i.MX 6SoloLite Linux Reference Manual

Document Number: IMXL6SLRM
Rev L3.0.35_4.1.0, 09/2013

freescale™

Contents

Chapter 1
About This Book

1.1 Audience

This document is targeted to individuals who will port the i.MX Linux BSP to customer-specific products.

The audience is expected to have a working knowledge of the Linux 3.0 kernel internals, driver models, and i.MX processors.

1.1.1 Conventions

This document uses the following notational conventions:

- Courier monospaced type indicate commands, command parameters, code examples, and file and directory names.
- *Italic* type indicates replaceable command or function parameters.
- **Bold** type indicates function names.

1.1.2 Definitions, Acronyms, and Abbreviations

The following table defines the acronyms and abbreviations used in this document.

Definitions and Acronyms

Term	Definition
ADC	Asynchronous Display Controller
address translation	Address conversion from virtual domain to physical domain
API	Application Programming Interface
ARMÂ®	Advanced RISC Machines processor architecture

Table continues on the next page...

Term	Definition
AUDMUX	Digital audio MUX-provides a programmable interconnection for voice, audio, and synchronous data routing between host serial interfaces and peripheral serial interfaces
BCD	Binary Coded Decimal
bus	A path between several devices through data lines
bus load	The percentage of time a bus is busy
CODEC	Coder/decoder or compression/decompression algorithm-used to encode and decode (or compress and decompress) various types of data
CPU	Central Processing Unit-generic term used to describe a processing core
CRC	Cyclic Redundancy Check-Bit error protection method for data communication
CSI	Camera Sensor Interface
DFS	Dynamic Frequency Scaling
DMA	Direct Memory Access-an independent block that can initiate memory-to-memory data transfers
DPM	Dynamic Power Management
DRAM	Dynamic Random Access Memory
DVFS	Dynamic Voltage Frequency Scaling
EMI	External Memory Interface-controls all IC external memory accesses (read/write/erase/program) from all the masters in the system
Endian	Refers to byte ordering of data in memory. Little endian means that the least significant byte of the data is stored in a lower address than the most significant byte. In big endian, the order of the bytes is reversed
EPIT	Enhanced Periodic Interrupt Timer-a 32-bit set and forget timer capable of providing precise interrupts at regular intervals with minimal processor intervention
FCS	Frame Checker Sequence
FIFO	First In First Out
FIPS	Federal Information Processing Standards-United States Government technical standards published by the National Institute of Standards and Technology (NIST). NIST develops FIPS when there are compelling Federal government requirements such as for security and interoperability but no acceptable industry standards
FIPS-140	Security requirements for cryptographic modules-Federal Information Processing Standard 140-2(FIPS 140-2) is a standard that describes US Federal government requirements that IT products should meet for Sensitive, but Unclassified (SBU) use
Flash	A non-volatile storage device similar to EEPROM, where erasing can be done only in blocks or the entire chip.
Flash path	Path within ROM bootstrap pointing to an executable Flash application
Flush	Procedure to reach cache coherency. Refers to removing a data line from cache. This process includes cleaning the line, invalidating its VBR and resetting the tag valid indicator. The flush is triggered by a software command
GPIO	General Purpose Input/Output
hash	Hash values are produced to access secure data. A hash value (or simply hash), also called a message digest, is a number generated from a string of text. The hash is substantially smaller than the text itself, and is generated by a formula in such a way that it is extremely unlikely that some other text produces the same hash value.
I/O	Input/Output
ICE	In-Circuit Emulation
IP	Intellectual Property
ISR	Interrupt Service Routine

Table continues on the next page...

Term	Definition
JTAG	JTAG (IEEE Standard 1149.1) A standard specifying how to control and monitor the pins of compliant devices on a printed circuit board
Kill	Abort a memory access
KPP	KeyPad Port-16-bit peripheral used as a keypad matrix interface or as general purpose input/output (I/O)
line	Refers to a unit of information in the cache that is associated with a tag
LRU	Least Recently Used-a policy for line replacement in the cache
MMU	Memory Management Unit-a component responsible for memory protection and address translation
MPEG	Moving Picture Experts Group-an ISO committee that generates standards for digital video compression and audio. It is also the name of the algorithms used to compress moving pictures and video
MPEG standards	Several standards of compression for moving pictures and video: • MPEG-1 is optimized for CD-ROM and is the basis for MP3 • MPEG-2 is defined for broadcast video in applications such as digital television set-top boxes and DVD • MPEG-3 was merged into MPEG-2 • MPEG-4 is a standard for low-bandwidth video telephony and multimedia on the World-Wide Web
MQSPI	Multiple Queue Serial Peripheral Interface-used to perform serial programming operations necessary to configure radio subsystems and selected peripherals
NAND Flash	Flash ROM technology-NAND Flash architecture is one of two flash technologies (the other being NOR) used in memory cards such as the Compact Flash cards. NAND is best suited to flash devices requiring high capacity data storage. NAND flash devices offer storage space up to 512-Mbyte and offers faster erase, write, and read capabilities over NOR architecture
NOR Flash	See NAND Flash
PCMCIA	Personal Computer Memory Card International Association-a multi-company organization that has developed a standard for small, credit card-sized devices, called PC Cards. There are three types of PCMCIA cards that have the same rectangular size (85.6 by 54 millimeters), but different widths
physical address	The address by which the memory in the system is physically accessed
PLL	Phase Locked Loop-an electronic circuit controlling an oscillator so that it maintains a constant phase angle (a lock) on the frequency of an input, or reference, signal
RAM	Random Access Memory
RAM path	Path within ROM bootstrap leading to the downloading and the execution of a RAM application
RGB	The RGB color model is based on the additive model in which Red, Green, and Blue light are combined to create other colors. The abbreviation RGB comes from the three primary colors in additive light models
RGBA	RGBA color space stands for Red Green Blue Alpha. The alpha channel is the transparency channel, and is unique to this color space. RGBA, like RGB, is an additive color space, so the more of a color placed, the lighter the picture gets. PNG is the best known image format that uses the RGBA color space
RNGA	Random Number Generator Accelerator-a security hardware module that produces 32-bit pseudo random numbers as part of the security module
ROM	Read Only Memory
ROM bootstrap	Internal boot code encompassing the main boot flow as well as exception vectors
RTIC	Real-Time Integrity Checker-a security hardware module
SCC	SeCurity Controller-a security hardware module
SDMA	Smart Direct Memory Access
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System on a Chip
SPBA	Shared Peripheral Bus Arbiter-a three-to-one IP-Bus arbiter, with a resource-locking mechanism

Table continues on the next page...

Term	Definition
SPI	Serial Peripheral Interface-a full-duplex synchronous serial interface for connecting low-/medium-bandwidth external devices using four wires. SPI devices communicate using a master/slave relationship over two data lines and two control lines: *Also see SS, SCLK, MISO, and MOSI*
SRAM	Static Random Access Memory
SSI	Synchronous-Serial Interface-standardized interface for serial data transfer
TBD	To Be Determined
UART	Universal Asynchronous Receiver/Transmitter-asynchronous serial communication to external devices
UID	Unique ID-a field in the processor and CSF identifying a device or group of devices
USB	Universal Serial Bus-an external bus standard that supports high speed data transfers. The USB 1.1 specification supports data transfer rates of up to 12 Mb/s and USB 2.0 has a maximum transfer rate of 480 Mbps. A single USB port can be used to connect up to 127 peripheral devices, such as mice, modems, and keyboards. USB also supports Plug-and-Play installation and hot plugging
USBOTG	USB On The Go-an extension of the USB 2.0 specification for connecting peripheral devices to each other. USBOTG devices, also known as dual-role peripherals, can act as limited hosts or peripherals themselves depending on how the cables are connected to the devices, and they also can connect to a host PC
word	A group of bits comprising 32-bits

Chapter 2
Introduction

2.1 Overview

The i.MX family Linux Board Support Package (BSP) supports the Linux Operating System (OS) on the following processor:

- i.MX 6SoloLite Applications Processor

The purpose of this software package is to support Linux on the i.MX 6SoloLite family of Integrated Circuits (ICs) and their associated platforms. It provides the necessary software to interface the standard open-source Linux kernel to the i.MX hardware. The goal is to enable Freescale customers to rapidly build products based on i.MX devices that use the Linux OS.

The BSP is not a platform or product reference implementation. It does not contain all of the product-specific drivers, hardware-independent software stacks, Graphical User Interface (GUI) components, Java Virtual Machine (JVM), and applications required for a product. Some of these are made available in their original open-source form as part of the base kernel.

The BSP is not intended to be used for silicon verification. While it can play a role in this, the BSP functionality and the tests run on the BSP do not have sufficient coverage to replace traditional silicon verification test suites.

2.1.1 Software Base

The i.MX BSP is based on version 3.0.35 of the Linux kernel from the official Linux kernel web site (http://www.kernel.org). It is enhanced with the features provided by Freescale.

2.1.2 Features

Table below describes the features supported by the Linux BSP for specific platforms.

Table 2-1. Linux BSP Supported Features

Feature	Description	Chapter Source	Applicable Platform
Machine Specific Layer			
MSL	Machine Specific Layer (MSL) supports interrupts, Timer, Memory Map, GPIO/IOMUX, SPBA, SDMA. • Interrupts GIC: The linux kernel contains common ARM GIC interrupts handling code. • Timer (GPT): The General Purpose Timer (GPT) is set up to generate an interrupt as programmed to provide OS ticks. Linux facilitates timer use through various functions for timing delays, measurement, events, alarms, high resolution timer features, and so on. Linux defines the MSL timer API required for the OS-tick timer and does not expose it beyond the kernel tick implementation. • GPIO/EDIO/IOMUX: The GPIO and EDIO components in the MSL provide an abstraction layer between the various drivers and the configuration and utilization of the system, including GPIO, IOMUX, and external board I/O. The IO software module is board-specific, and resides in the MSL layer as a self-contained set of files. I/O configuration changes are centralized in the GPIO module so that changes are not required in the various drivers. • SPBA: The Shared Peripheral Bus Arbiter (SPBA) provides an arbitration mechanism among multiple masters to allow access to the shared peripherals. The SPBA implementation under MSL defines the API to allow different masters to take or release ownership of a shared peripheral.	Machine Specific Layer (MSL)	All
SDMA API	The Smart Direct Memory Access (SDMA) API driver controls the SDMA hardware. It provides an API to other drivers for transferring data between MCU, DSP and peripherals. . The SDMA controller is responsible for transferring data between the MCU memory space, peripherals, and the DSP memory space. The SDMA API allows other drivers to initialize the scripts, pass parameters and control their execution. SDMA is based on a microRISC engine that runs channel-specific scripts.	Smart Direct Memory Access (SDMA) API	i.MX 6SoloLite
Low-level PM Drivers	The low-level power management driver is responsible for implementing hardware-specific operations to meet power requirements and also to conserve power on the	Low-level Power Management (PM) Driver	i.MX 6SoloLite

Table continues on the next page...

Table 2-1. Linux BSP Supported Features (continued)

Feature	Description	Chapter Source	Applicable Platform
	development platforms. Driver implementations are often different for different platforms. It is used by the DPM layer.		
CPU Frequency Scaling	The CPU frequency scaling device driver allows the clock speed of the CPUs to be changed on the fly.	CPU Frequency Scaling (CPUFREQ) Driver	i.MX 6SoloLite
DVFS	The Dynamic Voltage Frequency Scaling (DVFS) device driver allows simple dynamic voltage frequency scaling. The frequency of the core clock domain and the voltage of the core power domain can be changed on the fly with all modules continuing their normal operations.	Dynamic Voltage Frequency Scaling (DVFS) Driver	i.MX 6SoloLite
Multimedia Drivers			
LCD	The LCD interface driver supports the Samsung LMS430xx 4.3" WQVGA LCD panel.	ELCDIF Frame Buffer Driver	i.MX 6SoloLite
EPDC	The Electrophoretic Display Controller (EPDC) is a direct-drive active matrix EPD controller designed to drive E Ink EPD panels supporting a wide variety of TFT backplanes.	Electrophoretic Display Controller (EPDC) Frame Buffer	i.MX 6SoloLite
SPDC	SPDC is a direct-drive active matrix EPD controller designed to drive Sipix panel for E-Book application. The SPDC provides control signals for the source driver and gate drivers. This IP provides a high performance, low cost solution for SiPix EPDs (Electronic Paper Display).	Sipix Display Controller (SPDC) Frame Buffer	i.MX 6SoloLite
PxP	The Pixel Pipeline (PxP) DMA-ENGINE driver provides a unique API, which are implemented as a dmaengine client that smooths over the details of different hardware offload engine implementations.	PXP DMA-ENGINE Driver	i.MX 6SoloLite
Sound Drivers			
ALSA Sound	The Advanced Linux Sound Architecture (ALSA) is a sound driver that provides ALSA and OSS compatible applications with the means to perform audio playback and recording functions. ALSA has a user-space component called ALSAlib that can extend the features of audio hardware by emulating the same in software (user space), such as resampling, software mixing, snooping, and so on. The ASoC Sound driver supports stereo CODEC playback and capture through SSI.	ALSA Sound Driver	i.MX 6SoloLite
Memory Drivers			
SPI NOR MTD	The SPI NOR MTD driver provides the support to the Atmel data Flash using the SPI interface.	SPI NOR Flash Memory Technology Device (MTD) Driver	i.MX 6SoloLite
Input Device Drivers			
Networking Drivers			
ENET	The ENET Driver performs the full set of IEEE 802.3/ Ethernet CSMA/CD media access control and channel interface functions. The FEC requires an external interface adaptor and transceiver function to complete	Fast Ethernet Controller (FEC) Driver	i.MX 6SoloLite

Table continues on the next page...

Table 2-1. Linux BSP Supported Features (continued)

Feature	Description	Chapter Source	Applicable Platform
	the interface to the Ethernet media. It supports half or full-duplex operation on 10M\100M related Ethernet networks.		
Bus Drivers			
I^{2}C	The I2C bus driver is a low-level interface that is used to interface with the I2C bus. This driver is invoked by the I2C chip driver; it is not exposed to the user space. The standard Linux kernel contains a core I2C module that is used by the chip driver to access the bus driver to transfer data over the I2C bus. This bus driver supports: • Compatibility with the I2C bus standard • Bit rates up to 400 Kbps • Standard I2C master mode • Power management features by suspending and resuming I2C.	Inter-IC (I2C) Driver	i.MX 6SoloLite
CSPI	The low-level Enhanced Configurable Serial Peripheral Interface (ECSPI) driver interfaces a custom, kernel-space API to both ECSPI modules. It supports the following features: • Interrupt-driven transmit/receive of SPI frames • Multi-client management • Priority management between clients • SPI device configuration per client	Enhanced Configurable Serial Peripheral Interface (ECSPI) Driver	i.MX 6SoloLite
MMC/SD/SDIO - uSDHC	The MMC/SD/SDIO Host driver implements the standard Linux driver interface to eSDHC.	MMC/SD/SDIO Host Driver	i.MX 6SoloLite
UART Drivers			
MXC UART	The Universal Asynchronous Receiver/Transmitter (UART) driver interfaces the Linux serial driver API to all of the UART ports. A kernel configuration parameter gives the user the ability to choose the UART driver and also to choose whether the UART should be used as the system console.	Universal Asynchronous Receiver/ Transmitter (UART) Driver	i.MX 6SoloLite
General Drivers			
USB	The USB driver implements a standard Linux driver interface to the ARC USB-OTG controller.	ARC USB Driver	i.MX 6SoloLite
WatchDog	The Watchdog Timer module protects against system failures by providing an escape from unexpected hang or infinite loop situations or programming errors. This WDOG implements the following features: • Generates a reset signal if it is enabled but not serviced within a predefined time-out value • Does not generate a reset signal if it is serviced within a predefined time-out value	Watchdog (WDOG) Driver	i.MX 6SoloLite
MXC PWM driver	The MXC PWM driver provides the interfaces to access MXC PWM signals	Pulse-Width Modulator (PWM) Driver	i.MX 6SoloLite

Table continues on the next page...

Table 2-1. Linux BSP Supported Features (continued)

Feature	Description	Chapter Source	Applicable Platform
Thermal Driver	Thermal driver is a necessary driver for monitoring and protecting the SoC. The thermal driver will monitor the SoC's temperature in a certain frequency. It defines three trip points: critical, hot, and active.	Thermal Driver	i.MX 6SoloLite
OProfile	OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.	OProfile	i.MX 6SoloLite

Chapter 3
Machine Specific Layer (MSL)

3.1 Introduction

The Machine Specific Layer (MSL) provides the Linux kernel with the following machine-dependent components:

- Interrupts including GPIO and EDIO (only on certain platforms)
- Timer
- Memory map
- General Purpose Input/Output (GPIO) including IOMUX on certain platforms
- Shared Peripheral Bus Arbiter (SPBA)
- Smart Direct Memory Access (SDMA)

These modules are normally available in the following directory:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6 for i.MX 6 platform
```

The header files are implemented under the following directory:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/include/mach
```

The MSL layer contains not only the modules common to all the boards using the same processor, such as the interrupts and timer, but it also contains modules specific to each board, such as the memory map. The following sections describe the basic hardware and software operations and the software interfaces for MSL modules. First, the common modules, such as Interrupts and Timer are discussed. Next, the board-specific modules, such as Memory Map and General Purpose Input/Output (GPIO) (including IOMUX on some platforms) are detailed. Because of the complexity of the SDMA module, its design is explained in SDMA relevant chapter.

Each of the following sections contains an overview of the hardware operation. For more information, see the corresponding device documentation.

3.2 Interrupts (Operation)

This section explains the hardware and software operation of interrupts on the device.

3.2.1 Interrupt Hardware Operation

The Interrupt Controller controls and prioritizes a maximum of 128 internal and external interrupt sources.

Each source can be enabled or disabled by configuring the Interrupt Enable Register or using the Interrupt Enable/Disable Number Registers. When an interrupt source is enabled and the corresponding interrupt source is asserted, the Interrupt Controller asserts a normal or a fast interrupt request depending on the associated Interrupt Type Register settings.

Interrupt Controller registers can only be accessed in supervisor mode. The Interrupt Controller interrupt requests are prioritized in the following order: fast interrupts and normal interrupts for the highest priority level, then highest source number with the same priority. There are 16 normal interrupt levels for all interrupt sources, with level zero being the lowest priority. The interrupt levels are configurable through eight normal interrupt priority level registers. Those registers, along with the Normal Interrupt Mask Register, support software-controlled priority levels for normal interrupts and priority masking.

3.2.2 Interrupt Software Operation

For ARM-based processors, normal interrupt and fast interrupt are two different exception types. The exception vector addresses can be configured to start at low address (0x0) or high address (0xFFFF0000).

The ARM Linux implementation chooses the high vector address model.

The following file describes the ARM interrupt architecture.

```
<ltib_dir>/rpm/BUILD/linux/Documentation/arm/Interrupts
```

The software provides a processor-specific interrupt structure with callback functions defined in the irqchip structure and exports one initialization function, which is called during system startup.

3.2.3 Interrupt Features

The interrupt implementation supports the following features:

- Interrupt Controller interrupt disable and enable
- Functions required by the Linux interrupt architecture as defined in the standard ARM interrupt source code (mainly the <ltib_dir>/rpm/BUILD/linux/arch/arm/kernel/irq.c file)

3.2.4 Interrupt Source Code Structure

The interrupt module is implemented in the following file (located in the directory <ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc):

```
irq.c (If CONFIG_MXC_TZIC is not selected)
tzic.c (If CONFIG_MXC_TZIC is selected)
gic.c (If CONFIG_ARM_GIC is selected)
```

There are also two header files (located in the include directory specified at the beginning of this chapter):

```
hardware.h
irqs.h
```

The following table lists the source files for interrupts.

Table 3-1. Interrupt Files

File	Description
hardware.h	Register descriptions
irqs.h	Declarations for number of interrupts supported
gic.c	Actual interrupt functions for GIC modules

3.2.5 Interrupt Programming Interface

The machine-specific interrupt implementation exports a single function.

This function initializes the Interrupt Controller hardware and registers functions for interrupt enable and disable from each interrupt source.

This is done with the global structure irq_desc of type struct irqdesc. After the initialization, the interrupt can be used by the drivers through the request_irq() function to register device-specific interrupt handlers.

In addition to the native interrupt lines supported by the Interrupt Controller, the number of interrupts is also expanded to support GPIO interrupt and (on some platforms) EDIO interrupts. This allows drivers to use the standard interrupt interface supported by ARM Linux, such as the request_irq() and free_irq() functions.

3.3 Timer

The Linux kernel relies on the underlying hardware to provide support for both the system timer (which generates periodic interrupts) and the dynamic timers (to schedule events).

After the system timer interrupt occurs, it performs the following operations:

- Updates the system uptime.
- Updates the time of day.
- Reschedules a new process if the current process has exhausted its time slice.
- Runs any dynamic timers that have expired.
- Updates resource usage and processor time statistics.

The timer hardware on most i.MX platforms consists of either Enhanced Periodic Interrupt Timer (EPIT) or general purpose timer (GPT) or both. GPT is configured to generate a periodic interrupt at a certain interval (every 10 ms) and is used by the Linux kernel.

3.3.1 Timer Software Operation

The timer software implementation provides an initialization function that initializes the GPT with the proper clock source, interrupt mode and interrupt interval.

The timer then registers its interrupt service routine and starts timing. The interrupt service routine is required to service the OS for the purposes mentioned in Timer. Another function provides the time elapsed as the last timer interrupt.

3.3.2 Timer Features

The timer implementation supports the following features:

- Functions required by Linux to provide the system timer and dynamic timers.
- Generates an interrupt every 10 ms.

3.3.3 Timer Source Code Structure

The timer module is implemented in the arch/arm/plat-mxc/time.c file.

3.3.4 Timer Programming Interface

The timer module utilizes four hardware timers, to implement clock source and clock event objects.

This is done with the clocksource_mxc structure of struct clocksource type and clockevent_mxc structure of struct clockevent_device type. Both structures provide routines required for reading current timer values and scheduling the next timer event. The module implements a timer interrupt routine that services the Linux OS with timer events for the purposes mentioned in the beginning of this chapter.

3.4 Memory Map

A predefined virtual-to-physical memory map table is required for the device drivers to access to the device registers since the Linux kernel is running under the virtual address space with the Memory Management Unit (MMU) enabled.

3.4.1 Memory Map Hardware Operation

The MMU, as a part of the ARM core, provides the virtual-to-physical address mapping defined by the page table. For more information, see the *ARM Technical Reference Manual* (TRM) from ARM Limited.

3.4.2 Memory Map Software Operation

A table mapping the virtual memory to physical memory is implemented for i.MX platforms as defined in the file in <ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/mm.c .

3.4.3 Memory Map Features

The Memory Map implementation programs the Memory Map module to create the physical-to-virtual memory map for all the I/O modules.

3.4.4 Memory Map Source Code Structure

The Memory Map module implementation is in mm.c under the platform-specific MSL directory. The hardware.h header file is used to provide macros for all the I/O module physical and virtual base addresses and physical to virtual mapping macros. All of the memory map source code is in the following directory:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/include/mach
```

The following table lists the source files for the memory map.

Table 3-2. Memory Map Files

File	Description
mx6.h	Header files for the I/O module physical addresses

3.4.5 Memory Map Programming Interface

The Memory Map is implemented in the mm.c file to provide the map between physical and virtual addresses. It defines an initialization function to be called during system startup.

3.5 IOMUX

The limited number of pins of highly integrated processors can have multiple purposes.

The IOMUX module controls a pin usage so that the same pin can be configured for different purposes and can be used by different modules.

This is a common way to reduce the pin count while meeting the requirements from various customers. Platforms that do not have the IOMUX hardware module can do pin muxing through the GPIO module.

The IOMUX module provides the multiplexing control so that each pin may be configured either as a functional pin or as a GPIO pin. A functional pin can be subdivided into either a primary function or alternate functions. The pin operation is controlled by a specific hardware module. A GPIO pin, is controlled by the user through software with further configuration through the GPIO module. For example, the TXD1 pin might have the following functions:

- TXD1: internal UART1 Transmit Data. This is the primary function of this pin.
- UART2 DTR: alternate mode 3
- LCDC_CLS: alternate mode 4
- GPIO4[22]: alternate mode 5
- SLCDC_DATA[8]: alternate mode 6

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose and cannot be changed by software. Otherwise, the IOMUX module needs to be configured to serve a particular purpose that is dictated by the system (board) design.

- If the pin is connected to an external UART transceiver and therefore to be used as the UART data transmit signal, it should be configured as the primary function.
- If the pin is connected to an external Ethernet controller for interrupting the ARM core, it should be configured as GPIO input pin with interrupt enabled.

The software does not have control over what function a pin should have. The software only configures pin usage according to the system design.

3.5.1 IOMUX Hardware Operation

The following information applies only to those processors that have an IOMUX hardware module.

The IOMUX controller registers are briefly described in this section.

For detailed information, see the pin multiplexing section of the IC reference manual.

- SW_MUX_CTL: Selects the primary or alternate function of a pin, and enables loopback mode when applicable.
- SW_SELECT_INPUT: Controls pin input path. This register is only required when multiple pads drive the same internal port.
- SW_PAD_CTL: Controls pad slew rate, driver strength, pull-up/down resistance, and so on.

3.5.2 IOMUX Software Operation

The IOMUX software implementation provides an API to set up pin functions and pad features.

3.5.3 IOMUX Features

The IOMUX implementation programs the IOMUX module to configure the pins that are supported by the hardware.

3.5.4 IOMUX Source Code Structure

The following table lists the source files for the IOMUX module. The files are in the directory:

```
<ltib_dir>/rpm/BUILD/arch/arm/plat-mxc/

<ltib_dir>/rpm/BUILD/arch/arm/plat-mxc/include/mach
```

Table 3-3. IOMUX Files

File	Description
iomux-v3.c	IOMUX function implementation
iomux-mx6sl.h	Pin definitions in the iomux_pins enum

3.5.5 IOMUX Programming Interface

All the IOMUX functions required for the Linux port are implemented in the iomux-v3.c file.

3.5.6 IOMUX Control Through GPIO Module

For a multi-purpose pin, the GPIO controller provides the multiplexing control so that each pin may be configured either as a functional pin or a GPIO pin.

The operation of the functional pin, which can be subdivided into either major function or one alternate function, is controlled by a specific hardware module. If it is configured as a GPIO pin, the pin is controlled by the user through software with further configuration through the GPIO module. In addition, there are some special configurations for a GPIO pin (such as output based A_IN, B_IN, C_IN or DATA register, but input based A_OUT or B_OUT).

The following discussion applies to those platforms that control the muxing of a pin through the general purpose input/output (GPIO) module.

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose which can not be changed by software. Otherwise, the GPIO module needs to be configured properly to serve a particular purpose that is dictated with the system (board) design.

- If this pin is connected to an external UART transceiver, it should be configured as the primary function.
- If this pin is connected to an external Ethernet controller for interrupting the core, it should be configured as GPIO input pin with interrupt enabled.

The software does not have control over what function a pin should have. The software only configures a pin for that usage according to the system design.

3.5.6.1 GPIO Hardware Operation

The GPIO controller module is divided into MUX control and PULLUP control sub modules. The following sections briefly describe the hardware operation. For detailed information, refer to the relevant device documentation.

3.5.6.1.1 Muxing Control

The GPIO In Use Registers control a multiplexer in the GPIO module.

The settings in these registers choose if a pin is utilized for a peripheral function or for its GPIO function. One 32-bit general purpose register is dedicated to each GPIO port. These registers may be used for software control of IOMUX block of the GPIO.

3.5.6.1.2 PULLUP Control

The GPIO module has a PULLUP control register (PUEN) for each GPIO port to control every pin of that port.

3.5.6.2 GPIO Software Operation (general)

The GPIO software implementation provides an API to setup pin functions and pad features.

3.5.6.3 GPIO Implementation

The GPIO implementation programs the GPIO module to configure the pins that are supported by the hardware.

3.5.6.4 GPIO Source Code Structure

The GPIO module is implemented in the iomux.cgpio_mux.c file under the relevant MSL directory. The header file to define the pin names is under:

```
<ltib_dir>/rpm/BUILD/arch/arm/plat-mxc/include/mach
```

The following table lists the source files for the IOMUX.

Table 3-4. IOMUX Through GPIO Files

File	Description
iomux-mx6sl.h	Pin name definitions

3.5.6.5 GPIO Programming Interface

All the GPIO muxing functions required for the Linux port are implemented in the iomux-v3.c file.

3.6 General Purpose Input/Output(GPIO)

The GPIO module provides general-purpose pins that can be configured as either inputs or outputs.

When configured as an output, the pin state (high or low) can be controlled by writing to an internal register. When configured as an input, the pin input state can be read from an internal register.

3.6.1 GPIO Software Operation

The general purpose input/output (GPIO) module provides an API to configure the i.MX processor external pins and a central place to control the GPIO interrupts.

The GPIO utility functions should be called to configure a pin instead of directly accessing the GPIO registers. The GPIO interrupt implementation contains functions, such as the interrupt service routine (ISR) registration/un-registration and ISR dispatching once an interrupt occurs. All driver-specific GPIO setup functions should be made during device initialization at the MSL layer to provide better portability and maintainability. This GPIO interrupt is initialized automatically during the system startup.

If a pin is configured to GPIO by the IOMUX, the state of the pin should also be set because it is not initialized by a dedicated hardware module. Setting the pad pull-up, pull-down, slew rate and so on, with the pad control function may be required as well.

3.6.1.1 API for GPIO

API for GPIO lists the features supported by the GPIO implementation.

The GPIO implementation supports the following features:

- An API for registering an interrupt service routine to a GPIO interrupt. This is made possible as the number of interrupts defined by NR_IRQS is expanded to accommodate all the possible GPIO pins that are capable of generating interrupts.
- Functions to request and free an IOMUX pin. If a pin is used as GPIO, another set of request/free function calls are provided. The user should check the return value of the request calls to see if the pin has already been reserved before modifying the pin state. The free function calls should be made when the pin is not needed. See the API document for more details.
- Aligned parameter passing for both IOMUX and GPIO function calls. In this implementation the same enumeration for iomux_pins is used for both IOMUX and GPIO calls and the user does not have to figure out in which bit position a pin is located in the GPIO module.
- Minimal changes required for the public drivers such as Ethernet and UART drivers as no special GPIO function call is needed for registering an interrupt.

3.6.2 GPIO Features

This GPIO implementation supports the following features:

- Implementing the functions for accessing the GPIO hardware modules
- Provideing a way to control GPIO signal direction and GPIO interrupts

3.6.3 GPIO Module Source Code Structure

All of the GPIO module source code is at the MSL layer, in the following files, located in the directories indicated at the beginning of this chapter:

Table 3-5. GPIO Files

File	Description
iomux-mx 6sl.h	IOMUX common header file
gpio.h	GPIO public header file
gpio.c	Function implementation

3.6.4 GPIO Programming Interface 2

For more information, see the Documentation/gpio.txt under the Linux source code directory for the programming interface.

Chapter 4
Smart Direct Memory Access (SDMA) API

4.1 Overview

The Smart Direct Memory Access (SDMA) API driver controls the SDMA hardware.

It provides an API to other drivers for transferring data between MCU memory space and the peripherals. It supports the following features:

- Loading channel scripts from the MCU memory space into SDMA internal RAM
- Loading context parameters of the scripts
- Loading buffer descriptor parameters of the scripts
- Controlling execution of the scripts
- Callback mechanism at the end of script execution

4.1.1 Hardware Operation

The SDMA controller is responsible for transferring data between the MCU memory space and peripherals. It has the following features:

- Multi-channel DMA, supporting up to 32 time-division multiplexed DMA channels.
- Powered by a 16-bit Instruction-Set micro-RISC engine.
- Each channel executes specific script.
- Very fast context-switching with two-level priority based preemptive multi-tasking.
- 4-KB ROM containing startup scripts (that is, boot code) and other common utilities that can be referenced by RAM-located scripts.
- 8-KB RAM area is divided into a processor context area and a code space area used to store channel scripts that are downloaded from the system memory.

4.1.2 Software Operation

The driver provides an API for other drivers to control SDMA channels. SDMA channels run dedicated scripts according to peripheral and transfer types. The SDMA API driver is responsible for loading the scripts into SDMA memory, initializing the channel descriptors, and controlling the buffer descriptors and SDMA registers.

The table below provides a list of drivers that use SDMA and the number of SDMA physical channels used by each driver. A driver can specify the SDMA channel number that it wishes to use, which is called static channel allocation. It can also have the SDMA driver and provide a free SDMA channel for the driver to use, which is called dynamic channel allocation. For dynamic channel allocation, the list of SDMA channels is scanned from channel 32 to channel 1. Upon finding a free channel, that channel is allocated for the requested DMA transfers.

Table 4-1. SDMA Channel Usage

Driver Name	Number of SDMA Channels	SDMA Channel Used
SDMA CMD	1	Static Channel allocation-uses SDMA channels 0
SSI	2 per device	Dynamic channel allocation
UART	2 per device	Dynamic channel allocation
SPDIF	2 per device	Dynamic channel allocation
ESAI	2 per device	Dynamic channel allocation

4.1.3 Source Code Structure

The dmaengine.h (header file for SDMA API) is available in the directory /<ltib_dir>/rpm/BUILD/linux/include/linux

The following table shows the source files available in the directory /<ltib_dir>/rpm/BUILD/linux/drivers/dma

Table 4-2. SDMA API Source Files

File	Description
dmaengine.c	SDMA management routine
imx-sdma.c	SDMA implement driver

The following table shows the image files available in the directory /<ltib_dir>/rpm/BUILD/linux/firmware/imx/sdma

Table 4-3. SDMA Script Files

File	Description
sdma-mx6q-to1.bin.ihex	SDMA RAM scripts

4.1.4 Menu Configuration Options

The following Linux kernel configuration option is provided for this module. To get to this options, use the *./ltib -c* command when located in the *<ltib dir>*. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following option to enable this module:

- CONFIG_IMX_SDMA_: This is the configuration option for the SDMA API driver. In menuconfig, this option is available under DMA Engine support.

4.1.5 Programming Interface

The module implements standard DMA API. For more information on the functions implemented in the driver, refer to the API documents, which are included in the Linux documentation package. For additional information, refer to the ESAI driver.

4.1.6 Usage Example

Refer to one of the drivers, such as SPDIF driver, UART driver or SSI driver, that uses the SDMA API driver as a usage example.

Chapter 5
Electrophoretic Display Controller (EPDC) Frame Buffer Driver

5.1 Introduction

The Electrophoretic Display Controller (EPDC) is a direct-drive active matrix EPD controller designed to drive E Ink EPD panels supporting a wide variety of TFT backplanes. The EPDC framebuffer driver acts as a standard Linux frame buffer device while also supporting a set of custom API extensions, accessible from user space (via IOCTL) or another kernel module (via direct function call) in order to provide the user with access to EPD-specific functionality. The EPDC driver is abstracted from any specific E Ink panel type, providing flexibility to work with a range of E Ink panel types and specifications.

The EPDC driver supports the following features:

- Support for EPDC driver as a loadable or built-in module.
- Support for RGB565 and Y8 frame buffer formats.
- Support for full and partial EPD screen updates.
- Support for up to 256 panel-specific waveform modes.
- Support for automatic optimal waveform selection for a given update.
- Support for synchronization by waiting for a specific update request to complete.
- Support for screen updates from an alternate (overlay) buffer.
- Support for automated collision handling.
- Support for 64 simultaneous update regions.
- Support for pixel inversion in a Y8 frame buffer format.
- Support for 90, 180, and 270 degree HW-accelerated frame buffer rotation.
- Support for panning (y-direction only).
- Support for automated full and partial screen updates through the Linux fb_deferred_io mechanism.
- Support for three EPDC driver display update schemes: Snapshot, Queue, and Queue and Merge.

- Support for setting the ambient temperature through either a one-time designated API call or on a per-update basis.
- Support for user control of the delay between completing all updates and powering down the EPDC.

5.2 Hardware Operation

The detailed hardware operation of the EPDC is discussed in the i.*MX 6SoloLite Applications Processor Reference Manual.*

5.3 Software Operation

The EPDC frame buffer driver is a self-contained driver module in the Linux kernel. It consists of a standard frame buffer device API coupled with a custom EPD-specific API extension, accessible through the IOCTL interface. This combined functionality provides the user with a robust and familiar display interface while offering full control over the contents and update mode of the E Ink display.

This section covers the software operation of the EPDC driver, both through the standard frame buffer device architecture, and through the custom E Ink API extensions. Additionally, panel intialization and framebuffer formats are discussed.

5.3.1 EPDC Frame Buffer Driver Overview

The frame buffer device provides an abstraction for the graphics hardware. It represents the frame buffer video hardware and allows application software to access the graphics hardware through a well-defined interface, so that the software is not required to know anything about the low-level hardware registers. The EPDC driver supports this model with one key caveat: the contents of the frame buffer are not automatically updated to the E Ink display. Instead, a custom API function call is required to trigger an update to the E Ink display. The details of this process are explained in the EPDC Frame Buffer Driver Extensions.

The frame buffer driver is enabled by selecting the frame buffer option under the graphics parameters in the kernel configuration. To supplement the frame buffer driver, the kernel builder may also include support for fonts and a startup logo. The frame buffer device depends on the virtual terminal (VT) console to switch from serial to graphics mode. The device is accessed through special device nodes, located in the /dev directory, as /dev/fb*. fb0 is generally the primary frame buffer.

A frame buffer device is a memory device, such as /dev/mem, and it has features similar to a memory device. Users can read it, write to it, seek to some location in it, and mmap() it (the main use). The difference is that the memory that appears in the special file is not the whole memory, but the frame buffer of some video hardware.

The EPDC frame buffer driver (drivers/video/mxc/mxc_epdc_fb.c) interacts closely with the generic Linux frame buffer driver (drivers/video/fbmem.c).

For additional details on the frame buffer device, please refer to documentation in the Linux kernel found in Documentation/fb/framebuffer.txt.

5.3.2 EPDC Frame Buffer Driver Extensions

E Ink display technology, in conjunction with the EPDC, has several features that distinguish it from standard LCD-based frame buffer devices. These differences introduce the need for API extensions to the frame buffer interface. The EPDC refreshes the E Ink display asynchronously and supports partial screen updates. Therefore, the EPDC requires notification from the user when the frame buffer contents have been modified and which region needs updating. Another unique characteristic of EPDC updates to the E Ink display is the long screen update latencies (between 300-980ms), which introduces the need for a mechanism to allow the user to wait for a given screen update to complete.

The custom API extensions to the frame buffer device are accessible both from user space applications and from within kernel space. The standard device IOCTL interface provides access to the custom API for user space applications. The IOCTL extensions, along with relevant data structures and definitions, can be found in include/linux/mxcfb.h. A full description of these IOCTLs can be found in the Programming Interface section Programming Interface.

For kernel mode access to the custom API extensions, the IOCTL interface should be bypassed in favor of direct access to the underlying functions. These functions are included in include/linux/mxcfb_epdc_kernel.h, and are documented in the Programming Interface section Programming Interface.

5.3.3 EPDC Panel Configuration

The EPDC driver is designed to flexibly support E Ink panels with a variety of panel resolutions, timing parameters, and waveform modes. The EPDC driver is kept panel-agnostic through the use of an EPDC panel mode structure, mxc_epdc_fb_mode, which can be found in arch/arm/plat-mxc/include/mach/epdc.h.

```
struct mxc_epdc_fb_mode {
                    struct fb_videomode *vmode;
                    int vscan_holdoff;
                    int sdoed_width;
                    int sdoed_delay;
                    int sdoez_width;
                    int sdoez_delay;
                    int gdclk_hp_offs;
                    int gdsp_offs;
                    int gdoe_offs;
                    int gdclk_offs;
                    int num_ce;
};
```

The mxc_epdc_fb_mode structure consists of an fb_videomode structure and a set of EPD timing parameters. The fb_videomode structure defines the panel resolution and the basic timing parameters (pixel clock frequency, hsync and vsync margins) and the additional timing parameters in mxc_epdc_fb_mode define EPD-specific timing parameters, such as the source and gate driver timings. Please refer to the EPDC programming model section within the i.MX 6SoloLite Applications Processor Reference Manual for details on how E Ink panel timing parameters should be configured.

This EPDC panel mode is part of the mxc_epdc_fb_platform_data structure that is passed to the EPDC driver during driver registration.

```
struct mxc_epdc_fb_platform_data {
                    struct mxc_epdc_fb_mode *epdc_mode;
                    int num_modes;
                    void (*get_pins) (void);
                    void (*put_pins) (void);
                    void (*enable_pins) (void);
                    void (*disable_pins) (void);
};
```

In addition to the EPDC panel mode data, functions may be passed to the EPDC driver to define how to handle the EPDC pins when the EPDC driver is enabled or disabled. These functions should disable the EPDC pins for purposes of power savings.

5.3.3.1 Boot Command Line Parameters

Additional configuration for the EPDC driver is provided through boot command line parameters. The format of the command line option is as follows:

```
video=mxcepdcfb:[panel_name],bpp=16
```

The EPDC driver parses these options and tries to match panel_name to the name of video mode specified in the mxc_epdc_fb_mode panel mode structure. If no match is found, then the first panel mode provided in the platform data is used by the EPDC driver. The bpp setting from this command line sets the initial bits per pixel setting for the frame buffer. A setting of 16 selects RGB565 pixel format, while a setting of 8 selects 8-bit grayscale (Y8) format.

5.3.4 EPDC Waveform Loading

The EPDC driver requires a waveform file for proper operation. This waveform file contains the waveform information needed to generate the waveforms that drive updates to the E Ink panel. A pointer to the waveform file data is programmed into the EPDC before the first update is performed.

There are two options for selecting a waveform file:

1. Select one of the default waveform files included in this BSP and built into the kernel.
2. Use a new waveform file that is specific to the E Ink panel being used.

The waveform file is loaded by the EPDC driver using the Linux firmware APIs.

5.3.4.1 Using a Default Waveform File

The quickest and easiest way to get started using an E Ink panel and the EPDC driver is to use one of the default waveform files provided in the Linux BSP. This should enable updates to several different types of E Ink panel without a panel-specific waveform file. The drawback is that optimal quality should not be expected. Typically, using a non-panel-specific waveform file for an E Ink panel results in more ghosting artifacts and overall poorer color quality.

The following default waveform files included in the BSP reside in firmware/imx/:

- epdc_E60_V110.fw - Default waveform for the 6.0 inch V110 E Ink panel.
- epdc_E60_V220.fw - Default waveform for the 6.0 inch V220 E Ink panel (supports animation mode updates).
- epdc_E97_V110.fw - Default waveform for the 9.7 inch V110 E Ink panel.
- epdc_E060SCM.fw - Default waveform for the 6.0 inch Pearl E Ink panel (supports animation mode updates).

The EPDC driver attempts to load a waveform file with the name "imx/epdc_[panel_name].fw", where panel_name refers to the string specified in the fb_videomode `name` field. This panel_name information should be provided to the EPDC driver through the kernel command line parameters described in the preceding chapter. For example, to load the epdc_E060SCM.fw default firmware file for a Pearl panel, set the EPDC kernel command line paratmeter to the following:

```
video=mxcepdcfb:E060SCM,bpp=16
```

5.3.4.2 Using a Custom Waveform File

To ensure the optimal E Ink display quality, use a waveform file specific to E Ink panel being used. The raw waveform file type (.wbf) requires conversion to a format that can be understood and read by the EPDC. This conversion script is not included as part of the BSP, so please contact Freescale to acquire this conversion script.

Once the waveform conversion script has been run on the raw waveform file, the converted waveform file should be renamed so that the EPDC driver can find it and load it. The driver is going to search for a waveform file with the name "imx/ epdc_[panel_name].fw", where panel_name refers to the string specified in the fb_videomode `name` field. For example, if the panel is named "E60_ABCD", then the converted waveform file should be named epdc_E60_ABCD.fw.

The firmware script firmware.sh (lib/udev/firmware in the Linux root file system) contains the search path used to locate the firmware file. The default search path for firmware files is /lib/firmware;/usr/local/lib/firmware. A custom search path can be specified by modifying firmware.sh. You’ll need to create an imx directory in one of these paths and add your new epdc_[panel_name].fw file there.

NOTE

If the EPDC driver is searching for a firmware waveform file that matches the names of one of the default waveform files (see preceding chapter), it will choose the default firmware files that are built into the BSP over any firmware file that has been added in the firmware search path. Thus, if you leave the BSP so that it builds those default firmware files into the OS image, be sure to use a panel name other than those associated with the default firmware files, since those default waveform files will be preferred and selected over a new waveform file placed in the firmware search path.

5.3.5 EPDC Panel Initialization

The framebuffer driver will not typically (see note below for exceptions) go through any hardware initialization steps when the framebuffer driver module is loaded. Instead, a subsequent user mode call must be made to request that the driver initialize itself for a specific EPD panel. To initialize the EPDC hardware and E-ink panel, an FBIOPUT_VSCREENINFO ioctl call must be made, with the xres and yres fields of the fb_var_screeninfo parameter set to match the X and Y resolution of a supported E-ink panel type. To ensure that the EPDC driver receives the initialization request, the activate field of the fb_var_screeninfo parameter should be set to FB_ACTIVATE_FORCE.

NOTE

The exception is when the FB Console driver is included in the kernel. When the EPDC driver registers the framebuffer device, the FB Console driver will subsequently make an FBIOPUT_VSCREENINFO ioctl call. This will in turn initialize the EPDC panel.

5.3.6 Grayscale Framebuffer Selection

The EPDC framebuffer driver supports the use of 8-bit grayscale (Y8) and 8-bit inverted grayscale (Y8 inverted) pixel formats for the framebuffer (in addition to the more common RGB565 pixel format). In order to configure the framebuffer format as 8-bit grayscale, the application would call the FBIOPUT_VSCREENINFO framebuffer ioctl. This ioctl takes an fb_var_screeninfo pointer as a parameter. This parameter specifies the attributes of the framebuffer and allows the application to request changes to the framebuffer format. There are two key members of the fb_var_screeninfo parameter that must be set in order to request a change to 8-bit grayscale format: bits_per_pixel and grayscale. bits_per_pixel must be set to 8 and grayscale must be set to one of the 2 valid grayscale format values: GRAYSCALE_8BIT or GRAYSCALE_8BIT_INVERTED.

The following code snippet demonstrates a request to change the framebuffer to use the Y8 pixel format:

```
fb_screen_info screen_info;
screen_info.bits_per_pixel = 8;
screen_info.grayscale = GRAYSCALE_8BIT;
retval = ioctl(fd_fb0, FBIOPUT_VSCREENINFO, &screen_info);
```

5.3.7 Enabling An EPDC Splash Screen

By default, the EPDC support in U-Boot is disabled, and therefore splash screen support is also disabled. To enable splash screen support, edit the configuration file /include/configs/mx6sl_evk.h and enable the following defines:

```
#define CONFIG_SPLASH_SCREEN
#define CONFIG_MXC_EPDC
```

Once this change has been made, rebuild the U-Boot image and flash it to your SD card. Then perform the following steps to flash a waveform file to an SD card where U-Boot can find it:

1. Identify the EPDC waveform file from the Linux kernel firmware directory that is the best match for the panel you are using. For the DC2/DC3 boards, that would be the waveform file /firmware/imx/epdc_E060SCM.fw.ihex.
2. Convert the ihex firmware file to a stripped-down binary using the script ihex2bin.py. Please contact Freescale to acquire this script.

   ```
   python ihex2bin.py -I epdc_E060SCM.fw.ihex -o epdc_E060SCM_splash.bin
   ```
3. Write the firmware file to the SD card at an offset of 6MB where U-Boot will look for it.

   ```
   dd if=./epdc_E060SCM_splash.bin of=/dev/sdX bs=1024 seek=6144
   ```

5.4 Source Code Structure

Table below lists the source files associated with the EPDC driver. These files are available in the following directory:

```
drivers/video/mxc
```

Table 5-1. EPDC Driver Files

File	Description
mxc_epdc_fb.c	The EPDC frame buffer driver.
epdc_regs.h	Register definitions for the EPDC module.

Table below lists the global header files associated with the EPDC driver. These files are available in the following directory:

```
include/linux/
```

Table 5-2. EPDC Global Header Files

File	Description
mxcfb.h	Header file for the MXC framebuffer drivers
mxcfb_epdc_kernel.h	Header file for direct kernel access to the EPDC API extension

5.5 Menu Configuration Options

To get to the Linux kernel configuration options available for the EPDC module, use the command ./ltib -c when located in the <ltib dir>. On the screen displayed, select **Configure the kernel** and exit. When the next screen appears select the options to configure. The following Linux kernel configuration options are provided for the EPDC module:

- CONFIG_MXC_EINK_PANEL includes support for the Electrophoretic Display Controller. In menuconfig, this option is available under:
  - Device Drivers > Graphics Support > E-Ink Panel Framebuffer
- CONFIG_MXC_EINK_AUTO_UPDATE_MODE enables support for auto-update mode, which provides automated EPD updates through the deferred I/O framebuffer driver. This option is dependent on the MXC_EINK_PANEL option. In menuconfig, this option is available under:
  - Device Drivers > Graphics Support > E-Ink Auto-update Mode Support

NOTE

This option only enables the use of auto-update mode. Turning on auto-update mode requires an additional IOCTL call using the MXCFB_SET_AUTO_UPDATE_MODE IOCTL.

- CONFIG_FB to include frame buffer support in the Linux kernel. In menuconfig, this option is available under:
  - Device Drivers > Graphics support > Support for frame buffer devices
  - By default, this option is Y for all architectures.
- CONFIG_FB_MXC is a configuration option for the MXC Frame buffer driver. This option is dependent on the CONFIG_FB option. In menuconfig, this option is available under:
  - Device Drivers > Graphics support > MXC Framebuffer support
  - By default, this option is Y for all architectures.
- CONFIG_MXC_PXP enables support for the PxP. The PxP is required by the EPDC driver for processing (color space conversion, rotation, auto-waveform selection) framebuffer update regions. This option must be selected for the EPDC framebuffer driver to operate correctly. In menuconfig, this option is available under:
  - Device Drivers > DMA Engine support > MXC PxP support

5.6 Programming Interface

5.6.1 IOCTLs/Functions

The EPDC Frame Buffer is accessible from user space and from kernel space. A single set of functions describes the EPDC Frame Buffer driver extension. There are, however, two modes for accessing these functions. For user space access the IOCTL interface should be used. For kernel space access the functions should be called directly. For each function below both the IOCTL code and the corresponding kernel function is listed.

MXCFB_SET_WAVEFORM_MODES / mxc_epdc_fb_set_waveform_modes()

Description:

Defines a mapping for common waveform modes.

Parameters:

mxcfb_waveform_modes **modes*

Pointer to a structure containing the waveform mode values for common waveform modes. These values must be configured in order for automatic waveform mode selection to function properly.

MXCFB_SET_TEMPERATURE / mxc_epdc_fb_set_temperature

Description:

Set the temperature to be used by the EPDC driver in subsequent panel updates.

Parameters:

int*32_t temperature*

Temperature value, in degrees Celsius. Note that this temperature setting may be overridden by setting the temperature value parameter to anything other than TEMP_USE_AMBIENT when using the MXCFB_SEND_UPDATE ioctl.

MXCFB_SET_AUTO_UPDATE_MODE / mxc_epdc_fb_set_auto_update

Description:

Select between automatic and region update mode.

Parameters:

__u32 mode

In region update mode, updates must be submitted via the MXCFB_SEND_UPDATE IOCTL.

In automatic mode, updates are generated automatically by the driver by detecting pages in frame buffer memory region that have been modified.

MXCFB_SET_UPDATE_SCHEME / mxc_epdc_fb_set_upd_scheme

Description:

Select a scheme that dictates how the flow of updates within the driver.

Parameters:

__u32 scheme

Select of the following updates schemes:

UPDATE_SCHEME_SNAPSHOT - In the Snapshot update scheme, the contents of the framebuffer are immediately processed and stored in a driver-internal memory buffer. By the time the call to MXCFB_SEND_UPDATE has completed, the framebuffer region is free and can be modified without affecting the integrity of the last update. If the update frame submission is delayed due to other pending updates, the original buffer contents will be displayed when the update is finally submitted to the EPDC hardware. If the update results in a collision, the original update contents will be resubmitted when the collision has cleared.

UPDATE_SCHEME_QUEUE - The Queue update scheme uses a work queue to aynchronously handle the processing and submission of all updates. When an update is submitted via MXCFB_SEND_UPDATE, the update is added to the queue and then processed in order as EPDC hardware resources become available. As a result, the framebuffer contents processed and updated are not guaranteed to reflect what was present in the framebuffer when the update was sent to the driver.

UPDATE_SCHEME_QUEUE_AND_MERGE - The Queue and Merge scheme uses the queueing concept from the Queue scheme, but adds a merging step. This means that, before an update is processed in the work queue, it is first compared with other pending updates. If any update matches the mode and flags of the current update and also overlaps the update region of the current update, then that update will be merged with the current update. After attempting to merge all pending updates, the final merged update will be processed and submitted.

MXCFB_SEND_UPDATE / mxc_epdc_fb_send_update

Description:

Request a region of the frame buffer be updated to the display.

Parameters:

mxcfb_update_data **upd_data*

Pointer to a structure defining the region of the frame buffer, waveform mode, and collision mode for the current update. This structure also includes a flags field to select from one of the following update options:

EPDC_FLAG_ENABLE_INVERSION - Enables inversion of all pixels in the update region.

EPDC_FLAG_FORCE_MONOCHROME - Enables full black/white posterization of all pixels in the update region.

EPDC_FLAG_USE_ALT_BUFFER - Enables updating from an alternate (non-framebuffer) memory buffer.

If enabled, the final *upd_data* parameter includes detailed configuration information for the alternate memory buffer.

MXCFB_WAIT_FOR_UPDATE_COMPLETE / mxc_epdc_fb_wait_update_complete

Description:

Block and wait for a previous update request to complete.

Parameters:

mxfb_update_marker_data *marker_data*

The update_marker value used to identify a particular update (passed as a parameter in MXCFB_SEND_UPDATE IOCTL call) should be re-used here to wait for the update to complete. If the update was a collision test update, the collision_test variable will return the result indicating whether a collision occurred.

MXCFB_SET_PWRDOWN_DELAY / mxc_epdc_fb_set_pwrdown_delay

Description:

Set the delay between the completion of all updates in the driver and when the driver should power down the EPDC and the E Ink display power supplies.

Parameters:

int32_t *delay*

Input delay value in milliseconds. To disable EPDC power down altogether, use FB_POWERDOWN_DISABLE (defined below).

MXCFB_GET_PWRDOWN_DELAY / mxc_epdc_fb_get_pwrdown_delay

Description:

Retrieve the driver's current power down delay value.

Parameters:

int32_t *delay*

Output delay value in milliseconds.

5.6.2 Structures and Defines

```
#define GRAYSCALE_8BIT                                                         0x1
#define GRAYSCALE_8BIT_INVERTED                                                0x2


#define AUTO_UPDATE_MODE_REGION_MODE                                           0

#define AUTO_UPDATE_MODE_AUTOMATIC_MODE                                        1

#define UPDATE_SCHEME_SNAPSHOT                                                 0
#define UPDATE_SCHEME_QUEUE                                                    1
#define UPDATE_SCHEME_QUEUE_AND_MERGE                                          2

#define UPDATE_MODE_PARTIAL                                                    0x0
#define UPDATE_MODE_FULL                                                       0x1

#define WAVEFORM_MODE_AUTO                                                     257

#define TEMP_USE_AMBIENT                                                       0x1000

#define EPDC_FLAG_ENABLE_INVERSION                                             0x01
#define EPDC_FLAG_FORCE_MONOCHROME                                             0x02
#define EPDC_FLAG_USE_ALT_BUFFER                                               0x100
#define EPDC_FLAG_TEST_COLLISION 0x200

#define FB_POWERDOWN_DISABLE                                                   -1

struct mxcfb_rect {
    __u32 left; /* Starting X coordinate for update region */
    __u32 top; /* Starting Y coordinate for update region */
    __u32 width; /* Width of update region */
    __u32 height; /* Height of update region */
};

struct mxcfb_waveform_modes {
    int mode_init; /* INIT waveform mode */
    int mode_du; /* DU waveform mode */
    int mode_gc4; /* GC4 waveform mode */
    int mode_gc8; /* GC8 waveform mode */
    int mode_gc16; /* GC16 waveform mode */
    int mode_gc32; /* GC32 waveform mode */
};

struct mxcfb_alt_buffer_data {
    __u32 phys_addr; /* physical address of alternate image buffer */
    __u32 width; /* width of entire buffer */
    __u32 height; /* height of entire buffer */
    struct mxcfb_rect alt_update_region; /* region within buffer to update */
};

struct mxcfb_update_data {
    struct mxcfb_rect update_region; /* Rectangular update region bounds */
    __u32 waveform_mode; /* Waveform mode for update */
    __u32 update_mode; /* Update mode selection (partial/full) */
    __u32 update_marker; /* Marker used when waiting for completion */
    int temp; /* Temperature in Celsius */
    uint flags; /* Select options for the current update */
    struct mxcfb_alt_buffer_data alt_buffer_data; /* Alternate buffer data */
};
 struct mxcfb_update_marker_data { __u32 update_marker; __u32 collision_test; };
```

Chapter 6
Sipix Display Controller (SPDC) Frame Buffer Driver

6.1 Introduction

Electrophoretic Display Timing Controller (SPDC) is a direct-drive active matrix EPD controller designed to drive Sipix panel for E-Book application. The SPDC provides control signals for the source driver and gate drivers. This IP provides a high performance, low cost solution for SiPix EPDs (Electronic Paper Display). Partial update and concurrent display updates resulting in high responsive screen changes are also implemented for these applications. The SPDC module co-works in conjunction with the ePXP IP module to form a complete display processing solution (such as rotation and flip function etc).

The SPDC driver supports the following features:

- Support for SPDC driver as a loadable or built-in module.
- Support for RGB565 and Y4 frame buffer formats.
- Support 800x600 resolution.
- Support for full and partial EPD screen updates.
- Support for automatic optimal waveform selection for a given update.
- Support for synchronization by waiting for a specific update request to complete.
- Support for screen updates from an alternate (overlay) buffer.
- Support for 90, 180, and 270 degree HW-accelerated frame buffer rotation.
- Support for panning (y-direction only).
- Support for automated full and partial screen updates through the Linux fb_deferred_io mechanism.
- Support for three SPDC driver display update schemes: Snapshot, Queue, and Queue and Merge.
- Support for setting the ambient temperature through either a one-time designated API call or on a per-update basis.
- Support for user control of the delay between completing all updates and powering down the SPDC.

6.2 Hardware Operation

The detailed hardware operation of the SPDC is discussed in the *i.MX 6SoloLite Multimedia Applications Processor Reference Manual (i.MX 6SL RM)*.

6.3 Software Operation

The SPDC frame buffer driver is a self-contained driver module in the Linux kernel. It consists of a standard frame buffer device API coupled with a custom EPD-specific API extension which is accessible through the IOCTL interface. The combined functionality provides the user with a robust and familiar display interface while offering full control over the contents and update mode of the Sipix display.

This section covers the software operation of the SPDC driver, both through the standard frame buffer device architecture, and through the custom Sipix API extensions. Additionally, panel intialization and framebuffer formats are discussed.

6.3.1 SPDC Frame Buffer Driver Overview

The frame buffer device provides an abstraction for the graphics hardware. It represents the frame buffer video hardware and allows application software to access the graphics hardware through a well-defined interface, so that the software is not required to know anything about the low-level hardware registers. The SPDC driver supports this model with one key caveat: the contents of the frame buffer are not automatically updated to the Sipix display. Instead, a custom API function call is required to trigger an update to the Sipix display. The details of this process are explained in SPDC Frame Buffer Driver Extensions.

The frame buffer driver is enabled by selecting the frame buffer option under the graphics parameters in the kernel configuration. To supplement the frame buffer driver, the kernel builder may also include support for fonts and a startup logo. The frame buffer device depends on the virtual terminal (VT) console to switch from serial to graphics mode. The device is accessed through special device nodes, located in the /dev directory, as /dev/fb*. fb0 is generally the primary frame buffer.

A frame buffer device is a memory device, such as /dev/mem, and it has features similar to a memory device. Users can read it, write to it, seek to some location in it, and mmap() it (the main use). The difference is that the memory that appears in the special file is not the whole memory, but the frame buffer of some video hardware.

The SPDC frame buffer driver (<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc/mxc_spdc_fb.cdrivers/video/mxc/mxc_spdc_fb.c) interacts closely with the generic Linux frame buffer driver (drivers/video/fbmem.c).

For additional details on the frame buffer device, please refer to documentation in the Linux kernel found in Documentation/fb/framebuffer.txt.

6.3.2 SPDC Frame Buffer Driver Extensions

Sipix display technology, in conjunction with the SPDC, has several features that distinguish it from standard LCD-based frame buffer devices. These differences introduce the need for API extensions to the frame buffer interface. The SPDC refreshes the Sipix display asynchronously and supports partial screen updates. Therefore, the SPDC requires notification from the user when the frame buffer contents have been modified and which region needs updating. Another unique characteristic of SPDC updates to the Sipix display is the long screen update latencies (between 300-980ms), which introduces the need for a mechanism to allow the user to wait for a given screen update to complete.

The custom API extensions to the frame buffer device are accessible both from user space applications and from within kernel space. The standard device IOCTL interface provides access to the custom API for user space applications. The IOCTL extensions, along with relevant data structures and definitions, can be found in include/linux/mxcfb.h. A full description of these IOCTLs can be found in the Programming Interface section Programming Interface.

For kernel mode access to the custom API extensions, the IOCTL interface should be bypassed in favor of direct access to the underlying functions. These functions are included in include/linux/mxcfb_epdc_kernel.h, and are documented in the Programming Interface section Programming Interface.

6.3.3 SPDC Panel Configuration

The SPDC driver is designed to flexibly support Sipix panels with a variety of panel resolutions, timing parameters, and waveform modes. The SPDC driver is kept panel-agnostic through the use of an SPDC panel mode structure, imx_spdc_fb_mode, which can be found in arch/arm/plat-mxc/include/mach/epdc.h.

```
struct imx_spdc_fb_mode {
            struct fb_videomode *vmode;
            struct imx_spdc_panel_init_set *init_set;
            const char
*wave_timing;
};
```

The imx_spdc_fb_mode structure consists of an fb_videomode structure and a set of EPD timing parameters. The fb_videomode structure defines the panel resolution and the basic timing parameters (pixel clock frequency, hsync and vsync margins) and the additional timing parameters in imx_spdc_fb_mode define EPD-specific timing parameters, such as the source and gate driver timings. Please refer to the SPDC programming model section within the i.MX 6SoloLite Applications Processor Reference Manual for details on how Sipix panel timing parameters should be configured.

This SPDC panel mode is part of the mxc_spdc_fb_platform_data structure that is passed to the SPDC driver during driver registration.

```
struct imx_spdc_fb_platform_data {
            struct imx_spdc_fb_mode *spdc_mode;
            int num_modes;
            int (*get_pins) (void);
            void (*put_pins) (void);
            void (*enable_pins) (void);
            void (*disable_pins) (void);
};
```

In addition to the SPDC panel mode data, functions may be passed to the SPDC driver to define how to handle the SPDC pins when the SPDC driver is enabled or disabled. These functions should disable the SPDC pins for purposes of power savings.

6.3.3.1 Boot Command Line Parameters

Additional configuration for the SPDC driver is provided through boot command line parameters. The format of the command line option is as follows:

```
spdc video=mxcspdcfb:[panel_name],bpp=16
```

The SPDC driver parses these options and tries to match panel_name to the name of video mode specified in the mxc_spdc_fb_mode panel mode structure. If no match is found, then the first panel mode provided in the platform data is used by the SPDC driver. The bpp setting from this command line sets the initial bits per pixel setting for the frame buffer. A setting of 16 selects RGB565 pixel format, while a setting of 4 selects 4-bit grayscale (Y4) format.

6.3.4 SPDC Waveform Loading

The SPDC driver requires a waveform file for proper operation. This waveform file contains the waveform information needed to generate the waveforms that drive updates to the Sipix panel. A pointer to the waveform file data is programmed into the SPDC before the first update is performed.

There are two options for selecting a waveform file:

1. Select one of the default waveform files included in this BSP and built into the kernel.
2. Use a new waveform file that is specific to the Sipix panel being used.

The waveform file is loaded by the SPDC driver using the Linux firmware APIs.

6.3.4.1 Using a Default Waveform File

The quickest and easiest way to get started using an Sipix panel and the SPDC driver is to use one of the default waveform files provided in the Linux BSP. This should enable updates to several different types of Sipix panel without a panel-specific waveform file. The drawback is that optimal quality should not be expected. Typically, using a non-panel-specific waveform file for an Sipix panel results in more ghosting artifacts and overall poorer color quality.

The following default waveform files included in the BSP reside in firmware/imx/:

- spdc_pvi.fw - Default waveform for the AUO Sipix panel ERK_1_4_A01 version.

The SPDC driver attempts to load a waveform file with the name "imx/spdc_[wave_timing].fw", where wave_timing refers to the string specified in the imx_spdc_fb_mode `wave_timing` field.

6.3.4.2 Using a Custom Waveform File

To ensure the optimal Sipix display quality, use a waveform file specific to Sipix panel being used. The raw waveform file type requires conversion to a format that can be understood and read by the SPDC. This conversion script is not included as part of the BSP, so please contact Freescale to acquire this conversion script.

Once the waveform conversion script has been run on the raw waveform file, the converted waveform file should be renamed so that the SPDC driver can find it and load it. The driver is going to search for a waveform file with the name "imx/spdc_[wave_timing].fw", where wave_timing refers to the string specified in the imx_spdc_fb_mode `wave_timing` field. For example, if the panel waveform is named "pvi", then the converted waveform file should be named spdc_pvi.fw.

The firmware script firmware.sh (lib/udev/firmware in the Linux root file system) contains the search path used to locate the firmware file. The default search path for firmware files is /lib/firmware;/usr/local/lib/firmware. A custom search path can be specified by modifying firmware.sh. You’ll need to create an imx directory in one of these paths and add your new spdc_[wave_timing].fw file there.

NOTE

If the SPDC driver is searching for a firmware waveform file that matches the names of one of the default waveform files (see preceding chapter), it will choose the default firmware files that are built into the BSP over any firmware file that has been added in the firmware search path. Therefore, if you leave the BSP so that it builds those default firmware files into the OS image, be sure to use a panel name other than those associated with the default firmware files, since those default waveform files will be preferred and selected over a new waveform file placed in the firmware search path.

6.3.5 SPDC Panel Initialization

The framebuffer driver will not typically (see note below for exceptions) go through any hardware initialization steps when the framebuffer driver module is loaded. Instead, a subsequent user mode call must be made to request that the driver initialize itself for a specific EPD panel. To initialize the SPDC hardware and E-ink panel, an FBIOPUT_VSCREENINFO ioctl call must be made, with the xres and yres fields of the fb_var_screeninfo parameter set to match the X and Y resolution of a supported E-ink panel type. To ensure that the SPDC driver receives the initialization request, the activate field of the fb_var_screeninfo parameter should be set to FB_ACTIVATE_FORCE.

NOTE

The exception is when the FB Console driver is included in the kernel. When the SPDC driver registers the framebuffer device, the FB Console driver will subsequently make an FBIOPUT_VSCREENINFO ioctl call. This will in turn initialize the SPDC panel.

6.3.6 Grayscale Framebuffer Selection

The SPDC framebuffer driver supports the use of 4-bit grayscale (Y4) and 4-bit inverted grayscale (Y4 inverted) pixel formats for the framebuffer (in addition to the more common RGB565 pixel format). In order to configure the framebuffer format as 4-bit grayscale, the application would call the FBIOPUT_VSCREENINFO framebuffer ioctl. This ioctl takes an fb_var_screeninfo pointer as a parameter. This parameter specifies the attributes of the framebuffer and allows the application to request changes to the framebuffer format. There are two key members of the fb_var_screeninfo parameter that must be set in order to request a change to 4-bit grayscale format: bits_per_pixel and grayscale. bits_per_pixel must be set to 4, and grayscale must be set to one of the 2 valid grayscale format values: GRAYSCALE_4BIT or GRAYSCALE_4BIT_INVERTED.

The following code snippet demonstrates a request to change the framebuffer to use the Y4 pixel format:

```
fb_screen_info screen_info;
screen_info.bits_per_pixel = 4;
screen_info.grayscale = GRAYSCALE_4BIT;
retval = ioctl(fd_fb0, FBIOPUT_VSCREENINFO, &screen_info);
```

6.4 Source Code Structure

Table below lists the source files associated with the SPDC driver. These files are available in the following directory:

```
drivers/video/mxc
```

Table 6-1. SPDC Driver Files

File	Description
mxc_spdc_fb.c	The SPDC frame buffer driver.
mxc_spdc_fb.h	Register definitions for the SPDC module.

Table below lists the global header files associated with the SPDC driver. These files are available in the following directory:

```
include/linux/
```

Table 6-2. SPDC Global Header Files

File	Description
mxcfb.h	Header file for the MXC framebuffer drivers
mxcfb_epdc_kernel.h	Header file for direct kernel access to the SPDC API extension

6.5 Menu Configuration Options

The following Linux kernel configuration options are provided for the SPDC module. To get to these options use the command ./ltib -c when located in the <ltib dir>. On the screen displayed, select **Configure the kernel** and exit. When the next screen appears select the options to configure.

- CONFIG_FB_MXC_SIPIX_PANEL includes support for the Electrophoretic Display Controller. In menuconfig, this option is available under:
  - Device Drivers > Graphics Support > Sipix Panel Framebuffer
- CONFIG_MXC_SIPIX_AUTO_UPDATE_MODE option enables support for auto-update mode which provides automated EPD updates through the deferred I/O framebuffer driver. This option is dependent on the MXC_SIPIX_PANEL option. In menuconfig, this option is available under:
  - Device Drivers > Graphics Support > Sipix Auto-update Mode Support

NOTE

This option only enables the use of auto-update mode. Turning on auto-update mode requires an additional IOCTL call using the MXCFB_SET_AUTO_UPDATE_MODE IOCTL.

- CONFIG_FB is the configuration option to include frame buffer support in the Linux kernel. In menuconfig, this option is available under:
  - Device Drivers > Graphics support > Support for frame buffer devices
  - By default, this option is Y for all architectures.
- CONFIG_FB_MXC is the configuration option for the MXC Frame buffer driver. This option is dependent on the CONFIG_FB option. In menuconfig, this option is available under:
  - Device Drivers > Graphics support > MXC Framebuffer support
  - By default, this option is Y for all architectures.
- CONFIG_MXC_PXP configuration option enables support for the PxP. The PxP is required by the SPDC driver for processing (color space conversion, rotation, auto-waveform selection) framebuffer update regions. This option must be selected for the SPDC framebuffer driver to operate correctly. In menuconfig, this option is available under:
  - Device Drivers > DMA Engine support > MXC PxP support

6.6 Programming Interface

6.6.1 IOCTLs/Functions

The SPDC Frame Buffer is accessible from user space and from kernel space. A single set of functions describes the SPDC Frame Buffer driver extension, but there are two modes for accessing these functions. For user space access, the IOCTL interface should be used, and for kernel space access, the functions should be called directly. For each function below, both the IOCTL code and the corresponding kernel function is listed.

MXCFB_SET_WAVEFORM_MODES / mxc_spdc_fb_set_waveform_modes()

Description:

Defines a mapping for common waveform modes.

Parameters:

mxcfb_waveform_modes **modes*

Pointer to a structure containing the waveform mode values for common waveform modes. These values must be configured in order for automatic waveform mode selection to function properly.

MXCFB_SET_TEMPERATURE / mxc_spdc_fb_set_temperature

Description:

Set the temperature to be used by the SPDC driver in subsequent panel updates.

Parameters:

int*32_t temperature*

Temperature value, in degrees Celsius. Note that this temperature setting may be overridden by setting the temperature value parameter to anything other than SPDC_DEFAULT_TEMP when using the MXCFB_SEND_UPDATE ioctl.

MXCFB_SET_AUTO_UPDATE_MODE / mxc_spdc_fb_set_auto_update

Description:

Select between automatic and region update mode.

Parameters:

__u32 mode

In region update mode, updates must be submitted via the MXCFB_SEND_UPDATE IOCTL.

In automatic mode, updates are generated automatically by the driver by detecting pages in frame buffer memory region that have been modified.

MXCFB_SET_UPDATE_SCHEME / mxc_spdc_fb_set_upd_scheme

Description:

Select a scheme that dictates how the flow of updates within the driver.

Parameters:

__u32 scheme

Select of the following updates schemes:

UPDATE_SCHEME_SNAPSHOT - In the Snapshot update scheme, the contents of the framebuffer are immediately processed and stored in a driver-internal memory buffer. By the time the call to MXCFB_SEND_UPDATE has completed, the framebuffer region is free and can be modified without affecting the integrity of the last update. If the update frame submission is delayed due to other pending updates, the original buffer contents will be displayed when the update is finally submitted to the SPDC hardware.

UPDATE_SCHEME_QUEUE - The Queue update scheme uses a work queue to aynchronously handle the processing and submission of all updates. When an update is submitted via MXCFB_SEND_UPDATE, the update is added to the queue, and then processed in order, as SPDC hardware resources become available. As a result, the framebuffer contents processed and updated are not guaranteed to reflect what was present in the framebuffer when the update was sent to the driver.

UPDATE_SCHEME_QUEUE_AND_MERGE - The Queue and Merge scheme uses the queueing concept from the Queue scheme, but adds a merging step. This means that before an update is processed in the work queue, it is first compared with other pending updates. If any update matches the mode and flags of the current update, and also overlaps the update region of the current update, then that update will be merged with the current update. After attempting to merge all pending updates, the final merged update will be processed and submitted.

MXCFB_SEND_UPDATE / mxc_spdc_fb_send_update

Description:

Request a region of the frame buffer be updated to the display.

Parameters:

mxcfb_update_data **upd_data*

Pointer to a structure defining the region of the frame buffer, waveform mode, and collision mode for the current update. This structure also includes a flags field to select from one of the following update options:

SPDC_FLAG_ENABLE_INVERSION - Enables inversion of all pixels in the update region.

SPDC_FLAG_FORCE_MONOCHROME - Enables full black/white posterization of all pixels in the update region.

SPDC_FLAG_USE_ALT_BUFFER - Enables updating from an alternate (non-framebuffer) memory buffer.

If enabled, the final *upd_data* parameter includes detailed configuration information for the alternate memory buffer.

MXCFB_WAIT_FOR_UPDATE_COMPLETE / mxc_spdc_fb_wait_update_complete

Description:

Block and wait for a previous update request to complete.

Parameters:

mxfb_update_marker_data *marker_data*

The update_marker value used to identify a particular update (passed as a parameter in MXCFB_SEND_UPDATE IOCTL call) should be re-used here to wait for the update to complete. If the update was a collision test update, the collision_test variable will return the result indicating whether a collision occurred.

MXCFB_SET_PWRDOWN_DELAY / mxc_spdc_fb_set_pwrdown_delay

Description:

Set the delay between the completion of all updates in the driver and when the driver should power down the SPDC and the Sipix display power supplies.

Parameters:

int32_t *delay*

Input delay value in milliseconds. To disable SPDC power down altogether, use FB_POWERDOWN_DISABLE (defined below).

MXCFB_GET_PWRDOWN_DELAY / mxc_spdc_fb_get_pwrdown_delay

Description:

Retrieve the driver's current power down delay value.

Parameters:

int32_*t delay*

Output delay value in milliseconds.

6.6.2 Structures and Defines

```
#define GRAYSCALE_4BIT                          0x3
#define GRAYSCALE_4BIT_INVERTED                 0x4

#define AUTO_UPDATE_MODE_REGION_MODE                                        0

#define AUTO_UPDATE_MODE_AUTOMATIC_MODE                                     1

#define UPDATE_SCHEME_SNAPSHOT                                              0
#define UPDATE_SCHEME_QUEUE                                                 1
#define UPDATE_SCHEME_QUEUE_AND_MERGE                                       2

#define UPDATE_MODE_PARTIAL                                                 0x0
#define UPDATE_MODE_FULL                                                    0x1

#define WAVEFORM_MODE_AUTO                                                  257

#define SPDC_DEFAULT_TEMP                                                    30

#define EPDC_FLAG_ENABLE_INVERSION                                          0x01
#define EPDC_FLAG_FORCE_MONOCHROME                                          0x02
#define EPDC_FLAG_USE_ALT_BUFFER                                            0x100

#define FB_POWERDOWN_DISABLE                                                -1

struct mxcfb_rect {
    __u32 left; /* Starting X coordinate for update region */
    __u32 top; /* Starting Y coordinate for update region */
    __u32 width; /* Width of update region */
    __u32 height; /* Height of update region */
};

struct mxcfb_waveform_modes {
    int mode_init; /* INIT waveform mode */
    int mode_du; /* DU waveform mode */
    int mode_gc4; /* GC4 waveform mode */
    int mode_gc8; /* GC8 waveform mode */
    int mode_gc16; /* GC16 waveform mode */
    int mode_gc32; /* GC32 waveform mode */
};

struct mxcfb_alt_buffer_data {
    __u32 phys_addr; /* physical address of alternate image buffer */
    __u32 width; /* width of entire buffer */
    __u32 height; /* height of entire buffer */
    struct mxcfb_rect alt_update_region; /* region within buffer to update */
};

struct mxcfb_update_data {
    struct mxcfb_rect update_region; /* Rectangular update region bounds */
    __u32 waveform_mode; /* Waveform mode for update */
    __u32 update_mode; /* Update mode selection (partial/full) */
    __u32 update_marker; /* Marker used when waiting for completion */
    int temp; /* Temperature in Celsius */
    uint flags; /* Select options for the current update */
    struct mxcfb_alt_buffer_data alt_buffer_data; /* Alternate buffer data */
};
struct mxcfb_update_marker_data { __u32 update_marker; __u32 collision_test; };
```

Chapter 7
Pixel Pipeline (PxP) DMA-ENGINE Driver

7.1 Introduction

The Pixel Pipeline (PxP) DMA-ENGINE driver provides a unique API, which are implemented as a dmaengine client that smooths over the details of different hardware offload engine implementations. Typically, the users of PxP DMA-ENGINE driver include EPDC driver, V4L2 Output driver, and the PxP user-space library.

7.2 Hardware Operation

The PxP driver uses PxP registers to interact with the hardware. For detailed hardware operation, please refer to the *i.MX 6SoloLite Multimedia Applications Processor Reference Manual.*

7.3 Software Operation

7.3.1 Key Data Structs

The PxP DMA Engine driver implementation depends on the DMA Engine Framework. There are three important structs in the DMA Engine Framework which are extended by the PxP driver: struct dma_device, struct dma_chan, struct dma_async_tx_descriptor. The PxP driver implements several callback functions which are called by the DMA Engine Framework (or DMA slave) when a DMA slave (client) interacts with the DMA Engine.

The PxP driver implements the following callback functions in struct dma_device:

device_alloc_chan_resources / allocate resources and descriptors */*

device_free_chan_resources / release DMA channel's resources */*

device_tx_status / poll for transaction completion */*

device_issue_pending / push pending transactions to hardware */*

and,

device_prep_slave_sg / prepares a slave dma operation */*

device_terminate_all / manipulate all pending operations on a channel, returns zero or error code */*

The first four functions are used by the DMA Engine Framework, the last two are used by the DMA slave (DMA client). Notably, *device_issue_pending* is used to trigger the start of a PxP operation.

The PxP DMA driver also implements the interface *tx_submit in struct dma_async_tx_descriptor*, which is used to prepare the descriptor(s) which will be executed by the engine. When tasks are received in pxp_tx_submit, they are not configured and executed immediately. Rather, they are added to a task queue and the function call is allowed to return immediately.

7.3.2 Channel Management

Although ePxP does not have multiple channels in hardware, 16 virtual channels are supported in the driver; this provides flexibility in the multiple instance/client design. At any time, a user can call *dma_request_channel*() to get a free channel, and then configure this channel with several descriptors (a descriptor is required for each input plane and for the output plane). When the PxP is no longer being used, the channel should be released by calling *dma_release_channel*(). Detailed elements of channel management are handled by the driver and are transparent to the client.

7.3.3 Descriptor Management

The DMA Engine processes the task based on the descriptor. One DMA channel is usually associated with several descriptors. Descriptors are recycled resources, under control of the offload engine driver, to be reused as operations complete. The extended TX descriptor packet (pxp_tx_desc), allows the user to pass PxP configuration information to the driver. This includes everything that the PxP needs to execute a processing task.

7.3.4 Completion Notification

There are two ways for an application to receive notification that a PxP operation has completed.

- Call dma_wait_for_async_tx(). This call causes the CPU to spin while it polls for the completion of the operation.
- Specify a completion callback.

The latter method is recommended. After the PxP operation completes, the PxP output buffer data can be retrieved.

For general information for DMA Engine Framework, please refer to *Documentation/dmaengine.txt* in the Linux kernel source tree.

7.3.5 Limitations

- The driver currently does not support scatterlist objects in the way they are traditionally used. Instead of using the scatterlist parameter object to provide a chain of memory sources and destinations, the driver currently uses it to provide the input and output buffers (and overlay buffers, if needed) for one transfer.
- The PxP driver may not properly execute a series of transfers that is queued in rapid sequence. It is recommended to wait for each transfer to complete before submitting a new one.

7.4 Menu Configuration Options

The following Linux kernel configuration option is provided for this module:

Device Drivers --->

DMA Engine support --->

[*] MXC PxP support

[*] MXC PxP Client Device

7.5 Source Code Structure

The PxP driver source code is located in drivers/dma/ and include/linux/.

Chapter 8
ELCDIF Frame Buffer Driver

8.1 Introduction

The ELCDIF frame buffer driver is designed using the Linux kernel frame buffer driver framework. It implements the platform driver for a frame buffer device. The implementation uses the ELCDIF API for generic LCD low-level operations. The ELCDIF API is also defined in the ELCDIF frame buffer driver to realize low level hardware control. Only DOTCLK mode of the ELCDIF API is tested, so theoretically the ELCDIF frame buffer driver can work with a sync LCD panel driver to support a frame buffer device. The sync LCD driver is organized in a flexible and extensible manner and is abstracted from any specific sync LCD panel support. To support another sync LCD panel, the user can write a sync LCD driver by referring to the existing one.

8.2 Hardware Operation

For detailed hardware operation, please refer to the *i.MX 6SoloLite Multimedia Applications Processor Reference Manual.*

8.3 Software Operation

A frame buffer device is a memory device similar to /dev/mem and it has the same features. It can be read from, written to, or some location in it can be sought and maped using mmap(). The difference is that the memory that appears is not the whole memory, but only the frame buffer of the video hardware. The device is accessed through special device nodes, usually located in the /dev directory, /dev/fb*. /dev/fb* also has several IOCTLs which act on it and through which information about the hardware can be queried and set. The color map handling operates through IOCTLs as well. See linux/fb.h for more information on which IOCTLs there are and which data structures they use.

The frame buffer driver implementation for i.MX 6 is abstracted from the actual hardware. The default panel driver is picked up by video mode defined in platform data or passed in with 'video=mxc_elcdif_fb:resolution, bpp=bits_per_pixel' kernel bootup command during probing, where resolution should be in the common frame buffer video mode pattern and bits_per_pixel should be the frame buffer's color depth.

8.4 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_FB_MXC_SEIKO_WVGA_SYNC_PANEL [=Y|N|M]
- CONFIG_FB_MXC_ELCDIF_FB [=Y|N|M] Configuration option to compile support for the MXC ELCDIF frame buffer driver into the kernel.

8.5 Source Code Structure

The frame buffer driver source code is in drivers/video/mxc/mxc_elcdif_fb.c.

The panel support code is located in drivers/video/mxc/mxcfb_claa_wvga.c and drivers/video/mxc/mxcfb_seiko_wvga.c.

The frame buffer driver includes the source/header files shown in table below.

Table 8-1. Frame Buffer Driver Files

File	Description
drivers/video/mxc/elcdif_regs.h	The register head file for ELCDIF module
include/linux/mxcfb.h	The head file for MXC frame buffer drivers

Chapter 9
Graphics Processing Unit (GPU)

9.1 Introduction

The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D graphics applications.

The 2D graphics processing unit (GPU2D) is based on the Vivante GC320 core, which is an embedded 2D graphics accelerator targeting graphical user interfaces (GUI) rendering boost. The VG graphics processing unit (GPUVG) is based on the Vivante GC355 core, which is an embedded vector graphic accelerator for supporting the OpenVG 1.1 graphics API and feature set. The GPU driver kernel module source is in kernel source tree, but the libs are delivered as binary only.

9.1.1 Driver Features

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

- EGL (EGL is an interface between Khronos rendering APIs such as OpenGL ES or OpenVG and the underlying native platform window system) 1.4 API defined by Khronos Group.
- OpenVG (OpenVG is a 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.

9.1.1.1 Hardware Operation

For detailed hardware operation and programming information, see the GPU chapter in the *i.MX 6SoloLiteApplications Processor Reference Manual*.

9.1.1.2 Software Operation

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

- EGL 1.4 API
- OpenVG 1.1 API

9.1.1.3 Source Code Structure

Table below lists GPU driver kernel module source structure:

<ltib_dir>/rpm/BUILD/linux/drivers/mxc/gpu-viv

Table 9-1. GPU Driver Files

File	Description
Kconfig Kbuild config	kernel configure file and makefile
arch/XAQ2/hal/kernel	hardware specific driver code for and GC320
arch/GC350/hal/kernel	hardware specific driver code for GC350
hal/kernel	Kernel mode HAL driver
hal/os	os layer HAL driver

9.1.1.4 Library Structure

Table below lists GPU driver user mode library structure:

<ROOTFS>/usr/lib

Table 9-2. GPU Library Files

File	Description
libCLC.so	OpenCL frontend compiler library
libEGL.so*	EGL1.4 library
libGAL.so*	GAL user mode driver

Table continues on the next page...

Table 9-2. GPU Library Files (continued)

File	Description
libGLES_CL.so	OpenGL ES 1.1 common lite library (without EGL API, no float point support API)
libGL.so.1.2	OpenGL 2.1 common library
libGLES_CM.so	OpenGL ES 1.1 common library (without EGL API, include float point support API)
libGLESv1_CL.so	OpenGL ES 1.1 common lite library (with EGL API, no float point support API)
libGLESv1_CM.so	OpenGL ES 1.1 common library (with EGL API, include float point support API)
libGLESv2.so	OpenGL ES 2.0 library
libGLSLC.so	OpenGL ES shader language compiler library
libOpenCL.so	OpenCL 1.1 EP library
libOpenVG.so*	OpenVG 1.1 library
libVDK.so	VDK wrapper library.
libVIVANTE.so*	Vivante user mode driver.
directfb-1.4-0/gfxdrivers/libdirectfb_gal.so	DirectFB 2D acceleration library.
dri/vivante_dri.so	DRI library for OpenGL2.1.

* These libraries are actually symbolic links to the actual library file in the folder.

By default, these symbolic links are installed to point to the frame buffer version of the libraries as such:

```
libGAL.so -> libGAL-fb.so
libEGL.so -> libEGL-fb.so
libVIVANTE.so -> libVIVANTE-fb.so
```

On X11 systems, the symbolic links to these libraries need to be redirected. This can be done using the following sequence of commands:

```
> cd <ROOTFS>/usr/lib
> sudo ln -s libGAL-x11.so libGAL.so
> sudo ln -s libEGL-x11.so libEGL.so
> sudo ln -s libEGL-x11.so libEGL.so.1
> sudo ln -s libVIVANTE-x11.so libVIVANTE.so
```

On directFB backend, the symbolic links to these libraries need to be redirected. This can be done using the following sequence of commands:

```
> cd <ROOTFS>/usr/lib
> sudo ln -s libGAL-dfb.so libGAL.so
> sudo ln -s libEGL-dfb.so libEGL.so
> sudo ln -s libEGL-dfb.so libEGL.so.1
> sudo ln -s libVIVANTE-dfb.so libVIVANTE.so
```

9.1.1.5 API References

Refer to the following web sites for detailed specifications:

- EGL 1.4 API: http://www.khronos.org/egl/
- OpenVG 1.1 API: http://www.khronos.org/openvg/

9.1.1.6 Menu Configuration Options

The following Linux kernel configurations are provided for GPU driver:

CONFIG_MXC_GPU_VIV is a configuration option for GPU driver. In menucon figuration this option is available under Device Drivers > MXC support drivers > MXC Vivante GPU support > MXC Vivante GPU support.

To get to the GPU library package in LTIB, use the command ./ltib -c when located in the <ltib dir>. On the displayed screen, select **Configure the kernel**, select Device Drivers > MXC support drivers > MXC Vivante GPU support > MXC Vivante GPU support, and then exit. When the next screen appears, select the following options to enable the GPU driver:

- Package list > gpu-viv-bin-mx6q
- This package provides proprietary binary libraries, and test code built from the GPU for framebuffer

Chapter 10
High-Definition Multimedia Interface (HDMI)

10.1 Introduction

The High Definition Multimedia Interface (HDMI) driver supports the external SiI9022 HDMI hardware module, which provides the capability to transfer uncompressed video, audio, and data using a single cable.

The HDMI driver is divided into two sub-components: a video display device driver that integrates with the Linux Frame Buffer API and an S/PDIF audio driver that transfers S/PDIF audio data to SiI9022 HDMI hardware module.

The HDMI driver is only for demo application and supports the following features:

- HDMI video output supports 1080p60 and 720p60 resolutions.
- Support for reading EDID information from an HDMI sink device for video.
- Hotplug detection
- HDMI audio playback (2 channels, 16/24bit, 44.1KHz sample rate)

10.2 Software Operation

The HDMI driver is divided into sub-components based on its two primary purposes: providing video and audio to an HDMI sink device.

The audio output depends on video display.

10.2.1 Hotplug Handling and Video Mode Changes

Once the connection between the ELCDIF and the HDMI has been established through the MXC Display Driver interface, the HDMI video driver waits for a hotplug interrupt indicating that a valid HDMI sink device is connected and ready to receive HDMI video data. Subsequent communications between the HDMI and LECDIF FB are conducted through the Linux Frame Buffer APIs. The following list demonstrates the software flow to recognize a HDMI sink device and configure the ELCDIF FB driver to drive video output:

1. The HDMI video driver receives a hotplug interrupt and reads the EDID from the HDMI sink device constructing a list of video modes from the retrieved EDID information. Using either the video mode string from the Linux kernel command line (for the initial connection) or the most recent video mode (for a later HDMI cable connection), the HDMI driver selects a video mode from the mode list that is the closest match.
2. The HDMI video driver calls `fb_set_var()` to change the video mode in the ELCDIF FB driver. The ELCDIF FB driver completes its reconfiguration for the new mode.
3. As a result of calling fb_set_var(), a FB notification is sent back to the HDMI driver indicating that an FB_EVENT_MODE_CHANGE has occurred. The HDMI driver configures the HDMI hardware for the new video mode.
4. Finally, the HDMI module is enabled to generate output to the HDMI sink device.

10.3 Source Code Structure

The bulk of the source code for the HDMI driver is divided amongst the three software components that comprise the driver: the HDMI display driver, and the HDMI audio driver.

The source code for the HDMI display driver is available in the <ltib_dir>/rpm/BUILD/linux/drivers/video/mxc directory.

Table 10-1. HDMI Display Driver File List

File	Description
mxcfb_sii902x_elcdif.c	HDMI display driver implementation.

The source code for the HDMI audio driver is available in the <ltib_dir>/rpm/BUILD/linux/drivers and sound/soc/ director. HDMI Audio data source comes from S/PDIF TX.

Table 10-2. HDMI Audio Driver File List

File	Description
sound/codecs/mxc_spdif.c	S/PDIF Audio SoC CODEC driver implementation.
sound/soc/imx/imx-spdif.c	S/PDIF Audio SoC Machine driver implementation.
sound/soc/imx/imx-spdif-dai.c	S/PDIF Audio SoC DAI driver implementation.
sound/soc/imx/imx-pcm-dma-mx2.c	S/PDIF Audio SoC platform layer driver implementation.

10.3.1 Linux Menu Configuration Options

There are two main Linux kernel configuration options used to select and include HDMI driver functionality in the Linux OS image.

The CONFIG_FB_MXC_SII902X_ELCDIFI option provides support for the Sii902x HDMI video driver and can be selected in menuconfig at the following menu location:

- Device Drivers > Graphics support > MXC Framebuffer support.

HDMI video support is dependent on MXC ELCDIF Framebuffer.

The CONFIG_SND_MXC_SPDIF option provides support for the HDMI Audio driver and can be selected in menuconfig at the following menu location:

- Device Drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio for Freescale i.MX CPUs > SoC Audio support for IMX - S/PDIF

10.4 Unit Test

The HDMI video and audio drivers each have their own set of tests.

The preparation for HDMI test:

- Insert the HDMI daughter card into J13 on EVK.
- Insert the HDMI cable into the HDMI slots of both HDMI daughter board and the HDMI sink device.
- Power on the HDMI sink device.

10.4.1 Video

The following set of manual tests can be used to verify the proper operation of the HDMI video driver:

1. Hotplug testing: Connect and disconnect the HDMI cable several times, from either the end attached to the i.MX board, or the end attached to the HDMI sink device. Each time the cable is reconnected, the driver should re-determine the appropriate video mode based on the modes read via EDID from the HDMI sink and display that mode on the sink device.

2. HDMI output device testing: Test by dynamically switching the HDMI sink device. The HDMI driver should be able to detect the valid video modes for each different HDMI sink device and provide video to that display that is closest to the most recent video mode configured in the HDMI driver.

10.4.2 Audio

The following sequence of tests verifies the correct operation of the HDMI audio driver:

1. Ensure that an HDMI cable is connected between the HDMI daughter board and the HDMI sink device, and that the HDMI video image is being properly displayed on the device.
2. Use this command line to play out a pcm audio file "file.wav" to HDMI sink device:

```
$ aplay -Dplughw:1,0 file.wav
```

Chapter 11
X Windows Acceleration

11.1 Introduction

X-Windows System (aka X11 or X) is a portable, client-server based, graphics display system.

X-Windows System can run with a default frame buffer driver which handles all drawing operations to the main display. Because there is a 2D GPU (graphics processing unit) available, some drawing operations can be accelerated. High level X operations may get decomposed into many low-level drawing operations where these low level operations are accelerated for X-Windows System.

11.2 Hardware Operation

X-Windows System acceleration on i.MX 6 uses the Vivante GC320 2D GPU.

Acceleration is also dependent on the frame buffer memory.

11.3 Software Operation

X-Windows acceleration is supported by X.org X Server version 1.7.6 and later versions, as well as the EXA interface version 2.5.

The types of operations that are accelerated for X11 are as follows. All operations involve frame buffer memory which may be on screen or off screen:

- Solid fill of a rectangle.
- Upload image from the system memory to the video memory.

- Copy of a rectangle with same pixel format with possible source-target rectangle overlap.
- Copy of a rectangle supporting most XRender compositing operations with these options:
    - Pixel format conversion.
    - Repeating pattern source.
    - Porter-Duff blending of source with target.
    - Source alpha masking.

The following are additional features supported by the X-Windows acceleration:

- X pixmaps can be allocated directly in frame buffer memory.

11.3.1 X Windows Acceleration Architecture

The following block diagram shows the components that are involved in the acceleration of X-Windows System:

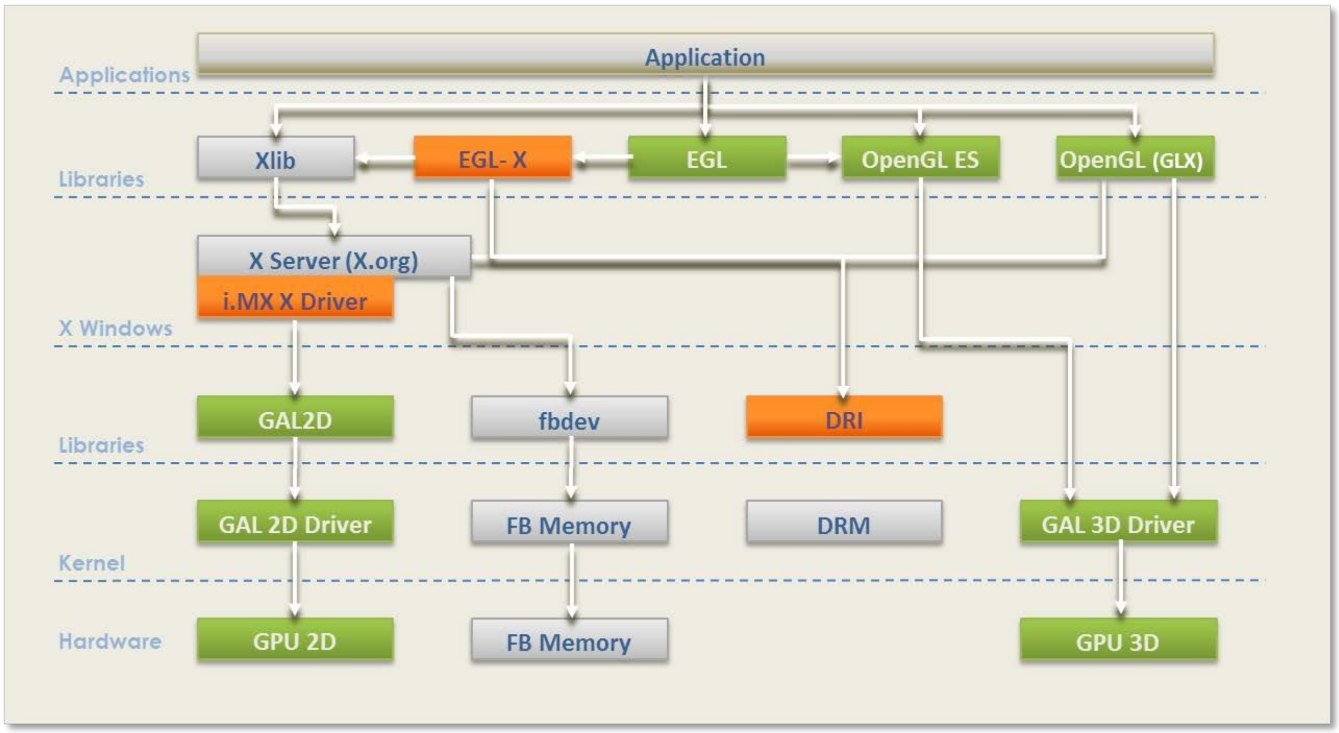


Figure 11-1. X Driver Architecture

The components shown in green are those provided as part of the Vivante 2D/3D GPU driver support which includes OpenGL/ES and EGL, though some of the families in the i.MX 6 series, such as the i.MX 6SoloLite, do not contain 3D HW module. The components shown in light gray are the standard components in the X-Windows System without acceleration. The components shown in orange are those added to support X-Windows System acceleration and are briefly described here.

The **i.MX X Driver** library module (`vivante-drv.so`) is loaded by the X server and contains the high-level implementation of the X-Windows acceleration interface for i.MX platforms containing the GC320 2D GPU core. The entire linearly contiguous frame buffer memory in `/dev/fb0` is used for allocating pixmaps for X both on screen and off screen. The driver supports a custom X extension which allows X clients to query the GPU address of any X pixmap stored in frame buffer memory.

The **libGAL.so** library module (`libGAL.so`) contains the register-level programming interface to the GC320 GPU module. This includes the storing of register programming commands into packets which can be streamed to the device. The functions in the libGAL.so library are called by the i.MX X Driver code.

11.3.2 i.MX 6 Driver for X-Windows System

The i.MX X Driver, referred to as vivante-drv.so, implements the EXA interface of the X server in providing acceleration.

The implementation details are as follows:

- The implementation builds upon the source from the fbdev frame buffer driver for X, so that it can be the fallback when the acceleration is disabled.
- The implementation is based on X server EXA version 2.5.0.
- The EXA solid fill operation is accelerated, except for source/target drawables containing less than 1024x1024 pixels, in which case software failure may occur.
- The EXA copy operation is accelerated, except for source/target drawables containing less than 1024x1024 pixels, in which case software failure may occur.
- EXA putimage (upload into video memory) is accelerated, except for source drawables containing less than 400x400 pixels, in which case software failure may occur.
- For EXA solid fill, only solid plane masks and only GXcopy raster-op operations are accelerated.
- For EXA copy operation, the raster-op operations (GXandInverted, GXnor, GXorReverse, GXorInverted, GXnand) are not accelerated.
- EXA composite allows for many options and combinations of source/mask/target for rendering. Commonly used EXA composite operations are accelerated.

The following types of EXA composite operations are accelerated:

- Composite operations for source/target drawables containing at least 640 pixels. If less than 640 pixels, the composite path falls to software.
- Simple source composite operations are used when source/target drawables contain more than 1024x1024 pixels (operations with mask not supported).
- Constant source (with or without alpha mask) composite with target.
- Repeating pattern source (with or without alpha mask) composite with target.
- Only these blending functions: SOURCE, OVER, IN, IN-REVERSE, OUT-REVERSE, and ADD (some of these need to support the accelerated component-alpha blending).
- In general, the following types of less commonly used EXA composite operations are not accelerated:
    - Transformed (meaning scaled, rotated) sources and masks.
    - Gradient sources.
    - Alpha masks with repeating patterns.

The implementation handles all pixmap allocation for X through the EXA callback interface. A first attempt is made to allocate the memory where it can be accessed by a physical GPU address. This attempt may fail if there is insufficient GPU accessible memory remaining, but it can also fail when the bits per pixel, which are being requested for the pixmap, are less than 8. If the attempt to allocate from the GPU accessible memory fails, the memory is allocated from the system. If the pixmap memory is allocated from the system, this pixmap cannot be involved in GPU accelerated option. The number of pitch bytes used to access the pixmap memory may be different depending on whether it was allocated from GPU accessible memory or from the system.

Once the memory for X pixmap has been allocated, no matter it is from GPU accessible memory or from the system, the pixmap is locked and can never migrate to other type of memory. Pixmap migration from GPU accessible memory to system memory is not necessary since a system virtual address is always available for GPU accessible memory. Pixmap migration from system memory to GPU accessible memory is not currently implemented, but would only help in situations where there was insufficient GPU accessible memory at initial allocation. More memory, however, becomes available (through de-allocation) at a later time.

The GPU accessible memory pitch (horizontal) alignment for the GC320 is 8 pixels.

All of the memory allocated for `/dev/fb0` is made available to an internal linear offscreen memory manager based on the one used in EXA. The portion of this memory beyond the screen memory is available for allocation of X pixmap, where this memory area is GPU accessible. The amount of memory allocated to `/dev/fb0` needs to be several MB more than the amount needed for the screen. The actual amount needed depends on the number of X-Windows and pixmaps used, the possible usage of X pixmaps as textures, and whether X-Windows are using the XComposite extension.

X extension is provided, so that X clients can query the physical GPU address associated with an X pixmap if that X pixmap was allocated in the GPU accessible memory.

11.3.3 xorg.conf for i.MX 6

The /etc/X11/xorg.conf file must be properly configured to use the i.MX 6 X Driver.

This configuration appears in a `Device` section of the file which contains both mandatory and optional entries. The following example shows a preferred configuration for using the i.MX 6 X Driver:

```
Section "Device"
        Identifier      "i.MX Accelerated Framebuffer Device"
        Driver          "vivante"
        Option          "fbdev"         "/dev/fb0"
        Option          "vivante_fbdev" "/dev/fb0"
        Option          "HWcursor"      "false"
EndSection



Section "Monitor"
        Identifier      "Configured Monitor"
EndSection



Section "Screen"
        Identifier      "Default Screen"
        Monitor         "Configured Monitor"
        Device          "i.MX Accelerated Framebuffer Device"
EndSection



Section "ServerLayout"
        Identifier      "Default Layout"
        Screen          "Default Screen"
EndSection
```

Some important entries recognized by the i.MX X Driver are described as follows:

- Device Identifier and Screen Device String

  The mandatory Identifierentry in the Device section specifies the unique name to associate with this graphics device.

  ```
  Section "Device"
          Identifier      "i.MX Accelerated Framebuffer Device"
  ```

  The following entry ties a specific graphics device to a screen. The Device Identifier string must match the `Device` string in a `Screen` section of the `xorg.conf` file. For example,

```
Section "Screen"
        ...
        Device          "i.MX Accelerated Framebuffer Device"
        ...
EndSection
```

- Device Driver String

  The mandatory Driver entry specifies the name of the loadable i.MX X driver.

```
Section "Device"
        ...
        Driver          "vivante"
        ...
EndSection
```

- Device fbdevPath Strings

  The mandatory entries fbdev and vivante_devspecifies the path for the frame buffer device to use.

```
Section "Device"
        ...
        Option          "fbdev"          "/dev/fb0"
        Option          "vivante_fbdev" "/dev/fb0"
        ...
EndSection
```

11.3.4 Setup X-Windows System Acceleration

Setup of X-Windows system acceleration consists of package installation and verification, file verification, and verifying acceleration. The debian packages are only available for ubuntu root fs. There's no gpu driver for X11 on gnome mobile root fs or LTIB

- Package installation and verification:
  - Verify that the following packages are available and installed:

    `gpu-viv-bin-mx6q_<bsp-version>_armel.deb`
  - This package contains gpu driver develop headers, and is installed in the `/usr/include` folder
  - This package contains gpu driver hal library`libGAL.so`
  - All above libraries are installed in the `/usr/lib` folder
  - `xserver-xorg-video-imx-viv_<bsp-version>_armel.deb`
  - This package contains the vivante-drv.so driver module for X acceleration and is installed in the folder with all the other X.org driver modules
- File verification:

- Verify that the device file `/dev/galcore` is present.
- Verify that the file `/etc/X11/xorg.conf` contains the correct entries as described in the previous section.

- Verify acceleration:
  - Assuming the above steps have been performed, do the following to verify that X Window System acceleration is indeed operating.
  - Examine the log file `/var/log/Xorg.0.log` and confirm that the following lines present:

```
[    33.767] (II) LoadModule: "vivante"
[    33.782] (II) Loading /usr/lib/xorg/modules/drivers/vivante_drv.so
...
[    33.881] (II) VIVANTE(0): using default device
[    33.881] (WW) VGA arbiter: cannot open kernel arbiter, no multi-card support
[    33.881] (II) VIVANTE(0): Creating default Display subsection in Screen section
    "Default Screen" for depth/fbbpp 16/16
[    33.881] (==) VIVANTE(0): Depth 16, (==) framebuffer bpp 16
[    33.881] (==) VIVANTE(0): RGB weight 565
[    33.881] (==) VIVANTE(0): Default visual is TrueColor
[    33.881] (==) VIVANTE(0): Using gamma correction (1.0, 1.0, 1.0)
[    33.881] (II) VIVANTE(0): hardware: mxc_elcdif_fb (video memory: 2250kB)
[    33.882] (II) VIVANTE(0): checking modes against framebuffer device...
[    33.882] (II) VIVANTE(0): checking modes against monitor...
[    33.882] (--) VIVANTE(0): Virtual size is 800x480 (pitch 800)
[    33.882] (**) VIVANTE(0):  Built-in mode "current": 33.5 MHz, 31.2 kHz, 58.6 Hz
[    33.882] (II) VIVANTE(0): Modeline "current"x0.0   33.50  800 964 974 1073  480
490 500 533 -hsync -vsync -csync (31.2 kHz)
[    33.882] (==) VIVANTE(0): DPI set to (96, 96)
...
[    34.228] (II) VIVANTE(0): FB Start = 0x333a8000  FB Base = 0x333a8000  FB
Offset = (nil)
[    34.228] (II) VIVANTE(0): test Initializing EXA
[    34.228] (II) EXA(0): Driver allocated offscreen pixmaps
[    34.229] (II) EXA(0): Driver registered support for the following operations:
[    34.229] (II)         Solid
[    34.229] (II)         Copy
[    34.229] (II)         Composite (RENDER acceleration)
[    34.229] (II)         UploadToScreen
[    34.244] (==) VIVANTE(0): Backing store disabled
[    34.244] (==) VIVANTE(0): DPMS enabled
```

Chapter 12
Camera Sensor Interface (CSI) Driver

The CSI driver enables the i.MX device to directly connect to external CMOS sensors and CCIR656 video sources. The CSI and sensor drivers are implemented in the Video for Linux Two (V4L2) driver framework. It consists of the image capture driver and the video output driver.

12.1 Hardware Operation

The CSI driver configures and operates with the hardware registers for the CSI module. It provides:

- Configurable interface logic to support most commonly available CMOS sensors.
- Full control of 8-bit/pixel, 10-bit/pixel or 16-bit/pixel data format to 32-bit receive FIFO packing.
- 128X32 FIFO to store received image pixel data.
- Receive FIFO overrun protection mechanism.
- Embedded DMA controllers to transfer data from receive FIFO or statistic FIFO through AHB bus.
- Support for double buffering two frames in the external memory.
- Single interrupt source to interrupt controller from maskable interrupt sources: Start of Frame, End of Frame, and so on.
- Configurable master clock frequency output to sensor.

For more information, see the CSI chapter in the *i.MX 6SoloLite Multimedia Applications Processor Reference Manual*.

12.2 Software Operation

This section includes the following:

- CSI Software Operation
- Video for Linux 2 (V4L2) APIs

12.2.1 CSI Software Operation

The CSI driver initializes the CSI interface. Applications use the V4L2 interface to operate the CSI interface.

12.2.2 Video for Linux 2 (V4L2) APIs

Video for Linux Two (V4L2) is a Linux standard. The API specification is available at http://v4l2spec.bytesex.org/spec/.

The V4L2 capture device includes the capture interface. The capture interface uses the CSI embedded DMA controller to implement the function. The V4L2 driver implements the standard V4L2 API for capture devices. The following is the data flow of capture.

1. The camera sends the data to the CSI receive FIFO, through the 8-bit/10-bit data port.
2. The embedded DMA controllers transfer data from the receive FIFO to external memory through the AHB bus.
3. The data is saved to the user space memory or exported to the frame buffer directly.

12.2.2.1 V4L2 Capture Device

V4L2 capture support can be selected during kernel configuration. The driver for this device is in the file under <ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture/csi_v4l2_capture.c

The memory map stream API is supported. Supported V4L2 IOCTLs include the following:

- VIDIOC_QUERYCAP
- VIDIOC_G_FMT
- VIDIOC_S_FMT
- VIDIOC_G_FBUF
- VIDIOC_S_FBUF
- VIDIOC_S_PARM
- VIDIOC_G_PARM
- VIDIOC_CROPCAP

- VIDIOC_S_CROP
- VIDIOC_G_CROP
- VIDIOC_QUERYBUF
- VIDIOC_REQBUFS
- VIDIOC_DQBUF
- VIDIOC_QBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF
- VIDIOC_ENUM_FMT
- VIDIOC_ENUM_FRAMESIZES
- VIDIOC_ENUM_FRAMEINTERVALS
- VIDIOC_DBG_G_CHIP_IDENT

12.2.2.2 Use of the V4L2 Capture APIs

The following is a sample process of using the V4L2 capture APIs:

1. Set the capture pixel format and size by using IOCTL VIDIOC_S_FMT.
2. Set the control information by using IOCTL VIDIOC_S_CTRL, for rotation.
3. Request a buffer by using IOCTL VIDIOC_REQBUFS. The common V4L2 driver creates a chain of buffers (currently the maximum number of frames is 10).
4. Memory maps the buffer to its user space.
5. Queue the buffer by using the IOCTL command VIDIOC_QBUF.
6. Start the stream by executing IOCTL VIDIOC_STREAMON.
7. Execute the IOCTL VIDIOC_DQBUF.
8. Pass the data that requires post-processing to the buffer.
9. Queue the buffer by using the IOCTL command VIDIOC_QBUF.
10. Go to step 7.
11. Stop the queuing by using the IOCTL command VIDIOC_STREAMOFF.

12.3 Source Code Structure

The following table shows the CSI sensor and V4L2 driver source files available in the directory: <ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture

Table 12-1. V4L2 and CSI Driver Files

File	Description
fsl_csi.c	CSI driver source file
fsl_csi.h	CSI driver header file

Table continues on the next page...

Table 12-1. V4L2 and CSI Driver Files (continued)

File	Description
csi_v4l2_capture.c	V4L2 capture device driver source file
mxc_v4l2_capture.h	V4L2 capture device driver header file

12.4 Linux Menu Configuration Options

The following Linux kernel configuration options are provided for this module.

- VIDEO_MXC_CSI_CAMERA: Includes support for the CSI Unit and V4L2 capture device. In menuconfig, this option is available under: Device Drivers > Multimedia support > Video capture adapters > MXC Video For Linux Camera > MXC Camera/ V4L2 PRP Features support

  By default, this option is enabled.
- MXC_CAMERA_OV5640: Option for the OV5640 sensor driver. In menuconfig, this option is available under: Device Drivers > Multimedia support > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support

  By default, this option is enabled.

12.5 Programming Interface

For more information, see the V4L2 Specification and the API Documents for the programming interface.

Chapter 13
OV5640 Using parallel interface

This is an introduction to the OV5640 camera driver that uses the parallel interface.

13.1 Hardware Operation

The OV5640 is a small camera sensor and lens module with low-power consumption. The camera driver is located under the Linux V4L2 architecture. and it implements the V4L2 capture interfaces. Applications cannot use the camera driver directly. Instead, the applications use the V4L2 capture driver to open and close the camera for preview and image capture, controlling the camera, getting images from camera, and starting the camera preview.

The OV5640 uses the serial camera control bus (SCCB) interface to control the sensor operation. It works as an I^2C client. V4L2 driver uses I^2C bus to control camera operation.

OV5640 supports only parallel interface.

Refer to OV5640 datasheet to get more information on the sensor.

Refer to the *i.MX 6 Multimedia Applications Processor Reference Manual* for more information on CSI.

13.2 Software Operation

The camera driver implements the V4L2 capture interface and applications and uses the V4L2 capture interface to operate the camera.

The supported operations of V4L2 capture are:

- Capture stream mode
- Capture still mode

The supported picture formats are:

- UYVY
- YUYV

The supported picture sizes are:
- QVGA
- VGA
- 720P
- 1080P
- QSXGA

13.3 Source Code Structure

Table below shows the camera driver source files available in the directory.

```
<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture.
```

Table 13-1. Camera Driver Files

File	Description
ov5640.c	Camera driver implementation for ov5640 using parallel interface

13.4 Linux Menu Configuration Options

The following Linux kernel configuration option is provided for this module.

To get to this option, use the ./ltib -c command when located in the <ltib dir>. On the displayed screen, select **Configure the Kernel** and exit. When the next screen appears, select the following option to enable this module:

- Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > OmniVision ov5640 camera support.

Chapter 14
Low-level Power Management (PM) Driver

14.1 Hardware Operation

This section describes the low-level Power Management (PM) driver which controls the low-power modes.

The i.MX 6 supports four low-power modes: RUN, WAIT, STOP, and DORMANT.

Table below lists the detailed clock information for different low-power modes.

Table 14-1. Low Power Modes

Mode	Core	Modules	PLL	CKIH/FPM	CKIL
RUN	Active	Active, Idle or Disable	On	On	On
WAIT	Disable	Active, Idle or Disable	On	On	On
STOP	Disable	Disable	Off	Off	On
DORMANT	Power off	Disable	Off	Off	On

For the detailed information about lower power modes, see the *i.MX 6SoloLite Multimedia Applications Processor Reference Manual (IMX6SLRM)*.

14.1.1 Software Operation

The i.MX 6 PM driver maps the low-power modes to the kernel power management states as follows:

- Standby: maps to STOP mode that offers significant power saving, as all blocks in the system are put into a low-power state, except for ARM core that is still powered on, and memory is placed in self-refresh mode to retain its contents.

- Mem (suspend to RAM): maps to DORMANT mode that offers most significant power saving as all blocks in the system are put into a low-power state, except for memory that is placed in self-refresh mode to retain its contents.
- System idle: maps to WAIT mode.

The i.MX 6 PM driver performs the following steps to enter and exit low-power mode:

1. Allow the Coretex-A9 platform to issue a deep-sleep mode request.
2. If it is in STOP or DORMANT mode:
    - Program CCM CLPCR register to set low-power control register.
    - If it is in DORMANT mode, request switching off CPU power when pdn_req is asserted.
    - Request switching off embedded memory peripheral power when pdn_req is asserted.
    - Program GPC mask register to unmask wakeup interrupts.
3. Call cpu_do_idle to execute WFI pending instructions for wait mode.
4. Execute mx6_do_suspend in IRAM.
5. If it is in DORMANT mode, save the ARM context, change the drive strength of MMDC PADs to "low" to minimize the power leakage in DDR PADs. Execute WFI pending instructions for stop mode.
6. Generate a wakeup interrupt and exit low-power mode. If it is in DORMANT mode, restore the ARM core and DDR drive strength.

In DORMANT and STOP mode, the i.MX 6 can assert the VSTBY signal to the PMIC and request a voltage change. The Machine Specific Layer (MSL) usually sets the standby voltage in STOP mode according to i.MX 6 data sheet.

14.1.2 Source Code Structure

Table below shows the PM driver source files. These files are available in <ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/.

Table 14-2. PM Driver Files

File	Description
pm.c	Supports suspend operation
system.c	Supports low-power modes
mx6_suspend.S	Assembly file for CPU suspend

14.1.3 Menu Configuration Options

The following Linux kernel configuration options are provided for this module. To get to these options, use the ./ltib -c command when located in the <ltib dir>. On the displayed screen, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

- CONFIG_PM builds support for power management. In menu configuration, this option is available under:
    - Power management options > Power Management support
    - By default, this option is Y.
- CONFIG_SUSPEND builds support for suspend. In menu configuration, this option is available under:
    - Power management options > Suspend to RAM and standby

14.1.4 Programming Interface

The mxc_cpu_lp_set API in the system.c function is provided for low-power modes. This implements all the steps required to put the system into WAIT and STOP modes.

14.1.5 Unit Test

To enter different system-level low-power modes:

```
echo mem > /sys/power/state
echo standby > /sys/power/state
```

To wake up system from low-power modes:

```
enable wake up source first, USB device, debug uart or RTC etc.
can be used as wakeup source, below is the example of uart wakeup and rtc wakeup:
echo enabled > /sys/devices/platform/imx-uart.'x'/tty/ttymxc'x'/power/wakeup; Here 'x' is
your debug uart's index;
echo +x > /sys/class/rtc/rtc0/wakealarm; RTC will wake up system after x seconds.
```

To test this mode automatically, refer to the script in /unit_tests/suspend_auto.sh

Chapter 15
PF100 Regulator Driver

15.1 Introduction

PF100 is a PMIC chip that is implemented on the i.MX 6 series development platforms.

The PF100 regulator driver provides the low-level control of the power supply regulators, selection of voltage levels, and enabling/disabling of regulators. This device driver makes use of the PF100 core driver to access the PF100 hardware control registers. The PF100 core driver is based on the MFD structure and it is attached to the kernel I2C bus.

15.2 Hardware Operation

PF100 provides reference and supply voltages for the application processor and peripheral devices.

Four buck (step down) converters (up to 6 independent output) and one boost (step up) converter are included. The buck converters provide the power supply to processor cores and to other low voltage circuits such as memory. Dynamic voltage scaling is provided to allow controlled supply rail adjustments for the processor cores and other circuitry.

Linear regulators are directly supplied from the battery or from the switchers, including supplies for I/O and peripherals, audio, camera, BT, and WLAN. Naming conventions are suggestive of typical or possible use case applications, but the switchers and regulators may be used for other system power requirements within the guidelines of specified capabilities.

The only power on event of PF100 is that PWRON is high, and the only power off event of PF100 is that PWRON is low. PMIC_ON_REQ pin of i.MX 6, which is controlled by SNVS block of i.MX 6, will connect with PWRON pin of PF100 to control PF100 on/off, so that system can power off.

15.2.1 Driver Features

The PF100 regulator driver is based on PF100 core driver and regulator core driver. It provides the following services for regulator control of the PMIC component:

- Switch ON/OFF all voltage regulators.
- Set the value for all voltage regulators.
- Get the current value for all voltage regulators.
- Write/Read PF100 registers by sysfs interface.

15.3 Software Operation

PF100 regulator client driver performs operations by reconfiguring the PMIC hardware control registers.

This is done by calling PF100 core driver APIs with the required register settings.

Some of the PMIC power management operations depend on the system design and configuration. For example, if the system is powered by a power source other than the PMIC, then turning off or adjusting the PMIC voltage regulators has no effect. Conversely, if the system is powered by the PMIC, any changes that use the power management driver and the regulator client driver can affect the operation or stability of the entire system.

15.3.1 Regulator APIs

The regulator power architecture is designed to provide a generic interface to voltage and current regulators within the Linux kernel.

It is intended to provide voltage and current control to client or consumer drivers and to provide status information to user space applications through a sysfs interface. The intention is to allow systems to dynamically control regulator output to save power and prolong battery life. This applies to both voltage regulators (where voltage output is controllable) and current sinks (where current output is controllable).

For more details, visit http://opensource.wolfsonmicro.com/node/15

Under this framework, most power operations can be done by the following unified API calls:

- `regulator_get` is an unified API call to lookup and obtain a reference to a regulator:

```
struct regulator *regulator_get(struct device *dev, const char *id);
```

- `regulator_put` is an unified API call to free the regulator source:

  ```
  void regulator_put(struct regulator *regulator, struct device *dev);
  ```
- `regulator_enable` is an unified API call to enable regulator output:

  ```
  int regulator_enable(struct regulator *regulator);
  ```
- `regulator_disable` is an unified API call to disable regulator output:

  ```
  int regulator_disable(struct regulator *regulator);
  ```
- `regulator_is_enabled` is the regulator output enabled:

  ```
  int regulator_is_enabled(struct regulator *regulator);
  ```
- `regulator_set_voltage` is an unified API call to set regulator output voltage:

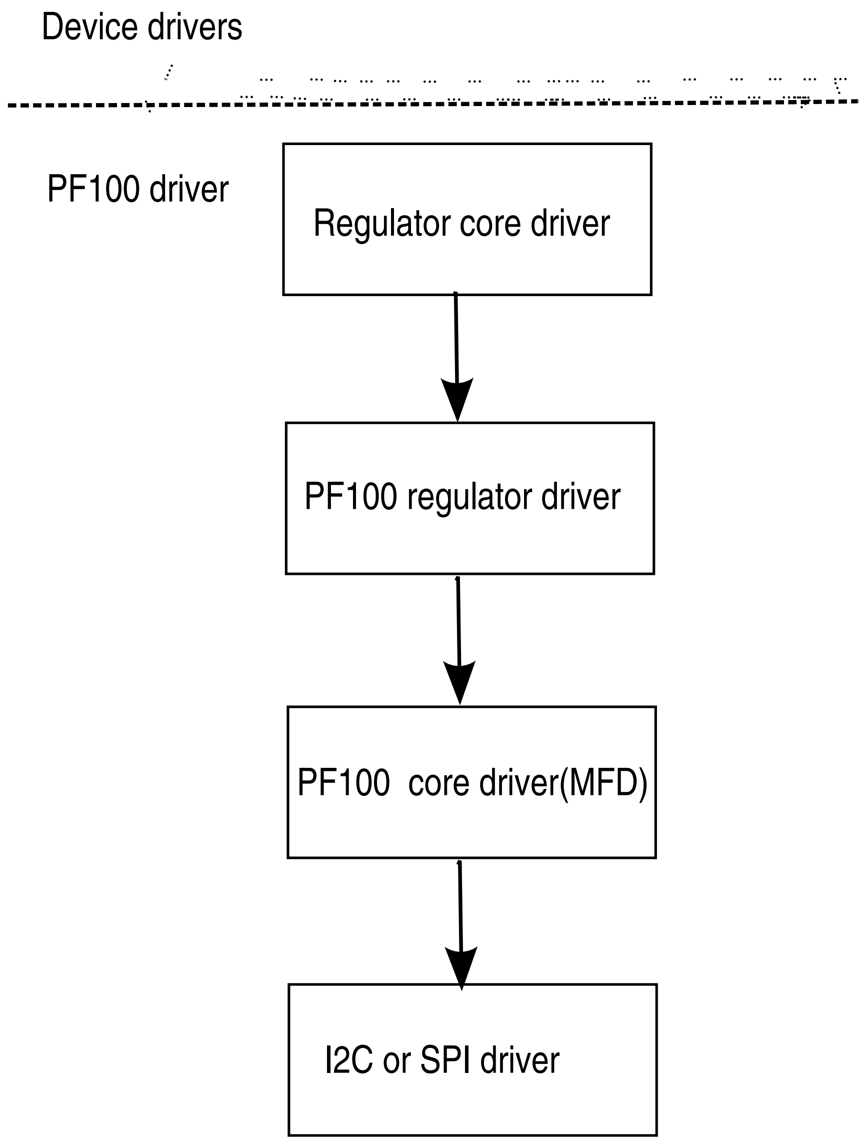
  ```
  int regulator_set_voltage(struct regulator *regulator, int uV);
  ```
- `regulator_get_voltage` is an unified API call to get regulator output voltage:

  ```
  int regulator_get_voltage(struct regulator *regulator);
  ```

You can find more APIs and details in the regulator core source code inside the Linux kernel at: `<ltib_dir>/rpm/BUILD/linux/drivers/regulator/core.c`.

15.4 Driver Architecture

Figure below shows the basic architecture of the PF100 regulator driver.



15.4.1 Driver Interface Details

Access to PF100 regulator is provided through the API of the regulator core driver.

The PF100 regulator driver provides the following regulator controls:

- Four buck switch regulators on normal mode (up to 6 independent rails): SW1AB, SW1C, SW2, SW3A, SW3B, and SW4.
- Buck switch can be programmed to a state of standby with specific register (PF100_SWxSTANDBY) in advance.
- Six Linear Regulators: VGEN1, VGEN2, VGEN3, VGEN4, VGEN5, and VGEN6.
- One LDO/Switch supply for VSNVS support on i.MX processors.
- One Low current, high accuracy, voltage reference for DDR Memory reference voltage.
- One Boost regulator with USB OTG support.
- Most power rails from PF100 have been programmed properly according to the hardware design. Therefore, you cannot find the kernel by using PF100 regulators. The PF100 regulator driver has implemented these regulators so that customers can use it freely if default PF100 value can't meet their hardware design.

15.4.2 Source Code Structure

The PF100 regulator driver is located in the regulator device driver directory:

```
<ltib_dir>/rpm/BUILD/linux/drivers/regulator
```

Table 15-1. PF100 core Driver Files

File	Description
`drivers/mfd/pf100-core.c`	Linux kernel interface for regulators.
`drivers/regulator/pf100-regulator.c`	Implementation of the PF100 regulator client driver.

The PF100 regulators for i.MX 6SoloLite EVK board are registered under <ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/mx6sl_evk_pmic_pf100.c.

15.4.3 Menu Configuration Options

The following are menu configuration options:

1. When located in the `<ltib dir>`, to get to the PMIC power configuration, use the command:

   ```
   ./ltib -c
   ```
2. On the configuration screen select Configure Kernel, and then exit. When the next screen appears, choose the following:

   Device Drivers > Voltage and Current regulator support > Support regulators on Freescale PF100 PMIC.

Chapter 16
CPU Frequency Scaling (CPUFREQ) Driver

16.1 Introduction

The CPU frequency scaling device driver allows the clock speed of the CPU to be changed on the fly. Once the CPU frequency is changed, the voltage VDDCORE, VDDSOC and VDDPU are changed to the voltage value defined in cpu_op-mx6.c . This method can reduce power consumption (thus saving battery power), because the CPU uses less power as the clock speed is reduced.

16.1.1 Software Operation

The CPUFREQ device driver is designed to change the CPU frequency and voltage on the fly.

If the frequency is not defined in cpu_op-mx6.c, the CPUFREQ driver changes the CPU frequency to the nearest higher frequency in the array. The frequencies are manipulated using the clock framework API, while the voltage is set using the regulators API. The CPU frequencies in the array are based on the boot CPU frequency which can be changed by using the clock command in U-Boot. By default, the conservative CPU frequency governor is used.

Refer to the API document for more information on the functions implemented in the driver.

To view what values the CPU frequency can be changed to in KHz (The values in the first column are the frequency values), use this command:

```
cat /sys/devices/system/cpu/cpu0/cpufreq/stats/time_in_state
```

To change the CPU frequency to a value that is given by using the command above (for example, to 792 MHz), use this command:

```
echo 792000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

The frequency 792000 is in KHz, which is 792 MHz.

The maximum frequency can be checked by using this command:

```
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_max_freq
```

Use the following command to view the current CPU frequency in KHz:

```
cat /sys/devices/system/cpu/cpu0/cpufreq/cpuinfo_cur_freq
```

Use the following command to view available governors:

```
cat /sys/devices/system/cpu/cpu0/cpufreq/scaling_available_governors
```

Use the following command to change to interactive CPU frequency governor:

```
echo interactive > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

16.1.2 Source Code Structure

Table below shows the source files and headers available in the following directory.

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/
```

Table 16-1. CPUFREQ Driver Files

File	Description
cpufreq.c	CPUFREQ functions

For CPU frequency working point settings, see arch/arm/mach-mx6/cpu_op-mx6.c.

16.2 Menu Configuration Options

The following Linux kernel configuration is provided for this module:

CONFIG_CPU_FREQ: In menu configuration, this option is located under: CPU Power Management > CPU Frequency scaling

The following options can be selected:

- CPU Frequency scaling
- CPU frequency translation statistics
- Default CPU frequency governor (conservative)
- Performance governor
- Powersave governor
- Userspace governor for userspace frequency scaling
- Interactive CPU frequency policy governor
- Conservative CPU frequency governor
- CPU frequency driver for i.MX CPUs

16.2.1 Board Configuration Options

There are no board configuration options for the CPUFREQ device driver.

Chapter 17
Dynamic Voltage Frequency Scaling (DVFS) Driver

17.1 Introduction

In order to improve power consumption, the Bus Frequency driver dynamically manages the various system frequencies.

The frequency changes are transparent to the higher layers and require no intervention from the drivers or middleware. Depending on activity of the peripheral devices and CPU loading, the bus frequency driver varies the DDR frequency between 24MHz and its maximum frequency. Similarly the AHB frequency is varied between 24MHz and 132MHz.

17.1.1 Operation

The Bus Frequency driver is part of the power management module in the Linux BSP. The main purpose of this driver is to scale the various operating frequency of the system clocks (like AHB, DDR, AXI etc) based on peripheral activity and CPU loading.

17.1.2 Software Operation

The bus frequency depends on the usecount of the various clocks in the system for its operation. Drivers enable/disable their clocks based on peripheral activity. Every peripheral is associated with a frequency setpoint. The bus frequency will set the system frequency to highest frequency setpoint based on the peripherals that are currently active.

The following setpoints are defined for all i.MX 6 platforms:

1. High Frequency Setpoint: AHB is at 132MHz, AXI is at 264Mhz and DDR is at the maximum frequency. This mode is used when most periphrals that need higher frequency for good performance are active. For ex, video playback, graphics processing etc.

2. Audio Playback setpoints : AHB is at 25MHz, AXI is at 50MHz and DDR is at 50MHz. This mode is used in audio playback mode.

3. Low Frequency setpoint: AHB is at 24MHz, AXI is at 24MHz and DDR is at 24MHz. This mode is used when the system is idle waiting for user input (display is off).

To Enable the bus frequency driver, use the following command:

```
echo 1 > /sys/devices/platform/imx_busfreq.0/enable
```

To Disable the bus frequency driver, use the following command:

```
echo 0 > /sys/devices/platform/imx_busfreq.0/enable
```

17.1.3 Source Code Structure

Table below lists the source files and headers available in the following directory:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6
```

Table 17-1. BusFrequency Driver Files

File	Description
bus_freq.c	Bus Frequency functions
`mx6_mmdc.c, mx6_ddr_freq.S, mx6sl_ddr.S`	DDR frequency change functions

17.2 Menu Configuration Options

There are no menu configuration options for this driver. The Bus Frequency drivers is included and enabled by default.

17.2.1 Board Configuration Options

There are no board configuration options for the Linux busfreq device driver.

Chapter 18
Thermal Driver

18.1 Introduction

Thermal driver is a necessary driver for monitoring and protecting the SoC. The thermal driver will monitor the SoC temperature in a certain frequency.

It defines three trip points: critical, hot, and active. Cooling device will take actions to protect the SoC according to different trip points that SoC has reached:

- When reaching critical point, cooling device will reset the system.
- When reaching hot point, cooling device will lower CPU frequency and notify GPU to run at a lower frequency.
- When the temperature drops to below active point, cooling device will release all the cooling actions.

Thermal driver has two parts:

- Thermal zone defines trip points and monitors the SoC's temperature.
- Cooling device takes the actions according to different trip points.

18.1.1 Thermal Driver Overview

The thermal driver implements the SoC temperature monitoring function and protection. It creates a system file interface of /sys/class/thermal/thermal_zone0/ for user. Internally, the thermal driver will monitor the SoC temperature and do necessary protection according to different trip points that SoC's temperature reaches.

18.2 Hardware Operation

The thermal driver uses an internal thermal sensor to monitor the SoC temperature. The cooling device uses the CPU frequency to protect the SoC.

All the related modules are in SoC.

18.2.1 Thermal Driver Software Operation

The thermal driver registers a thermal zone and a cooling device. The structure `thermal_zone_device_ops` describes the necessary interface that the thermal framework needs. The framework will call the related thermal zone interface to monitor the SoC temperature and do the cooling protection.

18.3 Driver Features

The thermal driver supports the following features:

- Thermal zone monitors the SoC temperature.
- Cooling device protects the SoC when the temperature reaches hot or critical points.

18.3.1 Source Code Structure

Table below shows the driver source files available in the directory:

<ltib_dir>/rpm/BUILD/linux/drivers/mxc/thermal

Table 18-1. Thermal Driver Files

File	Description
thermal.c	thermal zone driver source file
cooling.c	cooling device source file

18.3.2 Menu Configuration Options

The following Linux kernel configuration option is provided for this module. To get to this option, use the ./ltib -c command when located in the <ltib dir>. On the displayed screen, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

Device Drivers > MXC support drivers > ANATOP_THERMAL > Thermal Zone

18.3.3 Programming Interface

The thermal driver can be accessed through /sys/class/thermal/thermal_zone/.

18.3.4 Interrupt Requirements

The thermal driver uses irq #81. Set the alarm value to critical trip point. When the temperature exceeds the critical trip point, the interrupt handler will reset the system to protect SoC.

18.4 Unit Test

Modify the trip point's temperature through /sys/class/thermal/thermal_zone0/ trip_point_x_temp. Here, 'x' can be 0, 1 and 2, indicating critical, hot and active trip point, and the value of trip points should be critical > hot > active. Then run some program to make SoC in heavy loading. When the SoC temperature reaches the trip points, the thermal driver will take action to do some protections according to each trip point's mechanism. Restore the trip point's temperature. When SoC temperature drops to below active trip point, thermal driver will remove all the protections.

Chapter 19
Anatop Regulator Driver

19.1 Introduction

The Anatop regulator driver provides the low-level control of the power supply regulators, and selection of voltage levels.

This device driver makes use of the regulator core driver to access the Anatop hardware control registers.

19.1.1 Hardware Operation

The Power Management Unit on the die is built to simplify the external power interface and allow the die to be configured in a power appropriate manner. The power system consists of the input power sources and their characteristics, the integrated power transforming and controlling elements, and the final load interconnection and requirements.

Using seven LDO regulators, the number of external supplies is greatly reduced. If the backup coin and USB inputs are neglected, the number of external supplies is reduced to two. Missing from this external supply total are the necessary external supplies to power the desired memory interface. This will change depending on the type of external memory selected. Other supplies might also be necessary to supply the voltage to the different I/O power segments if their I/O voltage needs to be different than what is provided above.

Some internal regulators can be bypassed, so that external PMIC can supply these power directly to decrease power numer, such as VDD_SOC and VDD_ARM.

19.2 Driver Features

The Anatop regulator driver is based on regulator core driver. The following services are provided for regulator control:

- Switching ON/OFF all voltage regulators.
- Setting the value for all voltage regulators.
- Getting the current value for all voltage regulators.

19.2.1 Software Operation

The Anatop regulator client driver performs operations by reconfiguring the Anatop hardware control registers. This is done by calling regulator core APIs with the required register settings.

19.2.2 Regulator APIs

The regulator power architecture is designed to provide a generic interface to voltage and current regulators within the Linux 2.6 kernel. It is intended to provide voltage and current control to client or consumer drivers and also provide status information to user space applications through a sysfs interface. The intention is to allow systems to dynamically control regulator output to save power and prolong battery life. This applies to both voltage regulators (where voltage output is controllable) and current sinks (where current output is controllable).

For more details, visit http://opensource.wolfsonmicro.com/node/15

Under this framework, most power operations can be done by the following unified API calls:

- `regulator_get` used to lookup and obtain a reference to a regulator:
  - `struct regulator *regulator_get(struct device *dev, const char *id);`
- `regulator_put` used to free the regulator source:
  - `void regulator_put(struct regulator *regulator, struct device *dev);`
- `regulator_enable` used to enable regulator output:
  - `int regulator_enable(struct regulator *regulator);`
- `regulator_disable` used to disable regulator output:
  - `int regulator_disable(struct regulator *regulator);`
- `regulator_is_enabled` is the regulator output enabled:
  - `int regulator_is_enabled(struct regulator *regulator);`

- `regulator_set_voltage` used to set regulator output voltage:
  - `int regulator_set_voltage(struct regulator *regulator, int uV);`
- `regulator_get_voltage` used to get regulator output voltage:
  - `int regulator_get_voltage(struct regulator *regulator);`

For more APIs and details in the regulator core source code inside the Linux kernel, see: <ltib_dir>/rpm/BUILD/linux/drivers/regulator/core.c.

19.2.3 Driver Interface Details

Access to the Anatop regulator is provided through the API of the regulator core driver. The Anatop regulator driver provides the following regulator controls:

- Seven LDO regulators.
- All of the regulator functions are handled by setting the appropriate Anatop hardware register values. This is done by calling the regulator core APIs to access the Anatop hardware registers.

19.2.4 Source Code Structure

The Anatop regulator driver is located in the regulator device driver directory:

```
<ltib_dir>/rpm/BUILD/linux/drivers/regulator
```

Table 19-1. Anatop Power Management Driver Files

File	Description
`core.c`	Linux kernel interface for regulators.
`anatop-regulator.c`	Implementation of the Anatop regulator client driver

19.2.5 Menu Configuration Options

To get to the Anatop regulator configuration, use the command ./ltib -c when located in the <ltib dir>. On the configuration screen, select Configure Kernel, and then exit. The following Linux kernel configurations are provided for the Anatop Regulator driver:

- Device Drivers > Voltage and Current regulator support > Anatop Regulator Support.
- System Type > Freescale MXC Implementations > Internal LDO in i.MX 6Quad and i.MX 6DualLite bypass.

Chapter 20
SNVS Real Time Clock (SRTC) Driver

20.1 Introduction

The SNVS Real Time Clock (SRTC) module is used to keep the time and date. It provides a certifiable time to the user and can raise an alarm if tampering with counters is detected. The SRTC is composed of two sub-modules: Low power domain (LP) and High power domain (HP). The SRTC driver only supports the LP domain with low security mode.

20.1.1 Hardware Operation

The SRTC is a real-time clock with enhanced security capabilities.

It provides an accurate and constant time, regardless of the main system power state and without the need to use an external on-board time source, such as an external RTC. The SRTC can wake up the system when a preset alarm threshold is reached.

20.2 Software Operation

The following sections describe the software operation of the SRTC driver.

20.2.1 IOCTL

The SRTC driver complies with the Linux RTC driver model. See the Linux documentation in <ltib_dir>/rpm/BUILD/linux/Documentation/rtc.txt for information on the RTC API.

Besides the initialization function, the SRTC driver provides IOCTL functions to set up the RTC timers and alarm functions. The following RTC IOCTLs are implemented by the SRTC driver:

- RTC_RD_TIME
- RTC_SET_TIME
- RTC_AIE_ON
- RTC_AIE_OFF
- RTC_ALM_READ
- RTC_ALM_SET

The driver information can be access by the proc file system. For example:

```
root@freescale /unit_tests$ cat /proc/driver/rtc
rtc_time        : 12:48:29
rtc_date        : 2009-08-07
alrm_time       : 14:41:16
alrm_date       : 1970-01-13
alarm_IRQ       : no
alrm_pending    : no
24hr            : yes
```

20.2.2 Keeping Alive in the Power Off State

To preserve the time when the device is in the power-off state, the SRTC clock source should be set to CKIL and the voltage input, NVCC_SRTC_POW, should remain active. Usually these signals are connected to the PMIC and software can configure the PMIC registers to enable the SRTC clock source and power supply.

Generally, when the main battery is removed and the device is in power-off state, a coin-cell battery is used as a backup power supply. To avoid SRTC time loss, the voltage of the coin-cell battery should be sufficient to power the SRTC. If the coin-cell battery is chargeable, it is recommended to automatically enable the coin-cell charger so that the SRTC is properly powered.

20.3 Driver Features

The SRTC driver includes the following features:

- Implementing all the functions required by Linux to provide the real-time clock and alarm interrupt.
- Reserveing time in power-ff state.
- Alarm wakes up the system from low-power modes.

20.3.1 Source Code Structure

The RTC module is implemented in the following directory:

```
<ltib_dir>/rpm/BUILD/linux/drivers/rtc
```

Table below shows the RTC module files.

Table 20-1. RTC Driver Files

File	Description
`rtc-snvs.c`	SNVS RTC driver implementation file

The source file for the SRTC specifies the SRTC function implementations.

20.3.2 Menu Configuration Options

To get to the SRTC driver, use the command ./ltib -c when located in the <ltib dir>. On the displayed screen, select **Configure the kernel** and exit. When the next screen appears, select the following options to enable the SRTC driver:

- Device Drivers > Real Time Clock > Freescale SNVS Real Time Clock

Chapter 21
Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver

21.1 ALSA Sound Driver Introduction

The Advanced Linux Sound Architecture (ALSA), now the most popular architecture in Linux system, provides audio and MIDI functionality to the Linux operating system.

ALSA has the following significant features:

- Efficient support for all types of audio interfaces, from consumer sound cards to professional multichannel audio interfaces
- Fully modularized sound drivers
- SMP and thread-safe design
- User space library (alsa-lib) to simplify application programming and provide higher level functionality
- Support for the older Open Sound System (OSS) API, providing binary compatibility for most OSS programs

ALSA System on Chip (ASoC) layer is designed for SoC audio. The overall project goal of the ASoC layer provides better ALSA support for embedded system on chip processors and portable audio CODECs.

The ASoC layer also provides the following features:

- CODEC independence, allows reuse of CODEC drivers on other platforms and machines.
- Easy I2S/PCM audio interface setup between CODEC and SoC. Each SoC interface and CODEC registers its audio interface capabilities with the core.
- Dynamic Audio Power Management (DAPM). DAPM is an ASoC technology designed to minimize audio subsystem power consumption no matter what audio-use case is active. DAPM guarantees the lowest audio power state at all times and is completely transparent to user space audio components. DAPM is ideal for mobile devices or devices with complex audio requirements.

- Pop and click reduction. Pops and clicks can be reduced by powering the CODEC up/down in the correct sequence (including using digital mute). ASoC signals the CODEC when to change power states.
- Machine specific controls, allows machines to add controls to the sound card, for example, volume control for speaker amp.

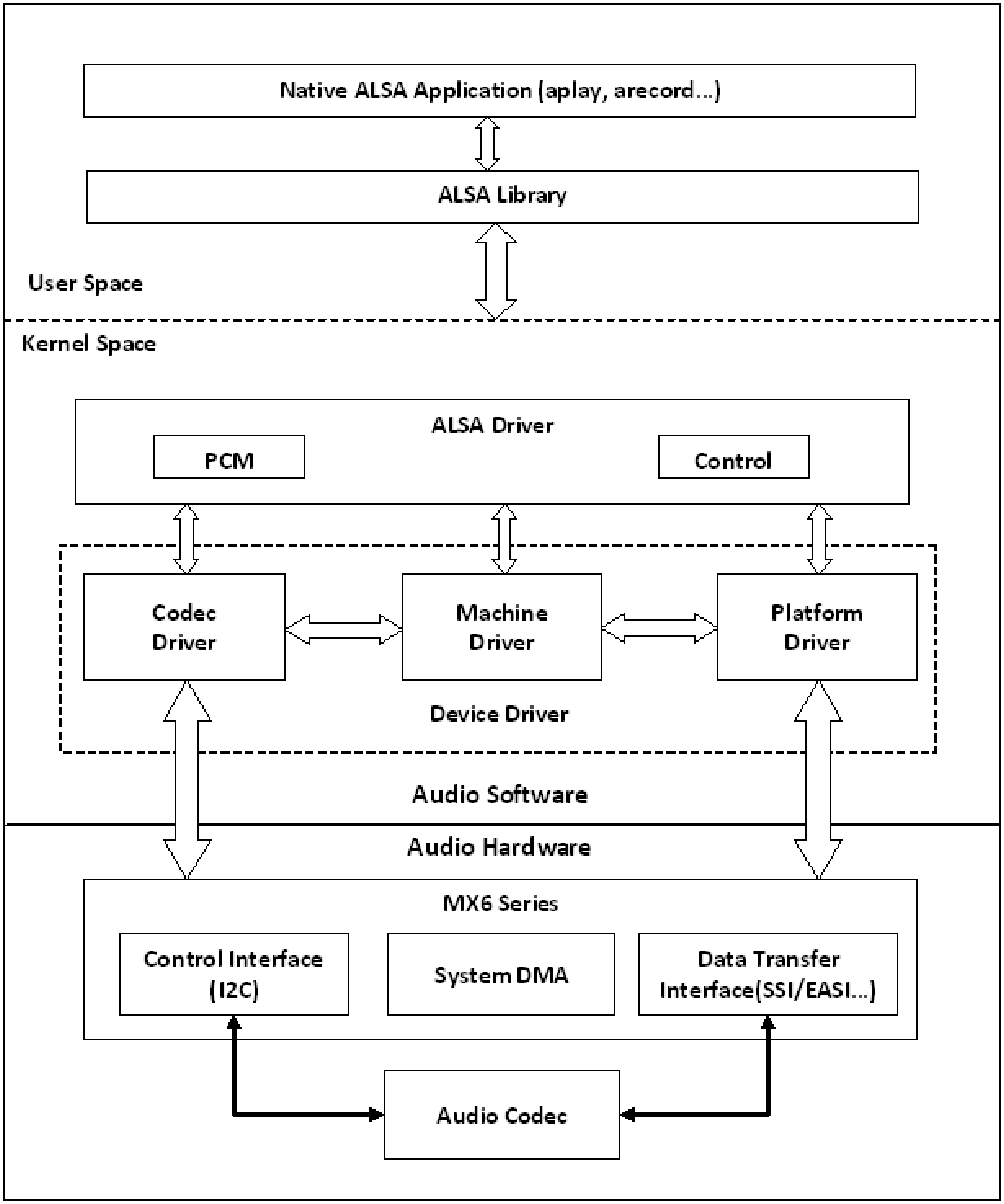


Figure 21-1. ALSA SoC Software Architecture

ASoC basically splits an embedded audio system into 3 components:

- Machine driver-handles any machine specific controls and audio events, such as turning on an external amp at the beginning of playback.
- Platform driver-contains the audio DMA engine and audio interface drivers (for example, I^2S, AC97, PCM) for that platform.
- CODEC driver-platform independent and contains audio controls, audio interface capabilities, the CODEC DAPM definition, and CODEC I/O functions.

More detailed information about ASoC can be found in the Linux kernel documentation in the linux source tree at linux/Documentation/sound/alsa/soc and at http://www.alsa-project.org/main/index.php/ASoC.

21.2 SoC Sound Card

Currently, the stereo CODEC (wm8962) drivers are implemented by using SoC architecture.

These sound card drivers are built in independently. The stereo sound card supports stereo playback and capture.

21.2.1 Stereo CODEC Features

The stereo CODEC supports the following features:

- Sample rates for playback and capture are 8KHz, 32 KHz, 44.1 KHz, 48 KHz, and 96 KHz
- Channels:
  - Playback: supports two channels.
  - Capture: supports two channels.
- Audio formats:
  - Playback:
    - SNDRV_PCM_FMTBIT_S16_LE
    - SNDRV_PCM_FMTBIT_S20_3LE
    - SNDRV_PCM_FMTBIT_S24_LE
  - Capture:
    - SNDRV_PCM_FMTBIT_S16_LE
    - SNDRV_PCM_FMTBIT_S20_3LE
    - SNDRV_PCM_FMTBIT_S24_LE

21.2.2 Sound Card Information

The following is the registered sound card information, using the commands aplay -l and arecord -l. For example, the stereo sound card is registered as card 0.

```
root@freescale /$ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: wm8962audio [wm8962-audio], device 0: HiFi wm8962-0 []
 Subdevices: 1/1
 Subdevice #0: subdevice #0
```

21.3 Hardware Operation

The following sections describe the hardware operation of the ASoC driver.

21.3.1 Stereo Audio CODEC

The stereo audio CODEC is controlled by the I^2C interface. The audio data is transferred from the user data buffer to/from the SSI FIFO through the DMA channel. The DMA channel is selected according to the audio sample bits. AUDMUX is used to set up the path between the SSI port and the output port which connects with the CODEC. The CODEC works in master mode and provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate.

The WM8962 ASoC CODEC driver exports the audio record/playback/mixer APIs according to the ASoC architecture.

The CODEC driver is generic and hardware independent code that configures the CODEC to provide audio capture and playback. It does not contain code that is specific to the target platform or machine. The CODEC driver handles:

- CODEC DAI and PCM configuration
- CODEC control I/O-using I^2C
- Mixers and audio controls
- CODEC audio operations
- DAC Digital mute control

The WM8962 CODEC is registered as an I^2C client when the module initializes. The APIs are exported to the upper layer by the structure snd_soc_dai_ops .

Headphone insertion/removal can be detected through a GPIO interrupt signal.

SSI dual FIFO features are enabled by default.

21.4 Software Operation

The following sections describe the software operation of the ASoC driver.

21.4.1 ASoC Driver Source Architecture

File imx-pcm-dma-mx2.c is shared by the stereo ALSA SoC driver, the 7.1 ALSA SoC driver and other CODEC driver. This file is responsible for preallocating DMA buffers and managing DMA channels.

The stereo CODEC is connected to the CPU through the SSI interface. imx-ssi.c registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip SSI interface. wm8962.c registers the stereo CODEC and hifi DAI drivers. The direct hardware operations on the stereo codec are in wm8962.c. imx-wm8962.c is the machine layer code which creates the driver device and registers the stereo sound card.

The following table shows the stereo CODEC SoC driver source files. These files are under the <ltib_dir>/rpm/BUILD/linux/sound/soc directory.

Table 21-1. Stereo Codec SoC Driver Files

File	Description
imx/imx-wm8962.c	Machine layer for stereo CODEC ALSA SoC
imx/imx-pcm-dma-mx2.c	Platform layer for stereo CODEC ALSA SoC
imx/imx-pcm.h	Header file for PCM driver and AUDMUX register definitions
imx/imx-ssi.c	Platform DAI link for stereo CODEC ALSA SoC
imx/imx-ssi.h	Header file for platform DAI link and SSI register definitions
codecs/wm8962.c	CODEC layer for stereo CODEC ALSA SoC
codecs/wm8962.h	Header file for stereo CODEC driver

21.4.2 Sound Card Registration

The CODECs have the same registration sequence:

1. The CODEC driver registers the CODEC driver, DAI driver, and their operation functions.

2. The platform driver registers the PCM driver, CPU DAI driver and their operation functions, pre-allocates buffers for PCM components and sets playback and capture operations as applicable.
3. The machine layer creates the DAI link between CODEC and CPU registers the sound card and PCM devices.

21.4.3 Device Open

The ALSA driver performs the following functions:

- Allocates a free substream for the operation to be performed.
- Opens the low-level hardware device.
- Assigns the hardware capabilities to ALSA runtime information (the runtime structure contains all the hardware, DMA, and software capabilities of an opened substream).
- Configures DMA read or write channel for operation.
- Configures CPU DAI and CODEC DAI interface.
- Configures CODEC hardware.
- Triggers the transfer.

After triggering for the first time, the subsequent DMA read/write operations are configured by the DMA callback.

21.4.4 Platform Data

struct mxc_audio_platform_data defined in include/linux/fsl_devices.h is used to pass the platform data of audio CODEC.

The value of platform data needs to be updated according to Hardware design.

Take wm8962 CODEC platform data as a example to show the parameter of mxc_audio_platform_data. See header file for the details of more variables.

- `ssi_num` indicates which SSI channel is used.
- `src_port` indicates which AUDMUX port is connected with SSI.
- `ext_port` indicates which AUDMUX port is connected with external audio CODEC.
- `hp_gpio`: The IRQ line used for headphone detection.
- `hp_active_low`: When headphone is inserted, the detection pin status. If pin voltage level is low, the value should be 1.
- `mic_gpio`: The IRQ line used for micphone detection
- `mic_active_low`: When micphone is inserted, the detection pin status, if pin voltage level is low, the value should be 1.

- `init`: The callback function to initialize audio CODEC. For example, configure the clock of audio CODEC.
- `clock_enable`: The callback function to enable or disable clock for audio CODEC.

21.4.5 Menu Configuration Options

The following Linux kernel configuration options are provided for this module:

To get to these options, use the ./ltib -c command when located in the <ltib dir>. Select **Configure the Kernel** on the displayed screen and exit. When the next screen appears, select the following options to enable this module:

- SoC Audio supports for wm8962 CODEC. In menu configuration, this is option is available under Device drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio for the Freescale i.MX CPU, SoC Audio support for WM8962

21.5 Unit Test

This section shows how to use ALSA driver, and assume the rootfs is GNOME.

21.5.1 Stereo CODEC Unit Test

Stereo CODEC driver supports playback and record features. There are a default volume, and you may adjust volume by alsamixer command.

Playback feature may be tested by the following command:

- aplay [-Dplughw:0,0] audio.wav

Because analog micphone is connected to IN3R port of WM8962 CODEC, the following amixer commands are needed to input into command line for enabling analog micphone.

- amixer sset 'MIXINR IN3R' on
- amixer sset 'INPGAR IN3R' on

The recording feature may be tested by the following command:

- arecord [-Dplughw:0,0] -r 44100 -f S16_LE -c 2 -d 5 record.wav

More usage for aplay/arecord/amixer may be obtained by the following commands.

- aplay --h
- arecord --h
- amixer --h

Chapter 22 SPI NOR Flash Memory Technology Device (MTD) Driver

22.1 Introduction

The SPI NOR Flash Memory Technology Device (MTD) driver provides the support to the data Flash though the SPI interface.

By default, the SPI NOR Flash MTD driver creates static MTD partitions to support data Flash. If RedBoot partitions exist, they have higher priority than static partitions, and the MTD partitions can be created from the RedBoot partitions.

22.1.1 Hardware Operation

On some boards, the SPI NOR - AT45DB321D is equipped, while on some boards M25P32 is equipped. Check the SPI NOR type on the boards and then configure it properly.

The AT45DB321D is a 2.7 V, serial-interface sequential access Flash memory. The AT45DB321D serial interface is SPI compatible for frequencies up to 66 MHz. The memory is organized as 8,192 pages of 512 bytes or 528 bytes. The AT45DB321D also contains two SRAM buffers of 512/528 bytes each which allow receiving of data while a page in the main memory is being reprogrammed, as well as writing a continuous data stream.

The M25P32 is a 32 Mbit (4M x 8) Serial Flash memory, with advanced write protection mechanisms, accessed by a high speed SPI-compatible bus up to 75MHz. The memory is organized as 64 sectors, each containing 256 pages. Each page is 256 bytes wide. Thus, the whole memory can be viewed as consisting of 16384 pages, or 4,194,304 bytes. The memory can be programmed 1 to 256 bytes at a time using the Page Program instruction. The whole memory can be erased using the Bulk Erase instruction, or a sector at a time, using the Sector Erase instruction.

Unlike conventional Flash memories that are accessed randomly, these two SPI NOR access data sequentially. They operate from a single 2.7-3.6 V power supply for program and read operations. They are enabled through a chip select pin and accessed through a three-wire interface: Serial Input, Serial Output, and Serial Clock.

22.1.2 Software Operation

In a Flash-based embedded Linux system, a number of Linux technologies work together to implement a file system. Figure below illustrates the relationships between some of the standard components.

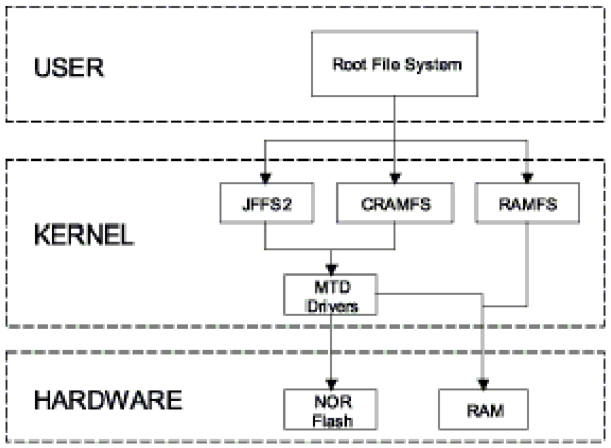


Figure 22-1. Components of a Flash-Based File System

The MTD subsystem for Linux is a generic interface to memory devices, such as Flash and RAM, providing simple read, write, and erase access to physical memory devices. Devices called mtdblock devices can be mounted by JFFS, JFFS2 and CRAMFS file systems. The SPI NOR MTD driver is based on the MTD data Flash driver in the kernel by adding SPI access. In the initialization phase, the SPI NOR MTD driver detects a data Flash by reading the JEDEC ID. Then the driver adds the MTD device. The SPI NOR MTD driver also provides the interfaces to read, write, and erase NOR Flash.

22.1.3 Driver Features

This NOR MTD implementation supports the following features:

- Provides necessary information for the upper layer MTD driver

22.1.4 Source Code Structure

The SPI NOR MTD driver is implemented in the following directory:

<ltib_dir>/rpm/BUILD/linux/drivers/mtd/devices/

Table below shows the driver files:

Table 22-1. SPI NOR MTD Driver Files

File	Description
m25p80.c	Source file

22.1.5 Menu Configuration Options

To get to the SPI NOR MTD driver, use the command ./ltib -c when located in the <ltib dir>. On the screen displayed, select Configure the kernel and exit. When the next screen appears select the following options to enable the SPI NOR MTD driver accordingly:

- CONFIG_MTD_M25P80: This config enables access to most modern SPI flash chips, used for program and data storage.
- Device Drivers > Memory Technology Device (MTD) support >Self-contained MTD device drivers > Support most SPI Flash chips (AT26DF, M25P, W25X, ...)

Chapter 23
MMC/SD/SDIO Host Driver

23.1 Introduction

The MultiMediaCard (MMC)/ Secure Digital (SD)/ Secure Digital Input Output (SDIO) Host driver implements a standard Linux driver interface to the ultra MMC/SD host controller (uSDHC) .

The host driver is part of the Linux kernel MMC framework.

The MMC driver has the following features:

- 1-bit or 4-bit operation for SD3.0 and SDIO 2.0 cards (so far we support SDIO v2.0 (AR6003 is verified)).
- Supports card insertion and removal detections.
- Supports the standard MMC commands.
- PIO and DMA data transfers.
- Power management.
- Supports 1/4/8-bit operations for MMC cards.
- Support eMMC4.4 SDR and DDR modes.
- Support SD3.0 SDR50 and SDR104 modes.

23.1.1 Hardware Operation

The MMC communication is based on an advanced 11-pin serial bus designed to operate in a low voltage range. The uSDHC module supports MMC along with SD memory and I/O functions. The uSDHC controls the MMC, SD memory, and I/O cards by sending commands to cards and performing data accesses to and from the cards. The SD memory card system defines two alternative communication protocols: SD and SPI. The uSDHC only supports the SD bus protocol.

The uSDHC command transfer type and uSDHC command argument registers allow a command to be issued to the card. The uSDHC command, system control, and protocol control registers allow the users to specify the format of the data and response and to control the read wait cycle.

There are four 32-bit registers used to store the response from the card in the uSDHC. The uSDHC reads these four registers to get the command response directly. The uSDHC uses a fully configurable 128x32-bit FIFO for read and write. The buffer is used as temporary storage for data being transferred between the host system and the card, and vice versa. The uSDHC data buffer access register bits hold 32-bit data upon a read or write transfer.

For receiving data, the steps are as follows:

1. The uSDHC controller generates a DMA request when there are more words received in the buffer than the amount set in the RD_WML register
2. Upon receiving this request, DMA engine starts transferring data from the uSDHC FIFO to system memory by reading the data buffer access register.

For transmitting data, the steps are as follows:

1. The uSDHC controller generates a DMA request whenever the amount of the buffer space exceeds the value set in the WR_WML register.
2. Upon receiving this request, the DMA engine starts moving data from the system memory to the uSDHC FIFO by writing to the Data Buffer Access Register for a number of pre-defined bytes.

The read-only uSDHC Present State and Interrupt Status Registers provide uSDHC operations status, application FIFO status, error conditions, and interrupt status.

When certain events occur, the module has the ability to generate interrupts as well as set the corresponding Status Register bits. The uSDHC interrupt status enable and signal-enable registers allow the user to control if these interrupts occur.

23.1.2 Software Operation

The Linux OS contains an MMC bus driver which implements the MMC bus protocols. The MMC block driver handles the file system read/write calls and uses the low level MMC host controller interface driver to send the commands to the uSDHC.

The MMC driver is responsible for implementing standard entry points for `init`, `exit`, `request`, and `set_ios`. The driver implements the following functions:

- The `init` function `esdhc_pltfm_init()` initializes the platform hardware and set platform dependant flags or values to sdhci_host structure.
- The `exit` function `esdhc_pltfm_exit()` deinitializes the platform hardware and frees the memory allocated.
- The function `esdhc_pltfm_get_max_clock()` gets the maximum SD bus clock frequency supported by the platform.
- The function `esdhc_pltfm_get_min_clock()` gets the minimum SD bus clock frequency supported by the platform.
- `esdhc_pltfm_get_ro()` gets the card read only status.
- `plt_8bit_width()` handles 8 bit mode switching on the platform.
- `plt_clk_ctrl()` handles clock management on the platform.
- `esdhc_prepare_tuning()` handles the preparation for tuning. It's only used for SD3.0 UHS-I mode.
- `esdhc_post_tuning()` handles the post operation for tuning.
- `esdhc_set_clock()` handles the clock change request.
- `cd_irq()` it's the interrupt routine for card detect.

Figure below shows how the MMC-related drivers are layered.

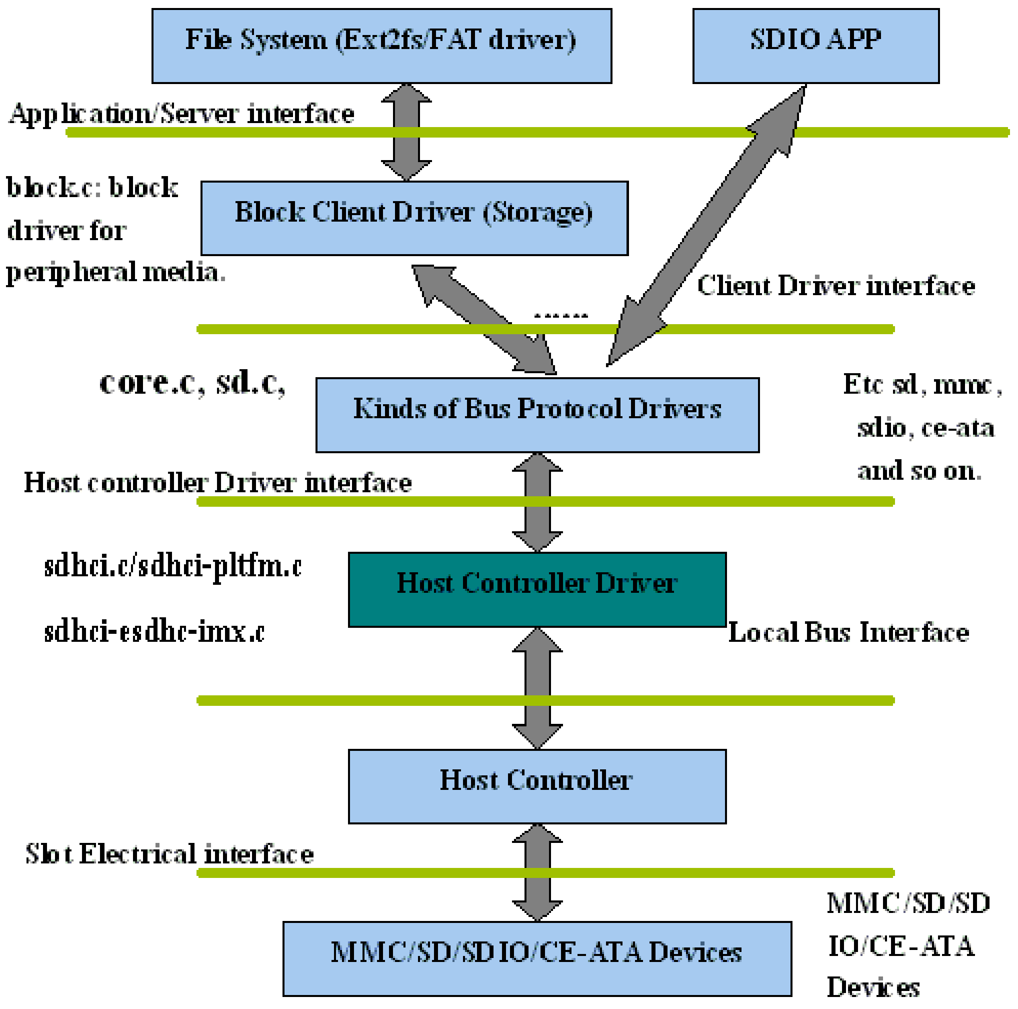

File System (Ext2fs/FAT driver)
SDIO APP
Application/Server interface
block.c: block driver for peripheral media.
Block Client Driver (Storage)
Client Driver interface
......
core.c, sd.c,
Kinds of Bus Protocol Drivers
Etc sd, mmc, sdio, ce-ata and so on.
Host controller Driver interface
sdhci.c/sdhci-pltfm.c
sdhci-esdhc-imx.c
Host Controller Driver
Local Bus Interface
Host Controller
Slot Electrical interface
MMC/SD/SD IO/CE-ATA Devices
MMC/SD/SD IO/CE-ATA Devices

Figure 23-1. MMC Drivers Layering

23.2 Driver Features

The MMC driver supports the following features:

- Supports multiple uSDHC modules.
- Provides all the entry points to interface with the Linux MMC core driver.
- MMC and SD cards.
- SDIO cards.
- SD3.0 cards.
- Recognizes data transfer errors such as command time outs and CRC errors.
- Power management.
- It supports to be built as loadable or builtin module

23.2.1 Source Code Structure

Table below shows the uSDHC source files available in the source directory: <ltib_dir>/rpm/BUILD/linux/drivers/mmc/host/.

Table 23-1. uSDHC Driver Files MMC/SD Driver Files

File	Description
sdhci.c	sdhci standard stack code
sdhci-pltfm.c	sdhci platform layer
sdhci-esdhc-imx.c	uSDHC driver
sdhci-esdhc.h	uSDHC driver header file

23.2.2 Menu Configuration Options

The following Linux kernel configuration options are provided for this module.

To get to these options, use the ./ltib -c command when located in the <ltib dir>. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

- CONFIG_MMC builds support for the MMC bus protocol. In menuconfig, this option is available under:
  - Device Drivers > MMC/SD/SDIO Card support
  - By default, this option is Y.
- CONFIG_MMC_BLOCK builds support for MMC block device driver which can be used to mount the file system. In menuconfig, this option is available under:
  - Device Drivers > MMC/SD Card Support > MMC block device driver
  - By default, this option is Y.
- CONFIG_MMC_SDHCI_ESDHC_IMX is used for the i.MX USDHC ports. In menuconfig, this option is found under:
  - Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller

  To compile SDHCI driver as a loadable module, several options should be selected as indicated below:
  - CONFIG_MMC_SDHCI=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support

    - CONFIG_MMC_SDHCI_PLTFM=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus.
    - CONFIG_MMC_SDHCI_ESDHC_IMX=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller

  To compile SDHCI driver as a builttin module, several options should be selected as indicated below:

    - CONFIG_MMC_SDHCI=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support
    - CONFIG_MMC_SDHCI_PLTFM=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus.
    - CONFIG_MMC_SDHCI_ESDHC_IMX=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller
- CONFIG_MMC_UNSAFE_RESUME is used for embedded systems which use a MMC/SD/SDIO card for rootfs. In menuconfig, this option is found under:
    - Device drivers > MMC/SD/SDIO Card Support > Assume MMC/SD cards are non-removable.

23.2.3 Platform Data

struct esdhc_platform_data defined in arch/arm/plat-mxc/include/mach/esdhc.h is used to pass platform informaton:

- .wp_gpio: GPIO used for write protect detection
- .cd_gpio: GPIO used for card detection
- .always_present: 1 indicates the card is inserted and non-removable, and the card detect is ignored
- .support_18v: indicate the board could provide 1.8v power to the card.
- .support_8bit: indicate 8 data pins are connected to the card slot.
- .platform_pad_change: callback function used to change the pad settings due to different SD bus clock frequency
- .keep_power_at_suspend: keep MMC/SD slot power when system enters suspend
- .delay_line: delay line setting for DDR mode

23.2.4 Programming Interface

This driver implements the functions required by the MMC bus protocol to interface with the i.MX uSDHC module.

See the Linux document generated from build: make htmldocs.

23.2.5 Loadable Module Operations

The SDHCI driver can be built as loadable or builtin module.

1. How to build SDHCI driver as loadable module.
    - CONFIG_MMC_SDHCI=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support
    - CONFIG_MMC_SDHCI_PLTFM=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus.
    - CONFIG_MMC_SDHCI_ESDHC_IMX=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller
2. How to load and unload SDHCI module.

    Due to dependency, please load or unload the module following the module sequence shown below.

    run the following commands to load module:
    - load modules via insmod command, assuming the files of sdhci.ko and sdhci-platform.ko exist in current directory.

    ```
    $> insmod sdhci.ko
    $> insmod sdhci-platform.ko
    ```
    - load modules via modprobe command, please make sure the files of sdhci.ko and sdhci-platform.ko exist in corresponding kernel module lib directory.

    ```
    $> modprobe sdhci.ko
    $> modprobe sdhci-platform.ko
    ```

    run the following commands to unload module.:
    - unload modules via insmod command.

    ```
    $> rmsmod sdhci-platform
    $> rmsmod sdhci
    ```
    - unload modules via modprobe command.

```
$> modprobe -r sdhci-platform
$> modprobe -r sdhci
```

Chapter 24
Inter-IC (I2C) Driver

24.1 Introduction

I2C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices.

The I2C driver for Linux has two parts:

- I2C bus driver-low level interface that is used to talk to the I2C bus
- I2C chip driver-acts as an interface between other device drivers and the I2C bus driver

24.1.1 I2C Bus Driver Overview

The I2C bus driver is invoked only by the I2C chip driver and is not exposed to the user space.

The standard Linux kernel contains a core I2C module that is used by the chip driver to access the I2C bus driver to transfer data over the I2C bus. The chip driver uses a standard kernel space API that is provided in the Linux kernel to access the core I2C module. The standard I2C kernel functions are documented in the files available under Documentation/i2c in the kernel source tree. This bus driver supports the following features:

- Compatible with the I2C bus standard
- Bit rates up to 400 Kbps
- Starts and stops signal generation/detection
- Acknowledge bit generation/detection
- Interrupt-driven, byte-by-byte data transfer
- Standard I2C master mode

24.1.2 I2C Device Driver Overview

The I2C device driver implements all the Linux I2C data structures that are required to communicate with the I2C bus driver. It exposes a custom kernel space API to the other device drivers to transfer data to the device that is connected to the I2C bus. Internally, these API functions use the standard I2C kernel space API to call the I2C core module. The I2C core module looks up the I2C bus driver and calls the appropriate function in the I2C bus driver to transfer data. This driver provides the following functions to other device drivers:

- Read function to read the device registers
- Write function to write to the device registers

The camera driver uses the APIs provided by this driver to interact with the camera.

24.1.3 Hardware Operation

The I2C module provides the functionality of a standard I2C master and slave.

It is designed to be compatible with the standard Philips I2C bus protocol. The module supports up to 64 different clock frequencies that can be programmed by setting a value to the Frequency Divider Register (IFDR). It also generates an interrupt when one of the following occurs:

- One byte transfer is completed
- Address is received that matches its own specific address in slave-receive mode
- Arbitration is lost

24.2 Software Operation

The I2C driver for Linux has two parts: an I2C bus driver and an I2C chip driver.

24.2.1 I2C Bus Driver Software Operation

The I2C bus driver is described by a structure called i2c_adapter. The most important field in this structure is struct i2c_algorithm *algo. This field is a pointer to the i2c_algorithm structure that describes how data is transferred over the I2C bus. The algorithm structure contains a pointer to a function that is called whenever the I2C chip driver wants to communicate with an I2C device.

During startup, the I2C bus adapter is registered with the I2C core when the driver is loaded. Certain architectures have more than one I2C module. If so, the driver registers separate i2c_adapter structures for each I2C module with the I2C core. These adapters are unregistered (removed) when the driver is unloaded.

After transmitting each packet, the I2C bus driver waits for an interrupt indicating the end of a data transmission before transmitting the next byte. It times out and returns an error if the transfer complete signal is not received. Because the I2C bus driver uses wait queues for its operation, other device drivers should be careful not to call the I2C API methods from an interrupt mode.

24.2.2 I2C Device Driver Software Operation

The I2C driver controls an individual I2C device on the I2C bus. A structure, i2c_driver, describes the I2C chip driver. The fields of interest in this structure are flags and attach_adapter. The flags field is set to a value I2C_DF_NOTIFY so that the chip driver can be notified of any new I2C devices, after the driver is loaded. The attach_adapter callback function is called whenever a new I2C bus driver is loaded in the system. When the I2C bus driver is loaded, this driver stores the i2c_adapter structure associated with this bus driver so that it can use the appropriate methods to transfer data.

24.3 Driver Features

The I2C driver supports the following features:

- I2C communication protocol
- I2C master mode of operation

NOTE

The I2C driver does not support the I2C slave mode of operation.

24.3.1 Source Code Structure

Table below shows the I2C bus driver source files available in the directory:

<ltib_dir>/rpm/BUILD/linux/drivers/i2c/busses.

Table 24-1. I2C Bus Driver Files

File	Description
i2c-imx.c	I2C bus driver source file

24.3.2 Menu Configuration Options

To get to the Linux kernel configuration option provided for this module, use the ./ltib -c command when located in the <ltib dir>.

On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

Device Drivers > I2C support > I2C Hardware Bus support > IMX I2C interface.

24.3.3 Programming Interface

The I2C device driver can use the standard SMBus interface to read and write the registers of the device connected to the I2C bus.

For more information, see <ltib_dir>/rpm/BUILD/linux/include/linux/i2c.h.

24.3.4 Interrupt Requirements

The I2C module generates many kinds of interrupts.

The highest interrupt rate is associated with the transfer complete interrupt as shown in table below.

Table 24-2. I2C Interrupt Requirements

Parameter	Equation	Typical	Best Case
Rate	Transfer Bit Rate/8	25,000/sec	50,000/sec
Latency	8/Transfer Bit Rate	40 Î¼s	20 Î¼s

The typical value of the transfer bit-rate is 200 Kbps. The best case values are based on a baud rate of 400 Kbps (the maximum supported by the I2C interface).

Chapter 25
Enhanced Configurable Serial Peripheral Interface (ECSPI) Driver

25.1 Introduction

The ECSPI driver implements a standard Linux driver interface to the ECSPI controllers.

It supports the following features:

- Interrupt-driven transmit/receive of bytes
- Multiple master controller interface
- Multiple slaves select
- Multi-client requests

25.1.1 Hardware Operation

ECSPI is used for fast data communication with fewer software interrupts than conventional serial communications.

Each ECSPI is equipped with a data FIFO and is a master/slave configurable serial peripheral interface module, allowing the processor to interface with external SPI master or slave devices.

The primary features of the ECSPI includes:

- Master/slave-configurable
- Four chip select signals to support multiple peripherals
- Up to 32-bit programmable data transfer
- 64 x 32-bit FIFO for both transmit and receive data
- Configurable polarity and phase of the Chip Select (SS) and SPI Clock (SCLK)

25.2 Software Operation

The following sections describe the ECSPI software operation.

25.2.1 SPI Sub-System in Linux

The ECSPI driver layer is located between the client layer (SPI-NOR Flash are examples of clients) and the hardware access layer. Figure below shows the block diagram for SPI subsystem in Linux.

The SPI requests go into I/O queues. Requests for a given SPI device are executed in FIFO order and they complete asynchronously through completion callbacks. There are also some simple synchronous wrappers for those calls including the ones for common transaction types such as writing a command and then reading its response.

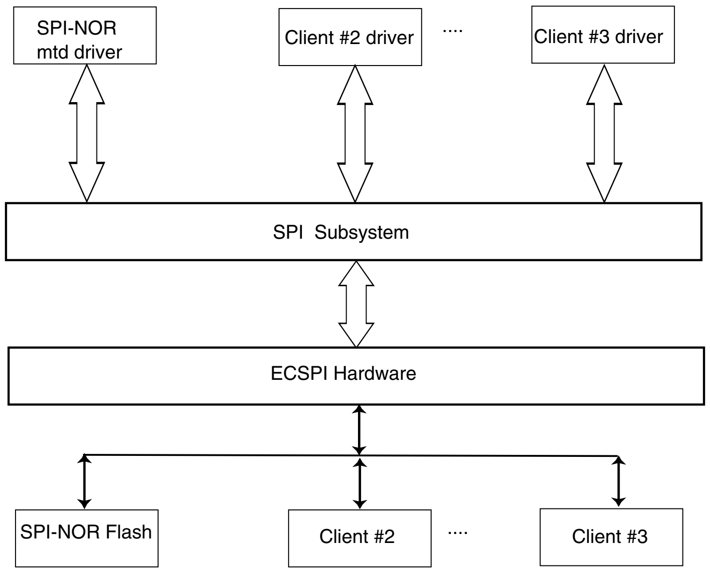


Figure 25-1. SPI Subsystem

All SPI clients must have a protocol driver associated with them and they all must be sharing the same controller driver. Only the controller driver can interact with the underlying SPI hardware module. Figure below shows how the different SPI drivers are layered in the SPI subsystem.

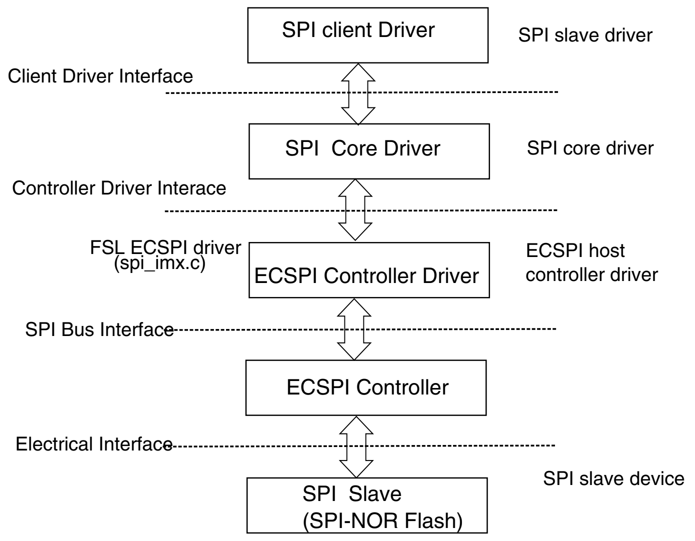


Figure 25-2. Layering of SPI Drivers in SPI Subsystem

25.2.2 Software Limitations

The ECSPI driver limitations are as follows:

- Does not currently have SPI slave logic implementation
- Does not support a single client connected to multiple masters
- Does not currently implement the user space interface with the help of the device node entry but supports sysfs interface

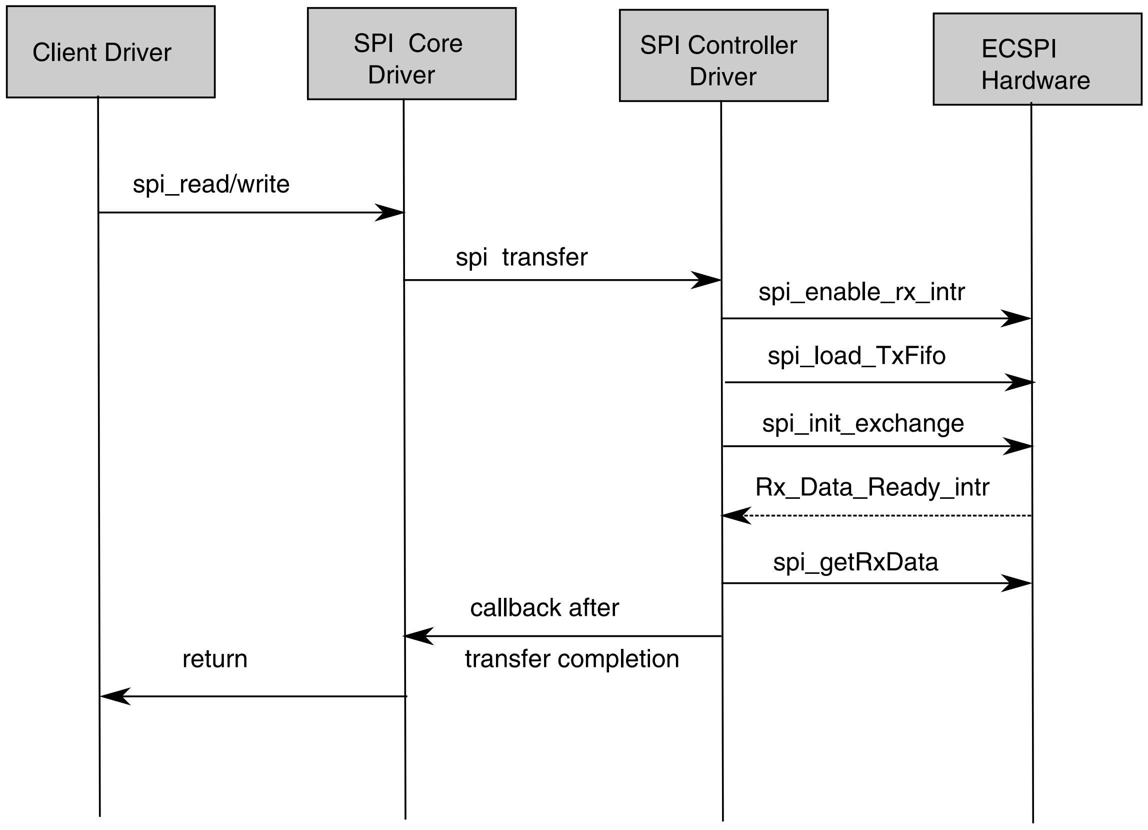
25.2.3 Standard Operations

The ECSPI driver is responsible for implementing standard entry points for init, exit, chip select, and transfer. The driver implements the following functions:

- Init function spi_imx_init() registers the device_driver structure.
- Probe function spi_imx_probe() performs initialization and registration of the SPI device specific structure with SPI core driver. The driver probes for memory and IRQ resources. Configures the IOMUX to enable ECSPI I/O pins, requests for IRQ and resets the hardware.
- Chip select function spi_imx_chipselect() configures the hardware ECSPI for the current SPI device. Sets the word size, transfer mode, data rate for this device.
- SPI transfer function spi_imx_transfer() handles data transfers operations.
- SPI setup function spi_imx_setup() initializes the current SPI device.
- SPI driver ISR spi_imx_isr() is called when the data transfer operation is completed and an interrupt is generated.

25.2.4 ECSPI Synchronous Operation

Figure below shows how the ECSPI provides synchronous read/write operations.


Client Driver
SPI Core Driver
SPI Controller Driver
ECSPI Hardware
spi_read/write
spi transfer
spi_enable_rx_intr
spi_load_TxFifo
spi_init_exchange
Rx_Data_Ready_intr
spi_getRxData
callback after transfer completion
return

25.3 Driver Features

The ECSPI module supports the following features:

- Implements each of the functions required by a ECSPI module to interface to Linux
- Multiple SPI master controllers
- Multi-client synchronous requests

25.3.1 Source Code Structure

Table below shows the source files available in the devices directory:

```
<ltib_dir>/rpm/BUILD/linux/drivers/spi/
```

Table 25-1. CSPI Driver Files

File	Description
spi_imx.c	SPI Master Controller driver

25.3.2 Menu Configuration Options

To get to the Linux kernel configuration options provided for this module, use the ./ltib -c command when located in the <ltib dir>.

On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

- CONFIG_SPI build support for the SPI core. In menuconfig, this option is available under:
  - Device Drivers > SPI Support.
- CONFIG_BITBANG is the Library code that is automatically selected by drivers that need it. SPI_IMX selects it. In menuconfig, this option is available under:
  - Device Drivers > SPI Support > Utilities for Bitbanging SPI masters.
- CONFIG_SPI_IMX implements the SPI master mode for ECSPI. In menuconfig, this option is available under:
  - Device Drivers > SPI Support > Freescale i.MX SPI controllers.

25.3.3 Programming Interface

This driver implements all the functions that are required by the SPI core to interface with the ECSPI hardware.

For more information, see the Linux document generated from build: make htmldocs.

25.3.4 Interrupt Requirements

The SPI interface generates interrupts.

ECSPI interrupt requirements are listed in table below.

Table 25-2. ECSPI Interrupt Requirements

Parameter	Equation	Typical	Worst Case
BaudRate/ Transfer Length	(BaudRate/(TransferLength)) * (1/Rxtl)	31250	1500000

The typical values are based on a baud rate of 1 Mbps with a receiver trigger level (Rxtl) of 1 and a 32-bit transfer length. The worst-case is based on a baud rate of 12 Mbps (max supported by the SPI interface) with a 8-bits transfer length.

Chapter 26
ARC USB Driver

26.1 Introduction

The universal serial bus (USB) driver implements a standard Linux driver interface to the ARC USB-HS OTG controller.

The USB provides a universal link that can be used across a wide range of PC-to-peripheral interconnects. It supports plug-and-play, port expansion, and any new USB peripheral that uses the same type of port.

The ARC USB controller is enhanced host controller interface (EHCI) compliant. This USB driver has the following features:

- High speed OTG core supported
- High Speed Host Only core(OTG2), high speed, full speed, and low devices are supported. An USB2Pci bridge is connected to OTG2 by default. Therefore, User may not be able to connect other USB devices on this port.
- High Speed Inter-Chip core(Host2)
- Host mode-Supports HID (Human Interface Devices), MSC (Mass Storage Class)
- Peripheral mode-Supports MSC, and CDC (Communication Devices Class) drivers which include ethernet and serial support
- Embedded DMA controller

26.1.1 Architectural Overview

The USB host system is composed of a number of hardware and software layers.

Figure below shows a conceptual block diagram of the building block layers in a host system that support USB 2.0.

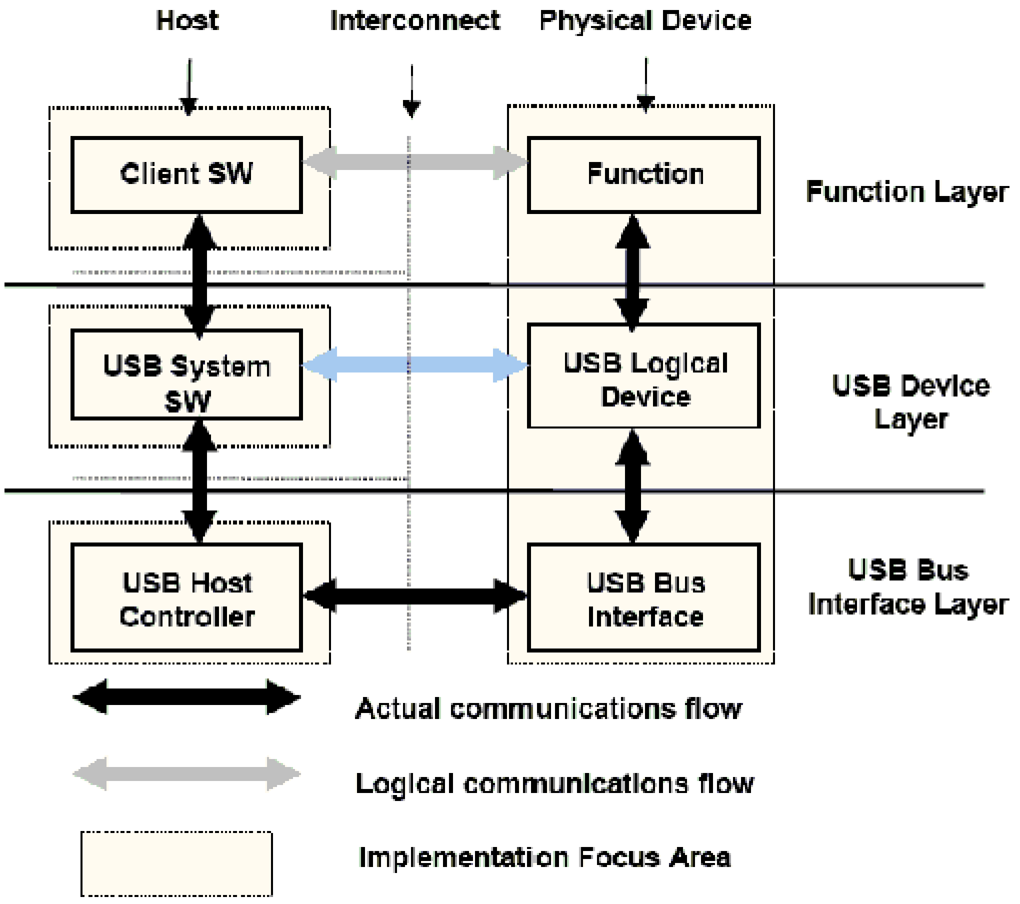


Figure 26-1. USB Block Diagram

26.2 Hardware Operation

For information on hardware operations, refer to the EHCI spec.ehci-r10.pdf.

The spec is available at http://www.usb.org/developers/docs/

26.2.1 Software Operation

The Linux OS contains a USB driver, which implements the USB protocols.

For the USB host, it only implements the hardware specified initialization functions. For the USB peripheral, it implements the gadget framework.

```
static struct usb_ep_ops fsl_ep_ops = {
          .enable = fsl_ep_enable,
          .disable = fsl_ep_disable,
          .alloc_request = fsl_alloc_request,
          .free_request = fsl_free_request,
          .queue = fsl_ep_queue,
          .dequeue = fsl_ep_dequeue,
          .set_halt = fsl_ep_set_halt,
          .fifo_status = arcotg_fifo_status,
          .fifo_flush = fsl_ep_fifo_flush,                                      /* flush 
fifo */
          };
static struct usb_gadget_ops fsl_gadget_ops = {
          .get_frame = fsl_get_frame,
          .wakeup = fsl_wakeup,
/*      .set_selfpowered = fsl_set_selfpowered, */                                  /* 
Always selfpowered */
          .vbus_session = fsl_vbus_session,
          .vbus_draw = fsl_vbus_draw,
          .pullup = fsl_pullup,
          };
```

- fsl_ep_enable-configures an endpoint making it usable
- fsl_ep_disable-specifies an endpoint is no longer usable
- fsl_alloc_request-allocates a request object to use with this endpoint
- fsl_free_request-frees a request object
- arcotg_ep_queue-queues (submits) an I/O request to an endpoint
- arcotg_ep_dequeue-dequeues (cancels, unlinks) an I/O request from an endpoint
- arcotg_ep_set_halt-sets the endpoint halt feature
- arcotg_fifo_status-get the total number of bytes to be moved with this transfer descriptor

For OTG, ID dynamic switch host/device modes are supported. Full OTG functions are temporarily not supported.

26.2.2 Source Code Structure

Table below shows the source files available in the source directory, <ltib_dir>/rpm/BUILD/linux/drivers/usb.

Table 26-1. USB Driver Files

File	Description
host/ehci-hcd.c	Host driver source file
host/ehci-arc.c	Host driver source file
host/ehci-mem-iram.c	Host driver source file for IRAM support
host/ehci-hub.c	Hub driver source file
host/ehci-mem.c	Memory management for host driver data structures
host/ehci-q.c	EHCI host queue manipulation
host/ehci-q-iram.c	Host driver source file for IRAM support
gadget/arcotg_udc.c	Peripheral driver source file
gadget/arcotg_udc.h	USB peripheral/endpoint management registers
otg/fsl_otg.c	OTG driver source file

Table continues on the next page...

Table 26-1. USB Driver Files (continued)

File	Description
otg/fsl_otg.h	OTG driver header file
otg/otg_fsm.c	OTG FSM implement source file
otg/otg_fsm.h	OTG FSM header file
gadget/fsl_updater.c	FSL manufacture tool USB char driver source file
gadget/fsl_updater.h	FSL manufacture tool USB char driver header file

Table below shows the platform related source files.

Table 26-2. USB Platform Source Files

File	Description
arch/arm/plat-mxc/include/mach/arc_otg.h	USB register define
include/linux/fsl_devices.h	FSL USB specific structures and enums

Table below shows the platform-related source files in the directory:

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/`

Table 26-3. USB Platform Header Files

File	Description
usb_dr.c	Platform-related initialization
usb_h1.c	Platform-related initialization
usb_h2.c	Platform-related initialization
usb_h3.c	Platform-related initialization

Table below shows the common platform source files in the directory:

`<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc.`

Table 26-4. USB Common Platform Files

File	Description
isp1504xc.c	ULPI PHY driver (USB3317 uses the same driver as ISP1504)
utmixc.c	Internal UTMI transceiver driver
usb_hsic_xcvr.c	HSIC featured phy's interface
usb_common.c	Common platform related part of USB driver
usb_wakeup.c	Handle USB wakeup events

26.2.3 Menu Configuration Options

To get to the Linux kernel configuration options available for this module, use the ./ltib -c command when located in the <ltib dir>.

On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

- CONFIG_USB-Build support for USB
- CONFIG_USB_EHCI_HCD-Build support for USB host driver. In menuconfig, this option is available under Device drivers > USB support > EHCI HCD (USB 2.0) support.

  By default, this option is Y.
- CONFIG_USB_EHCI_ARC-Build support for selecting the ARC EHCI host. In menuconfig, this option is available under Device drivers > USB support > Support for Freescale controller.

  By default, this option is Y.
- CONFIG_USB_EHCI_ARC_OTG-Build support for selecting the ARC EHCI OTG host. In menuconfig, this option is available under

  Device drivers > USB support > EHCI HCD (USB 2.0) support > Support for DR host port on Freescale controller.

  By default, this option is Y.
- CONFIG_USB_EHCI_ARC_HSIC Freescale HSIC USB Host Controller

  By default, this option is N.
- CONFIG_USB_EHCI_ROOT_HUB_TT-Some EHCI chips have vendor-specific extensions to integrate transaction translators, so that no OHCI or UHCI companion controller is needed. In menuconfig this option is available under

  Device drivers > USB support > Root Hub Transaction Translators.

  By default, this option is Y selected by USB_EHCI_ARC && USB_EHCI_HCD.
- CONFIG_USB_STORAGE-Build support for USB mass storage devices. In menuconfig this option is available under

  Device drivers > USB support > USB Mass Storage support.

  By default, this option is Y.
- CONFIG_USB_HID-Build support for all USB HID devices. In menuconfig this option is available under

Device drivers > HID Devices > USB Human Interface Device (full HID) support.

By default, this option is Y.

- CONFIG_USB_GADGET-Build support for USB gadget. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support.

  By default, this option is M.
- CONFIG_USB_GADGET_ARC-Build support for ARC USB gadget. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support > USB Peripheral Controller (Freescale USB Device Controller).

  By default, this option is Y.
- CONFIG_IMX_USB_CHARGER Freescale i.MX 6 USB Charger Detection

  By default, this option is Y.
- CONFIG_USB_OTG-OTG Support, support dual role with ID pin detection.

  By default, this option is Y.
- CONFIG_MXC_OTG-USB OTG pin detect support for Freescale USB OTG Controller

  By default, this option is Y.
- CONFIG_USB_ETH-Build support for Ethernet gadget. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support > Ethernet Gadget (with CDC Ethernet Support).

  By default, this option is M.
- CONFIG_USB_ETH_RNDIS-Build support for Ethernet RNDIS protocol. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support > Ethernet Gadget (with CDC Ethernet Support) > RNDIS support.

  By default, this option is Y.
- CONFIG_USB_FILE_STORAGE-Build support for Mass Storage gadget. In menuconfig, this option is available under

Device drivers > USB support > USB Gadget Support > File-backed Storage Gadget.

By default, this option is M.

- CONFIG_USB_G_SERIAL-Build support for ACM gadget. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support > Serial Gadget (with CDC ACM support).

  By default, this option is M.

26.2.4 Programming Interface

This driver implements all the functions that are required by the USB bus protocol to interface with the i.MX USB ports.

See the *BSP API* document, for more information.

26.3 System WakeUp

Both host and device connect/disconnect event can be system wakeup source, as well the device remote wakeup.

But all the wakeup functions depend on the USB PHY power supply, including 1p1, 2p5, 3p3, no power supply, all the wakeup function behavior will be unpredictable.

For host remote wake feature, there is a limitation that our system clock needs a short time to be stable after resume, if the resume signal sent by the connected device only last very short time (less than the time need to make clock stable), the remote wakeup may fail. At such case, we should not turn off some clocks to decrease the time needs to be stable to fix such issue.

26.3.1 USB Wakeup usage

USB wakeup usage is outlined in three procedures: how to enable USB wakeup system, what kinds of wakeup events USB supports, and how to close USB child device power.

26.3.2 How to Enable USB WakeUp System Ability

For otg port:

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup
```

For device-only port:

```
echo enabled > /sys/devices/platform/fsl-usb2-udc/power/wakeup
```

For host-only port:

```
echo enabled > /sys/devices/platform/fsl-ehci.x/power/wakeup
(x is the port num)
```

For USB child device:

```
echo enabled > /sys/bus/usb/devices/1-1/power/wakeup
```

26.3.3 WakeUp Events Supported by USB

USBOTG port is used as an example.

Device mode wakeup:

connect wakeup: when USB line connects to usb port, the other port is connected to PC (Wakeup signal: vbus change)

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup
```

Host mode wakeup:

connect wakeup: when USB device connects to host port (Wakeup signal: ID/(dm/dp) change)

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup
```

disconnect wakeup: when USB device disconnects to host port (Wakeup signal: ID/(dm/dp) change)

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup
```

remote wakeup: press USB device (i.e. press USB key on the USB keyboard) when USB device connects to host port (Wakeup signal: ID/(dm/dp) change):

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup
echo enabled > /sys/bus/usb/devices/1-1/power/wakeup
```

NOTE

For the hub on board, it is necessary to enable hub's wakeup first. For remote wakeup, it is necessary to perform the three steps outlined below:

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup (enable the roothub's wakeup)
echo enabled > /sys/bus/usb/devices/1-1/power/wakeup (enable the second level hub's wakeup)
(1-1 is the hub name)
echo enabled > /sys/bus/usb/devices/1-1.1/power/wakeup (enable the USB device wakeup, that
device connects at second level hub)
(1-1.1 is the USB device name)
```

26.3.4 How to Close the USB Child Device Power

The following code string outlines how to close the USB child device power:

```
echo auto > /sys/bus/usb/devices/1-1/power/control
echo auto > /sys/bus/usb/devices/1-1.1/power/control (If there is a hub at usb device)
```

Chapter 27
Fast Ethernet Controller (FEC) Driver

27.1 Introduction

The Fast Ethernet Controller (FEC) driver performs the full set of IEEE 802.3/Ethernet CSMA/CD media access control and channel interface functions.

The FEC requires an external interface adapter and transceiver function to complete the interface to the Ethernet media. It supports half or full-duplex operation on 10 Mbps, 100 Mbpsrelated Ethernet networks.

The FEC driver supports the following features:

- Full/Half duplex operation
- Link status change detect
- Auto-negotiation (determines the network speed and full or half-duplex operation)
- Transmits features such as automatic retransmission on collision and CRC generation
- Obtaining statistics from the device such as transmit collisions

The network adapter can be accessed through the ifconfig command with interface name ethx. The driver auto-probes the external adaptor (PHY device).

27.2 Hardware Operation

The FEC is an Ethernet controller that interfaces the system to the LAN network.

The FEC supports different standard MAC-PHY (physical) interfaces for connection to an external Ethernet transceiver. The FEC supports the 10/100 Mbps MII, and 10/100 Mbps RMII.

A brief overview of the device functionality is provided here. For details see the FEC chapter of the *i.MX 6 i.MX 6SoloLite Multimedia Applications Processor Reference Manual*.

In MII mode, there are 18 signals defined by the IEEE 802.3 standard and supported by the EMAC. MII, RMII modes uses a subset of the 18 signals. These signals are listed in table below.

Table 27-1. Pin Usage in MII, RMII

Direction	EMAC Pin Name	MII Usage	RMII Usage	RGMII Usage (i.MX 6SL cannot support)
In/Out	FEC_MDIO	Management Data Input/Output	Management Data Input/output	Management Data Input/Output
Out	FEC_MDC	Management Data Clock	General output	Management Data Clock
Out	FEC_TXD[0]	Data out, bit 0	Data out, bit 0	Data out, bit 0
Out	FEC_TXD[1]	Data out, bit 1	Data out, bit 1	Data out, bit 1
Out	FEC_TXD[2]	Data out, bit 2	Not Used	Data out, bit 2
Out	FEC_TXD[3]	Data out, bit 3	Not Used	Data out, bit 3
Out	FEC_TX_EN	Transmit Enable	Transmit Enable	Transmit Enable
Out	FEC_TX_ER	Transmit Error	Not Used	Not Used
In	FEC_CRS	Carrier Sense	Not Used	Not Used
In	FEC_COL	Collision	Not Used	Not Used
In	FEC_TX_CLK	Transmit Clock	Not Used	Synchronous clock reference (REF_CLK, can connect from PHY)
In	FEC_RX_ER	Receive Error	Receive Error	Not Used
In	FEC_RX_CLK	Receive Clock	Not Used	Synchronous clock reference (REF_CLK, can connect from PHY)
In	FEC_RX_DV	Receive Data Valid	Receive Data Valid and generate CRS	RXDV XOR RXERR on the falling edge of FEC_RX_CLK.
In	FEC_RXD[0]	Data in, bit 0	Data in, bit 0	Data in, bit 0
In	FEC_RXD[1]	Data in, bit 1	Data in, bit 1	Data in, bit 1
In	FEC_RXD[2]	Data in, bit 2	Not Used	Data in, bit 2
In	FEC_RXD[3]	Data in, bit 3	Not Used	Data in, bit 3

The MII management interface consists of two pins, FEC_MDIO, and FEC_MDC. The FEC hardware operation can be divided in the parts listed below. For detailed information consult the *i.MX 6SoloLite Multimedia Applications Processor Reference Manual*.

- Transmission-The Ethernet transmitter is designed to work with almost no intervention from software. Once ECR[ETHER_EN] is asserted and data appears in the transmit FIFO, the Ethernet MAC is able to transmit onto the network. When the transmit FIFO fills to the watermark (defined by the TFWR), the MAC transmit logic asserts FEC_TX_EN and starts transmitting the preamble (PA) sequence, the start frame delimiter (SFD), and then the frame information from the FIFO. However, the controller defers the transmission if the network is busy (FEC_CRS asserts).

- Before transmitting, the controller waits for carrier sense to become inactive, then determines if carrier sense stays inactive for 60 bit times. If the transmission begins after waiting an additional 36 bit times (96 bit times after carrier sense originally became inactive), both buffer (TXB) and frame (TXF) interrupts may be generated as determined by the settings in the EIMR.
- Reception-The FEC receiver is designed to work with almost no intervention from the host and can perform address recognition, CRC checking, short frame checking, and maximum frame length checking. When the driver enables the FEC receiver by asserting ECR[ETHER_EN], it immediately starts processing receive frames. When FEC_RX_DV asserts, the receiver checks for a valid PA/SFD header. If the PA/SFD is valid, it is stripped and the frame is processed by the receiver. If a valid PA/SFD is not found, the frame is ignored. In MII mode, the receiver checks for at least one byte matching the SFD. Zero or more PA bytes may occur, but if a 00 bit sequence is detected prior to the SFD byte, the frame is ignored.
- After the first six bytes of the frame have been received, the FEC performs address recognition on the frame. During reception, the Ethernet controller checks for various error conditions and once the entire frame is written into the FIFO, a 32-bit frame status word is written into the FIFO. This status word contains the M, BC, MC, LG, NO, CR, OV, and TR status bits, and the frame length. Receive Buffer (RXB) and Frame Interrupts (RXF) may be generated if enabled by the EIMR register. When the receive frame is complete, the FEC sets the L bit in the RxBD, writes the other frame status bits into the RxBD, and clears the E bit. The Ethernet controller next generates a maskable interrupt (RXF bit in EIR, maskable by RXF bit in EIMR), indicating that a frame has been received and is in memory. The Ethernet controller then waits for a new frame.
- Interrupt management-When an event occurs that sets a bit in the EIR, an interrupt is generated if the corresponding bit in the interrupt mask register (EIMR) is also set. The bit in the EIR is cleared if a one is written to that bit position; writing zero has no effect. This register is cleared upon hardware reset. These interrupts can be divided into operational interrupts, transceiver/network error interrupts, and internal error interrupts. Interrupts which may occur in normal operation are GRA, TXF, TXB, RXF, RXB. Interrupts resulting from errors/problems detected in the network or transceiver are HBERR, BABR, BABT, LC, and RL. Interrupts resulting from internal errors are HBERR and UN. Some of the error interrupts are independently counted in the MIB block counters. Software may choose to mask off these interrupts as these errors are visible to network management through the MIB counters.
- PHY management-phylib was used to manage all the FEC phy related operation such as phy discovery, link status, and state machine.MDIO bus will be created in FEC driver and registered into the system.You can refer to Documentation/networking/ phy.txt under linux source directory for more information.

27.2.1 Software Operation

The FEC driver supports the following functions:

- Module initialization-Initializes the module with the device specific structure
- Rx/Tx transmition
- Interrupt servicing routine
- PHY management
- FEC management such init/start/stop
- i.MX 6 FEC module use little-endian format

27.2.2 Source Code Structure

Table below shows the source files.

They are available in the

`<ltib_dir>/rpm/BUILD/linux/drivers/net directory.`

Table 27-2. FEC Driver Files

File	Description
fec.h	Header file defining registers
fec.c	Linux driver for Ethernet LAN controller

For more information about the generic Linux driver, see the <ltib_dir>/rpm/BUILD/linux/drivers/net/fec.c source file.

27.2.3 Menu Configuration Options

To get to the Linux kernel configuration option provided for this module, use the ./ltib -c command when located in the <ltib dir>.

On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following option to enable this module:

- CONFIG_FEC is provided for this module. This option is available under:
  - Device Drivers > Network device support > Ethernet (10, 100 or 1000 Mbit) > FEC Ethernet controller.
  - To mount NFS-rootfs through FEC, disable the other Network config in the menuconfig if need.

27.3 Programming Interface

Table 27-2 lists the source files for the FEC driver.

The following section shows the modifications that were required to the original Ethernet driver source for porting it to the i.MX device.

27.3.1 Device-Specific Defines

Device-specific defines are added to the header file (fec.h) and they provide common board configuration options.

fec.h defines the struct for the register access and the struct for the buffer descriptor. For example,

```
/*
 *      Define the buffer descriptor structure.
 */
struct bufdesc {
          unsigned short            cbd_datlen;       /* Data length */
          unsigned short            cbd_sc;           /* Control and status info */
          unsigned long             cbd_bufaddr;      /* Buffer address */
#ifdef CONFIG_ENHANCED_BD
          unsigned long cbd_esc;
          unsigned long cbd_prot;
          unsigned long cbd_bdu;
          unsigned long ts;
          unsigned short res0[4];
#endif
};
/*
 *      Define the register access structure.
 */
#define FEC_IEVENT                0x004 /* Interrupt event reg */
#define FEC_IMASK                 0x008 /* Interrupt mask reg */
#define FEC_R_DES_ACTIVE          0x010 /* Receive descriptor reg */
#define FEC_X_DES_ACTIVE          0x014 /* Transmit descriptor reg */
#define FEC_ECNTRL                0x024 /* Ethernet control reg */
#define FEC_MII_DATA              0x040 /* MII manage frame reg */
#define FEC_MII_SPEED             0x044 /* MII speed control reg */
#define FEC_MIB_CTRLSTAT          0x064 /* MIB control/status reg */
#define FEC_R_CNTRL               0x084 /* Receive control reg */
#define FEC_X_CNTRL               0x0c4 /* Transmit Control reg */
#define FEC_ADDR_LOW              0x0e4 /* Low 32bits MAC address */
#define FEC_ADDR_HIGH             0x0e8 /* High 16bits MAC address */
#define FEC_OPD                   0x0ec /* Opcode + Pause duration */
#define FEC_HASH_TABLE_HIGH       0x118 /* High 32bits hash table */
#define FEC_HASH_TABLE_LOW        0x11c /* Low 32bits hash table */
#define FEC_GRP_HASH_TABLE_HIGH   0x120 /* High 32bits hash table */
#define FEC_GRP_HASH_TABLE_LOW    0x124 /* Low 32bits hash table */
#define FEC_X_WMRK                0x144 /* FIFO transmit water mark */
#define FEC_R_BOUND               0x14c /* FIFO receive bound reg */
#define FEC_R_FSTART              0x150 /* FIFO receive start reg */
#define FEC_R_DES_START           0x180 /* Receive descriptor ring */
#define FEC_X_DES_START           0x184 /* Transmit descriptor ring */
#define FEC_R_BUFF_SIZE           0x188 /* Maximum receive buff size */
```

```
#define FEC_MIIGSK_CFGR         0x300 /* MIIGSK config register */
#define FEC_MIIGSK_ENR          0x308 /* MIIGSK enable register */
```

27.3.2 Getting a MAC Address

The following statement gets the MAC address through the OCOTP (IC Identification) by default for i.MX 6.

The MAC address can be set through bootloader such as u-boot. FEC driver will use it to confiure the MAC address for network devices. i.MX 6 user needs to provide MAC address by kernel command line so that user can use sb_loader to load kernel and run it without bootloader interaction.

Chapter 28
Universal Asynchronous Receiver/Transmitter (UART) Driver

28.1 Introduction

The low-level UART driver interfaces the Linux serial driver API to all the UART ports.

It has the following features:

- Interrupt-driven and SDMA-driven transmit/receive of characters
- Standard Linux baud rates up to 4 Mbps
- Transmit and receive characters with 7-bit and 8-bit character lengths
- Transmits one or two stop bits
- Supports TIOCMGET IOCTL to read the modem control lines. Only supports the constants TIOCM_CTS and TIOCM_CAR, plus TIOCM_RI in DTE mode only
- Supports TIOCMSET IOCTL to set the modem control lines. Supports the constants TIOCM_RTS and TIOCM_DTR only
- Odd and even parity
- XON/XOFF software flow control. Serial communication using software flow control is reliable when communication speeds are not too high and the probability of buffer overruns is minimal
- CTS/RTS hardware flow control-both interrupt-driven software-controlled hardware flow and hardware-driven hardware-controlled flow
- Send and receive break characters through the standard Linux serial API
- Recognizes frame and parity errors
- Ability to ignore characters with break, parity and frame errors
- Get and set UART port information through the TIOCGSSERIAL and TIOCSSERIAL TTY IOCTL. Some programs like setserial and dip use this feature to make sure that the baud rate was set properly and to get general information on the device. The UART type should be set to 52 as defined in the serial_core.h header file.
- Serial IrDA

- Power management feature by suspending and resuming the URT ports
- Standard TTY layer IOCTL calls

All the UART ports can be accessed from the device files /dev/ttymxc0 to /dev/ttymxc1. Autobaud detection is not supported.

NOTE

If you want to use the DMA support for UART please also enable the RTS/CTS for it. The DMA may be abnormal if you do not enable the RTS/CTS.

28.2 Hardware Operation

Refer to the *i.MX 6SoloLite Applications Processor Reference Manual* to determine the number of UART modules available in the device.

Each UART hardware port is capable of standard RS-232 serial communication and has support for IrDA 1.0.

Each UART contains a 32-byte transmitter FIFO and a 32-half-word deep receiver FIFO. Each UART also supports a variety of maskable interrupts when the data level in each FIFO reaches a programmed threshold level and when there is a change in state in the modem signals. Each UART can be programmed to be in DCE or DTE mode.

28.2.1 Software Operation

The Linux OS contains a core UART driver that manages many of the serial operations that are common across UART drivers for various platforms.

The low-level UART driver is responsible for supplying information such as the UART port information and a set of control functions to the core UART driver. These functions are implemented as a low-level interface between the Linux OS and the UART hardware. They cannot be called from other drivers or from a user application. The control functions used to control the hardware are passed to the core driver through a structure called uart_ops, and the port information is passed through a structure called uart_port. The low level driver is also responsible for handling the various interrupts for the UART ports, and providing console support if necessary.

Each UART can be configured to use DMA for the data transfer. These configuration options are provided in the mxc_uart.h header file. The user can specify the size of the DMA receive buffer. The minimum size of this buffer is 512 bytes. The size should be a multiple of 256. The driver breaks the DMA receive buffer into smaller sub-buffers of

256 bytes and registers these buffers with the DMA system. DMA transmit buffer size is fixed at 1024 bytes. The size is limited by the size of the Linux UART transmit buffer (1024).

The driver requests two DMA channels for the UARTs that need DMA transfer. On a receive transaction, the driver copies the data from the DMA receive buffer to the TTY Flip Buffer.

While using DMA to transmit, the driver copies the data from the UART transmit buffer to the DMA transmit buffer and sends this buffer to the DMA system. The user should use hardware-driven hardware flow control when using DMA data transfer. For more information, see the Linux documentation on the serial driver in the kernel source tree.

The low-level driver supports both interrupt-driven software-controlled hardware flow control and hardware-driven hardware flow control. The hardware flow control method can be configured using the options provided in the header file. The user has the capability to de-assert the CTS line using the available IOCTL calls. If the user wishes to assert the CTS line, then control is transferred back to the receiver, as long as the driver has been configured to use hardware-driven hardware flow control.

28.2.2 Driver Features

The UART driver supports the following features:

- Baud rates up to 4 Mbps
- Recognizes frame and parity errors only in interrupt-driven mode; does not recognize these errors in DMA-driven mode
- Sends, receives, and appropriately handles break characters
- Recognizes the modem control signals
- Ignores characters with frame, parity, and break errors if requested to do so
- Implements support for software and hardware flow control (software-controlled and hardware-controlled)
- Get and set the UART port information; certain flow control count information is not available in hardware-driven hardware flow control mode
- Implements support for Serial IrDA
- Power management
- Interrupt-driven and DMA-driven data transfer

28.2.3 Source Code Structure

Table below shows the UART driver source files that are available in the directory:

`<ltib_dir>/rpm/BUILD/linux/drivers/tty/serial.`

Table 28-1. UART Driver Files

File	Description
imx.c	Low level driver

Table below shows the header files associated with the UART driver.

Table 28-2. UART Global Header Files

File	Description
<ltib_dir>/rpm/BUILD/linux/ arch/arm/plat-mxc/ include/mach/imx-uart.h	UART header that contains UART configuration data structure definitions

28.3 Configuration

This section discusses configuration options associated with Linux, chip configuration options, and board configuration options.

28.3.1 Menu Configuration Options

To get to the Linux kernel configuration options provided for this module, use the ./ltib -c command when located in the <ltib dir>. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

- CONFIG_SERIAL_IMX -Used for the UART driver for the UART ports. In menuconfig, this option is available under

  Device Drivers > Character devices > Serial drivers > IMX serial port support.

  By default, this option is Y.
- CONFIG_SERIAL_IMX_CONSOLE-Chooses the Internal UART to bring up the system console. This option is dependent on the CONFIG_SERIAL_IMX option. In the menuconfig this option is available under

  Device Drivers > Character devices > Serial drivers > IMX serial port support > Console on IMX serial port

  By default, this option is Y.

28.3.2 Source Code Configuration Options

This section details the chip configuration options and board configuration options.

28.3.3 Chip Configuration Options

28.3.4 Board Configuration Options

For i.MX 6SoloLite, the board specific configuration options for the driver is set within:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/board-mx6sl_evk.c
```

28.4 Programming Interface

The UART driver implements all the methods required by the Linux serial API to interface with the UART port.

The driver implements and provides a set of control methods to the Linux core UART driver. For more information about the methods implemented in the driver, see the API document.

28.4.1 Interrupt Requirements

The UART driver interface generates only one interrupt.

The status is used to determine which kinds of interrupt occurs, such as RX or TX.

With the SDMA enabled, the DMA RX interrupt occurs only when the received data fills all the 4K buffer. The DMA TX interrupt occurs when the data is sent out.

Chapter 29
Pulse-Width Modulator (PWM) Driver

29.1 Introduction

The pulse-width modulator (PWM) has a 16-bit counter and is optimized to generate sound from stored sample audio images and generate tones.

The PWM has 16-bit resolution and uses a 4x16 data FIFO to generate sound. The software module is composed of a Linux driver that allows privileged users to control the backlight by the appropriate duty cycle of the PWM Output (PWMO) signal.

29.1.1 Hardware Operation

Figure below shows the PWM block diagram.

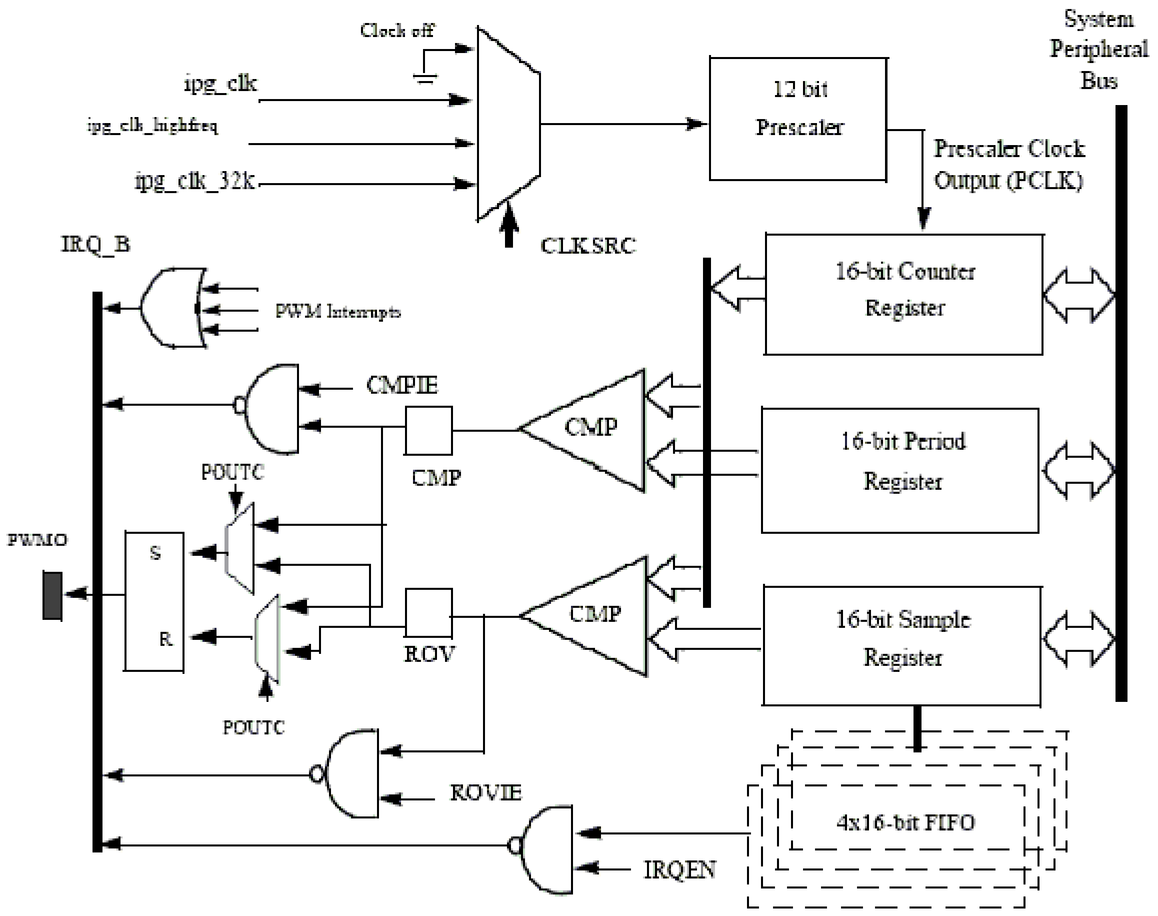


Figure 29-1. PWM Block Diagram

The PWM follows IP Bus protocol for interfacing with the processor core. It does not interface with any other modules inside the device except for the clock and reset inputs from the Clock Control Module (CCM) and interrupt signals to the processor interrupt handler. The PWM includes a single external output signal, PMWO. The PWM includes the following internal signals:

- Three clock inputs
- Four interrupt lines
- One hardware reset line
- Four low power and debug mode signals
- Four scan signals
- Standard IP slave bus signals

29.1.2 Clocks

The clock that feeds the prescaler can be selected from:

- High frequency clock-provided by the CCM. The PWM can be run from this clock in low power mode.
- Low reference clock-32 KHz low reference clock provided by the CCM. The PWM can be run from this clock in the low power mode.
- Global functional clock-for normal operations. In low power modes this clock can be switched off.

The clock input source is determined by the CLKSRC field of the PWM control register. The CLKSRC value should only be changed when the PWM is disabled.

29.1.3 Software Operation

The PWM device driver reduces the amount of power sent to a load by varying the width of a series of pulses to the power source. One common and effective use of the PWM is controlling the backlight of a QVGA panel with a variable duty cycle.

Table below provides a summary of the interface functions in source code.

Table 29-1. PWM Driver Summary

Function	Description
struct pwm_device *pwm_request(int pwm_id, const char *label)	Request a PWM device
void pwm_free(struct pwm_device *pwm)	Free a PWM device
int pwm_config(struct pwm_device *pwm, int duty_ns, int period_ns)	Change a PWM device configuration
int pwm_enable(struct pwm_device *pwm)	Start a PWM output toggling
int pwm_disable(struct pwm_device *pwm)	Stop a PWM output toggling

The function pwm_config() includes most of the configuration tasks for the PWM module, including the clock source option, and period and duty cycle of the PWM output signal. It is recommended to select the peripheral clock of the PWM module, rather than the local functional clock, as the local functional clock can change.

29.1.4 Driver Features

The PWM driver includes the following software and hardware support:

- Duty cycle modulation
- Varying output intervals
- Two power management modes-full on and full of

29.1.5 Source Code Structure

Table below lists the source files and headers available in the following directories:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/pwm.c
<ltib_dir>/rpm/BUILD/linux/include/linux/pwm.h
```

Table 29-2. PWM Driver Files

File	Description
pwm.h	Functions declaration
pwm.c	Functions definition

29.1.6 Menu Configuration Options

To get to the PWM driver, use the command ./ltib -c when located in the <ltib dir>. On the screen displayed, select **Configure the kernel** and exit. When the next screen appears select the following option to enable the PWM driver:

- System Type > Enable PWM driver
- Select the following option to enable the Backlight driver:

  Device Drivers > Graphics support > Backlight & LCD device support > Generic PWM based Backlight Driver

Chapter 30
Watchdog (WDOG) Driver

30.1 Introduction

The Watchdog Timer module protects against system failures by providing an escape from unexpected hang or infinite loop situations or programming errors.

Some platforms may have two WDOG modules with one of them having interrupt capability.

30.1.1 Hardware Operation

Once the WDOG timer is activated, it must be serviced by software on a periodic basis.

If servicing does not take place in time, the WDOG times out. Upon a time-out, the WDOG either asserts the wdog_b signal or a wdog_rst_b system reset signal, depending on software configuration. The watchdog module cannot be deactivated once it is activated.

30.1.2 Software Operation

The Linux OS has a standard WDOG interface that allows support of a WDOG driver for a specific platform.

WDOG can be suspended/resumed in STOP/DOZE and WAIT modes independently. Since some bits of the WGOD registers are only one-time programmable after booting, ensure these registers are written correctly.

30.2 Generic WDOG Driver

The generic WGOD driver is implemented in the <ltib_dir>/rpm/BUILD/linux/drivers/watchdog/imx2_wdt.c file.

It provides functions for various IOCTLs and read/write calls from the user level program to control the WDOG.

30.2.1 Driver Features

This WDOG implementation includes the following features:

- Generates the reset signal if it is enabled but not serviced within a predefined timeout value (defined in milliseconds in one of the WDOG source files)
- Does not generate the reset signal if it is serviced within a predefined timeout value
- Provides IOCTL/read/write required by the standard WDOG subsystem

30.2.2 Menu Configuration Options

To get to the Linux kernel configuration option provided for this module, use the ./ltib -c command when located in the <ltib dir>. On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following option to enable this module:

- CONFIG_IMX2_WDT-Enables Watchdog timer module. This option is available under Device Drivers > Watchdog Timer Support > IMX2+ Watchdog.

30.2.3 Source Code Structure

Table below shows the source files for WDOG drivers that are in the following directory:

`<ltib_dir>/rpm/BUILD/linux/drivers/watchdog.`

Table 30-1. WDOG Driver Files

File	Description
imx2_wdt.c	WDOG function implementations

Watchdog system reset function is located under <ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/system.c

30.2.4 Programming Interface

The following IOCTLs are supported in the WDOG driver:

- WDIOC_GETSUPPORT
- WDIOC_GETSTATUS
- WDIOC_GETBOOTSTATUS
- WDIOC_KEEPALIVE
- WDIOC_SETTIMEOUT
- WDIOC_GETTIMEOUT

For detailed descriptions about these IOCTLs, see <ltib_dir>/rpm/BUILD/linux/Documentation/watchdog.

Chapter 31
OProfile

31.1 Introduction

OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.

OProfile is released under the GNU GPL. It consists of a kernel driver, a daemon for collecting sample data, and several post-profiling tools for turning data into information.

31.1.1 Overview

OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling.

All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

31.1.2 Features

OProfile has the following features.

- Unobtrusive-No special recompilations or wrapper libraries are necessary. Even debug symbols (-g option to gcc) are not necessary unless users want to produce annotated source. No kernel patch is needed; just insert the module.
- System-wide profiling-All code running on the system is profiled, enabling analysis of system performance.
- Performance counter support-Enables collection of various low-level data and association for particular sections of code.
- Call-graph support-With an 2.6 kernel, OProfile can provide gprof-style call-graph profiling data.

- Low overhead-OProfile has a typical overhead of 1-8% depending on the sampling frequency and workload.
- Post-profile analysis-Profile data can be produced on the function-level or instruction-level detail. Source trees, annotated with profile information, can be created. A hit list of applications and functions that utilize the most CPU time across the whole system can be produced.
- System support-Works with almost any 2.2, 2.4 and 2.6 kernels, and works on based platforms.

31.1.3 Hardware Operation

OProfile is a statistical continuous profiler.

In other words, profiles are generated by regularly sampling the current registers on each CPU (from an interrupt handler, the saved PC value at the time of interrupt is stored), and converting that runtime PC value into something meaningful to the programmer.

OProfile achieves this by taking the stream of sampled PC values, along with the detail of which task was running at the time of the interrupt, and converting the values into a file offset against a particular binary file. Each PC value is thus converted into a tuple (group or set) of binary-image offset. The userspace tools can use this data to reconstruct where the code came from, including the particular assembly instructions, symbol, and source line (through the binary debug information if present).

Regularly sampling the PC value like this approximates what actually was executed and how often and, more often than not, this statistical approximation is good enough to reflect reality. In common operation, the time between each sample interrupt is regulated by a fixed number of clock cycles. This implies that the results reflect where the CPU is spending the most time. This is a very useful information source for performance analysis.

The ARM CPU provides hardware performance counters capable of measuring these events at the hardware level. Typically, these counters increment once per each event and generate an interrupt on reaching some pre-defined number of events. OProfile can use these interrupts to generate samples and the profile results are a statistical approximation of which code caused how many instances of the given event.

31.2 Software Operation

31.2.1 Architecture Specific Components

OProfile supports the hardware performance counters available on a particular architecture. Code for managing the details of setting up and managing these counters can be located in the kernel source tree in the relevant <ltib_dir>/rpm/BUILD/linux/arch/arm/oprofile directory. The architecture-specific implementation operates through filling in the oprofile_operations structure at initialization. This provides a set of operations, such as setup(), start(), stop(), and so on, that manage the hardware-specific details the performance counter registers.

The other important facility available to the architecture code is oprofile_add_sample(). This is where a particular sample taken at interrupt time is fed into the generic OProfile driver code.

31.2.2 oprofilefs Pseudo Filesystem

OProfile implements a pseudo-filesystem known as oprofilefs, which is mounted from userspace at /dev/oprofile. This consists of small files for reporting and receiving configuration from userspace, as well as the actual character device that the OProfile userspace receives samples from. At setup() time, the architecture-specific code may add further configuration files related to the details of the performance counters. The filesystem also contains a stats directory with a number of useful counters for various OProfile events.

31.2.3 Generic Kernel Driver

The generic kernel driver resides in <ltib_dir>/rpm/BUILD/linux/drivers/oprofile/, and forms the core of how OProfile operates in the kernel. The generic kernel driver takes samples delivered from the architecture-specific code (through oprofile_add_sample()), and buffers this data (in a transformed configuration) until releasing the data to the userspace daemon through the /dev/oprofile/buffer character device.

31.2.4 OProfile Daemon

The OProfile userspace daemon takes the raw data provided by the kernel and writes it to the disk. It takes the single data stream from the kernel and logs sample data against a number of sample files (available in /var/lib/oprofile/samples/current/). For the benefit of the separate functionality, the names and paths of these sample files are changed to reflect where the samples were from. This can include thread IDs, the binary file path, the event type used, and more.

After this final step from interrupt to disk file, the data is now persistent (that is, changes in the running of the system do not invalidate stored data). This enables the post-profiling tools to run on this data at any time (assuming the original binary files are still available and unchanged).

31.2.5 Post Profiling Tools

The collected data must be presented to the user in a useful form. This is the job of the post-profiling tools. In general, they collate a subset of the available sample files, load and process each one correlated against the relevant binary file, and produce user readable information.

31.3 Requirements

OProfile has the following requirements.

- Add Oprofile support with Cortex-A8 Event Monitor

31.3.1 Source Code Structure

Oprofile platform-specific source files are available in the directory:

`<ltib_dir>/rpm/BUILD/linux/arch/arm/oprofile/`

Table 31-1. OProfile Source Files

File	Description
op_arm_model.h	Header File with the register and bit definitions
common.c	Source file with the implementation required for all platforms

The generic kernel driver for Oprofile is located under <ltib_dir>/rpm/BUILD/linux/drivers/oprofile/

31.3.2 Menu Configuration Options

The following Linux kernel configurations are provided for this module.

To get to the Oprofile configuration, use the command ./ltib -c from the <ltib dir>. On the screen, first go to Package list and select oprofile. Then return to the first screen and, select **Configure Kernel**, then exit, and a new screen appears.

- CONFIG_OPROFILE-configuration option for the oprofile driver. In the menuconfig this option is available under
- General Setup > Profiling support (EXPERIMENTAL) > OProfile system profiling (EXPERIMENTAL)

31.3.3 Programming Interface

This driver implements all the methods required to configure and control PMU and L2 cache EVTMON counters.

More information, see the Linux document generated from build: make htmldocs.

31.3.4 Interrupt Requirements

The number of interrupts generated with respect to the OProfile driver are numerous. The latency requirements are not needed.

The rate at which interrupts are generated depends on the event.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: IMXL6SLRM
Rev. L3.0.35_4.1.0
09/2013