SP/MP/AP
   - MPEG-4 SP/ASP except GMC
   - H.263 Base Profile
   - MPEG-1/2 MP@HL
   - MJPEG standards up to HD (1280×720) resolution
   - JPG up to 8192×8192
8. Support encoding for:
   - H.264 up to BP@L3.0
   - H.263 Version 2 Interactive and Streaming Wireless Profile Level 60
   - MPEG4 up to SP@L5.0
   - MJPEG Baseline profile

For detailed VPU features, refer to the *i.MX51 VPU Application Programming Interface Windows Embedded CE 6.0 Reference Manual*.

27.3 Hardware Operation

Refer to the chapter on Video Processing Unit (VPU) Chapter in the *i.MX51 Applications Processor Reference Manual* for detailed hardware operation and programming information.

27.3.1 Conflicts with Other Peripherals and Catalog Items

No conflicts.

27.4 Software Operation

27.4.1 Communicating with the VPU

The VPU software is divided into two parts: the driver and the API static library. The VPU driver is implemented as a stream interface driver and is thus accessed through the file system APIs. The static library, `VPUSDK_<Target SOC>.lib`, that wraps the file system APIs to access the VPU driver, opens the VPU driver to get a handle and calls the IOCTL codes to the driver to control the VPU hardware. Applications can easily use the APIs from the static library to control the VPU hardware regardless of the VPU stream interface driver.

27.4.2 Power Management

The VPU driver consumes power primarily through the VPU decode and encode operations. Even when the VPU is idle, the internal BIT processor consumes power. When the system enters the suspend state,

the VPU module is powered off to save power. To facilitate power management of the VPU module, the display driver implements the power management I/O Control (IOCTL) codes, such as IOCTL_POWER_CAPABILITIES, IOCTL_POWER_QUERY, IOCTL_POWER_GET and IOCTL_POWER_SET.

27.4.3 Codecs Registry Settings

The following registry keys are required to properly load the decoder drivers:

```
[HKEY_LOCAL_MACHINE\Drivers\BuiltIn\VPU]
   "Prefix"="VPU"
   "Dll"="vpu.dll"
   "Order"=dword:5

[HKEY_LOCAL_MACHINE\Drivers\VPU\Decoder]
   "UserDataBufferSize"=dword:10000
```

27.5 Unit Test

The VPU can be tested using a custom VPU test application.

27.5.1 Unit Test Hardware

Table 27-2 lists the required hardware to run the VPU application test.

Table 27-2. Hardware Requirements

Requirement	Description
DVI Panel	DVI LCD panel

27.5.2 Unit Test Software

Table 27-3 lists the required software to run the VPU application test.

Table 27-3. Software Requirements

Requirement	Description
vpu.dll	VPU stream interface driver
decdemo.exe	Decoding the bitstream date file and displaying the decoded images on the LCD
encdemo.exe	Encoding the YUV(4:2:0) file and saving the encoded stream to a file

27.5.3 Running the VPU Application Test

27.5.3.1 Decoding Test

The following items are needed to run the decoding test:

- WinCE OS image with LCD support
- CE Target Control and KITL support

- Bitstream data file

The procedure for the decoding test is as follows:

1. Change the `dec.cfg` configuration file according to the bitsteam format, image size to display, frame rate and other parameters. Detailed information is in the `readme.txt` and `dec.cfg` files.
2. Run CE Target Control Debugging command `s decdemo.exe [path]\dec.cfg`.

The decoded image should be displayed on the LCD panel.

27.5.3.2 Encoding Test

The following items are needed to run the encoding test:

- WinCE OS image with LCD support
- CE Target Control and KITL support
- YUV(4:2:0) image to be encoded

The procedure for the encoding test is as follows:

1. Change the `enc.cfg` configuration file according to the bitsteam format, size of the YUV image, frame rate and other parameters. Detailed information is in the `readme.txt` and `enc.cfg` files.
2. Run CE Target Control Debugging command `s encdemo.exe [path]\nec.cfg`.

The encoded stream should be saved to a file.

27.6 VPU Driver API Reference

The API functions are defined by Freescale and a third party IP vender. For details, refer to the *i.MX51 VPU Application Programming Interface Windows Embedded CE 6.0 Reference Manual*.

27.7 Sample Demo Application

This section describes how to build and run the custom VPU test application. The VPU decoding demo application can be found in the following locations:

`\WINCE600\SUPPORT\APP\VPU\DECTEST`

The encoding demo application can be found under

`\WINCE600\SUPPORT\APP\VPU\ENCTEST`

The demo application provides an example of how to implement a video decoder or encoder using the VPU video acceleration hardware by calling the predefined API.

27.7.1 System Requirements

In order to build and run the VPU demo application, the following requirements must be met:

- The OS image must be built with the VPU driver from the Catalog
- The OS image must include SD Host Controller drivers or storage drivers, such as ATA, NAND, SD from the Catalog to enable fast loading of test data

27.7.2 Building the WinCE Image and VPU Test Application

27.7.2.1 Building the WinCE Image

To build the image:

1. Include **Third Party** > **BSPs** > **Freescale <*Target Platform*>** > **Device Drivers** > **Video Processing Unit** on Windows CE 6.0
2. Optionally include **Third Party** > **BSP** > **Freescale <*Target Platform*>** > **Device Drivers** > **SD Host Controller (**or **Storage Drivers)** on Windows CE 6.0

27.7.2.2 Building and Running the Decoding Demo Application

To build and run the decoding demo application:

1. Click **Build** > **Open Release Directory in Build Window** on Windows CE 6.0 to open the command prompt.
2. Run command **set wincerel=1** in command prompt window.
3. Change the current path to `\WINCE600\SUPPORT\APP\VPU\DECTEST`
4. Build the application with **build -c** command.
5. Run the VPU application from the CE Target Control with the command `s dectest \release\dec.cfg` (if the `dec.cfg` file is copied to `\release` directory). Make sure the parameters set in the `dec.cfg` file are correct for the bitstream and hardware display. For detailed information, refer to the `readme.txt` and `dec.cfg` files in `\WINCE600\SUPPORT\APP\VPU\DECTEST`.

27.7.2.3 Building and Running the Encoding Demo Application

To build and run the encoding demo application:

1. Click **Build** > **Open Release Directory in Build Window** on Windows CE 6.0 to open the command prompt.
2. Run command **set wincerel=1** in command prompt window.
3. Change the current path to `\WINCE600\SUPPORT\APP\VPU\ENCTEST`
4. Build the application with **build -c** command.
5. Run the VPU application from the CE Target Control with the command `s enctest \release\enc.cfg` (if `enc.cfg` file is copied to `\release` directory). Make sure the parameters set in the enc.cfg file are correct for the bitstream and hardware display. For detailed information, refer to the `readme.txt` and `enc.cfg` files in `\WINCE600\SUPPORT\APP\VPU\ENCTEST`.