i.MX50 RD3 Linux

Reference Manual

Part Number: 926-77210
Rev. 11.04.01
04/2011

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

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

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

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

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

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

Contents

About This Book

Chapter 1
Machine Specific Layer (MSL)

Chapter 2
Smart Direct Memory Access (SDMA) API

Chapter 3
Direct Memory Access Controller (DMAC) API

Chapter 4
MAX17135 Temperature Sensor Driver

Chapter 5
MAX17135 Regulator Driver

Chapter 6
MC34708 Regulator Driver

Chapter 7
MC34708 Digitizer Driver

Chapter 8
MC34708 RTC Driver

Chapter 9
CPU Frequency Scaling (CPUFREQ) Driver

Chapter 10
Dynamic Voltage Frequency Scaling (DVFS) Driver

Chapter 11
Software Based Peripheral Domain Frequency Scaling

Chapter 12
Low-level Power Management (PM) Driver

Chapter 13
ELCDIF Frame Buffer Driver

Chapter 14
Electrophoretic Display Controller (EPDC) Frame Buffer Driver

Chapter 15
Data Co-Processor (DCP) Driver

Chapter 16
Random Number Generator (RNG) Driver

Chapter 17
Video for Linux Two (V4L2) Driver

Chapter 18
Pixel Pipeline (PxP) DMA-ENGINE Driver

Chapter 19
Graphics Processing Unit (GPU)

Chapter 20
X Windows Acceleration

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

Chapter 22
NAND Flash Driver

Chapter 23
SPI NOR Flash Memory Technology Device (MTD) Driver

Chapter 24
Low-Level Keypad Driver

Chapter 25
Fast Ethernet Controller (FEC) Driver

Chapter 26
Inter-IC (I2C) Driver

Chapter 27
Configurable Serial Peripheral Interface (CSPI) Driver

Chapter 28
1-Wire Driver

Chapter 29
MMC/SD/SDIO Host Driver

Chapter 30
ARC USB Driver

Chapter 31
Universal Asynchronous Receiver/Transmitter (UART) Driver

Chapter 32
Secure Real Time Clock (SRTC) Driver

Chapter 33
Watchdog (WDOG) Driver

Chapter 34
Pulse-Width Modulator (PWM) Driver

Chapter 35
HDMI Driver

Chapter 36
Frequently Asked Questions

Chapter 37
OProfile

Tables

Figures

About This Book

The Linux Board Support Package (BSP) represents a porting of the Linux Operating System (OS) to the i.MX processors and its associated reference boards. The BSP supports many hardware features on the platforms and most of the Linux OS features that are not dependent on any specific hardware feature.

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 2.6 kernel internals, driver models, and i.MX processors.

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.

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
ARM®	Advanced RISC Machines processor architecture
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

Definitions and Acronyms (continued)

Term	Definition
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
IPU	Image Processing Unit —supports video and graphics processing functions and provides an interface to video/still image sensors and displays
IrDA	Infrared Data Association—a nonprofit organization whose goal is to develop globally adopted specifications for infrared wireless communication
ISR	Interrupt Service Routine

Definitions and Acronyms (continued)

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
MSHC	Memory Stick Host Controller
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

Definitions and Acronyms (continued)

Term	Definition
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
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

Suggested Reading

The following documents contain information that supplements this guide:

- *i.MX50_RD3_Linux_BSP_UserGuide.pdf*
- *MCIMX50 Multimedia Applications Processor Reference Manual (MCIMX50RM)*

Chapter 1
Machine Specific Layer (MSL)

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

- Interrupts including GPIO
- Timer
- Memory map
- General Purpose Input/Output (GPIO) including IOMUX
- Smart Direct Memory Access (SDMA)
- Direct Memory Access(DMA)

These modules are normally available in the following directory:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5 for MX5 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 operation 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) are detailed. Because of the complexity of the SDMA module, its design is explained in Chapter 2, “Smart Direct Memory Access (SDMA) API.” MX50 also uses DMA module, which is explained in Chapter 3, “Direct Memory Access Controller (DMAC) API.

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

1.1 Interrupts

The following sections explain the hardware and software operation of interrupts on the device.

1.1.1 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 setting.

Interrupt Controller registers can only be accessed in supervisor mode. The Interrupt Controller interrupt requests are prioritized in the order of fast interrupts, and normal interrupts in order of highest priority level, then highest source number with the same priority. There are sixteen 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.

1.1.2 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 has a description of 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 `irq_chip` structure and exports one initialization function, which is called during system startup.

1.1.3 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)

1.1.4 Interrupt Source Code Structure

The interrupt module is implemented in the following file:

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

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

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

Table 1-1 lists the source files for interrupts.

Table 1-1. Interrupt Files

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

1.1.5 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 irq_desc`. 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 from the Interrupt Controller, the number of interrupts is also expanded to support GPIO interrupt. This allows drivers to use the standard interrupt interface supported by ARM Linux, such as the `request_irq()` and `free_irq()` functions.

1.2 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 does the following:

- 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.

1.2.1 Timer Hardware Operation

The General Purpose Timer (GPT) has a 32 bit up-counter. The timer counter value can be captured in a register using an event on an external pin. The capture trigger can be programmed to be a rising or falling edge. The GPT can also generate an event on `ipp_do_cmpout` pins, or can produce an interrupt when the timer reaches a programmed value. It has a 12-bit prescaler providing a programmable clock frequency derived from multiple clock sources.

1.2.2 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 Section 1.2, “Timer.” Another function provides the time elapsed as the last timer interrupt.

1.2.3 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.

1.2.4 Timer Source Code Structure

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

1.3 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.

1.3.1 Memory Map Hardware Operation

The MMU, as 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.

1.3.2 Memory Map Software Operation

A table mapping the virtual memory to physical memory is implemented for i.MX platforms as defined in the `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mm.c` file.

1.3.3 Memory Map Features

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

1.3.4 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 IO module physical and virtual base addresses and physical to virtual mapping macros. All of the memory map source code is in the in the following directories:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/include/mach
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5
```

Table 1-2 lists the source file for the memory map.

Table 1-2. Memory Map Files

File	Description
mx5x.h	Header files for the IO module physical addresses
mm.c	Memory map definition file

1.3.5 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.

1.4 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 `UART1_TXD` pin might have the following functions:

- `UART1_TXD`–internal UART1 Transmit Data. This is the primary function of this pin.
- `GPIO6[6]`—alternate mode 1
- `USBPHY1 DATAOUT[14]`—alternate mode 7

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, then it should be configured as GPIO input pin with interrupt enabled. Again, be aware that the software does not have control over what function a pin should have. The software only configures pin usage according to the system design.

1.4.1 IOMUX Hardware Operation

The IOMUX controller registers are briefly described here. For detailed information, refer to the pin multiplexing section of the IC Reference Manual.

- SW_MUX_CTL—Selects the primary or alternate function of a pin. Also 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—Control pad slew rate, driver strength, pull-up/down resistance, and so on.

1.4.2 IOMUX Software Operation

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

1.4.3 IOMUX Features

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

1.4.4 IOMUX Source Code Structure

Table 1-3 lists the source files for the IOMUX module. The files are in the directory:

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

Table 1-3. IOMUX Files

File	Description
iomux-v3.c	IOMUX function implementation
include/mach/iomux-mx50.h	Pin definitions in the iomux pins

1.4.5 IOMUX Programming Interface

The iomux api is in arch/arm/plat-mxc/include/mach/iomux-v3.h. Read the comments at the head of this file to understand the iomux scheme.

1.5 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.

1.5.1 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 in the MSL layer to provide better portability and maintainability. This GPIO interrupt is initialized automatically during the system startup.

If a pin is configured as GPIO by the IOMUX, the state of the pin should also be set since it is not initialized by a dedicated hardware module.

1.5.1.1 API for GPIO

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. The macro `IOMUX_TO_IRQ_V3()` or `gpio_to_irq()` is used to convert GPIO pin to irq number,
- Functions to set an IOMUX pin, named `mxc_iomux_v3_setup_pad()`. If a pin is used as GPIO, another set of request/free function calls are provided, named `gpio_request()` and `gpio_free()`. 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. Furthermore, functions `gpio_direction_input()` and `gpio_direction_output()` are provided to set GPIO when it's used as input or output. See the API document and `Documentation/gpio.txt` for more details.

1.5.2 GPIO Features

This GPIO implementation supports the following features:

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

1.5.3 GPIO Source Code Structure

GPIO driver is implemented based on general gpiolib framework. The MSL-layer codes defines and registers gpio_chip instances for each bank of on-chip GPIOs, in the following files, located in the directories indicated at the beginning of this chapter:

Table 1-4. GPIO Files

File	Description
gpio.h	GPIO public header file
gpio.c	Function implementation

1.5.4 GPIO Programming Interface

For more information, see the API documents and `Documentation/gpio.txt` for the programming interface.

Chapter 2
Smart Direct Memory Access (SDMA) API

2.1 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, DSP 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

2.2 Hardware Operation

The SDMA controller is responsible for transferring data between the MCU memory space, the DSP memory space and peripherals and includes the following features.

- Multi-channel DMA supporting up to 32 time-division multiplexed DMA channels
- Powered by a 16-bit Instruction-Set microRISC engine
- Each channel executes specific script
- Very fast context-switching with two-level priority based preemptive multi-tasking
- 4 Kbytes ROM containing startup scripts (that is, boot code) and other common utilities that can be referenced by RAM-located scripts
- 8 Kbyte 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.

2.3 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.

Complete support for SDMA is provided in three layers (see Figure 2-1):

- I.API
- Linux DMA API
- TTY driver or DMA-capable drivers, such as ATA, SSI and the UART driver.

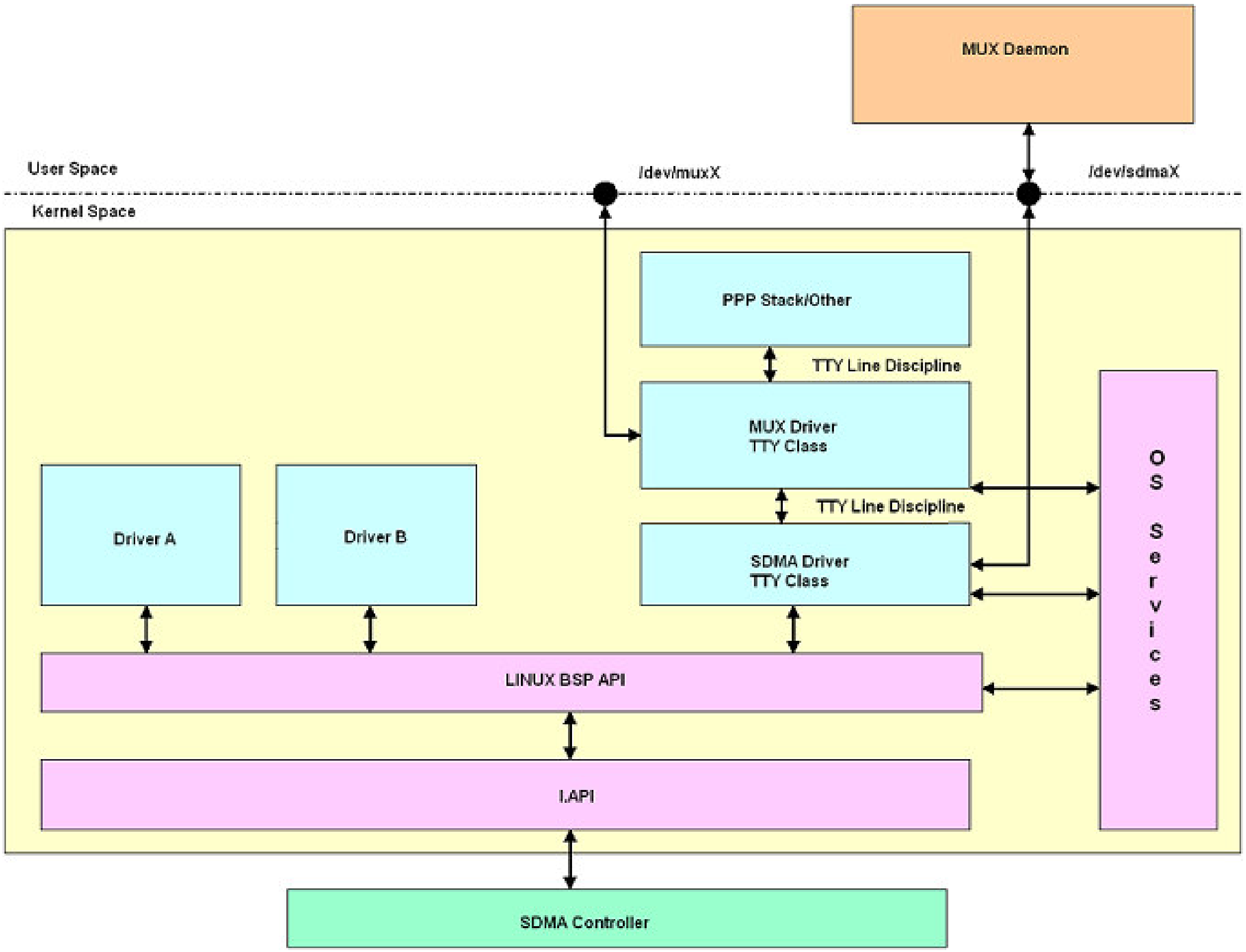

Figure 2-1. SDMA Block Diagram

The first two layers are part of the MSL and customized for each platform. I.API is the lowest layer and it interfaces with the Linux DMA API with the SDMA controller. The Linux DMA API interfaces other drivers (for example, MMC/SD, Sound) with the SDMA controller through the I.API.

Table 2-1 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 (static channel allocation) or can have the SDMA driver provide a free SDMA channel for the driver to use (dynamic channel allocation). For dynamic channel allocation, the list of SDMA channels is scanned from channel 32 to channel 1. On finding a free channel, that channel is allocated for the requested DMA transfers.

Table 2-1. 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

2.4 Source Code Structure

The source file, `sdma.h` (header file for SDMA API) is available in the directory `/<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/include/mach`.

Table 2-2 shows the source files available in the directory, `/<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/sdma`.

Table 2-2. SDMA API Source Files

File	Description
sdma.c	SDMA API functions
sdma_malloc.c	SDMA functions to get memory that allows DMA
iapi/	iAPI source files

Table 2-3 shows the header files available in the directory, `/<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/`.

Table 2-3. SDMA Script Files

File	Description
sdma_script_code_mx50.h	SDMA RAM scripts for i.MX50

2.5 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_MXC_SDMA_API—This is the configuration option for the SDMA API driver. In menuconfig, this option is available under

  System type > Freescale MXC implementations > MX5x Options: > Use SDMA API.

  By default, this option is Y.

- CONFIG_SDMA_IRAM—This is the configuration option to support Internal RAM as SDMA buffer or control structures. This option is available under System type > Freescale MXC implementations > MX5x Options > Use Internal RAM for SDMA transfer.

2.6 Programming Interface

The module implements custom API and partially standard DMA API. Custom API is needed for supporting non-standard DMA features such as loading scripts, interrupts handling and DVFS control. Standard API is supported partially. It can be used along with custom API functions only. Refer to the API document for more information on the functions implemented in the driver (in the doxygen folder of the documentation package).

2.7 Usage Example

Refer to one of the drivers from Table 2-1 that uses the SDMA API driver for a usage example.

Chapter 3
Direct Memory Access Controller (DMAC) API

The Direct Memory Access Controller (DMAC) provides 16 channels supporting linear memory, 2D memory, and FIFO transfers to provide support for a wide variety of DMA operations.

3.1 Hardware Operation

The i.MX50 device is equipped with AHB-to-APBH bridge with built-in DMA capability that allows programmed data transfers between SDRAM and peripheral devices. The DMA is abstracted as a number of channels dedicated to on-chip peripheral devices such as GPMI. Each DMA channel is programmed by a set of per-channel registers and a special DMA command structure located in memory. A command describes a single DMA transaction and can be chained with other commands to set up multiple DMA transfers.

Each DMA channel implements a semaphore used to start and stop the DMA channels. The semaphore may contain values from 0 to 255 that are set by software. The DMA channel starts transferring data on writing a semaphore value greater than zero and continues operation until the semaphore is decremented to zero or an error occurs. The semaphore is decremented after completion of a single DMA transfer if the corresponding flag is set within the command structure.

The DMA includes the following features:

- Sixteen channels support linear memory, 2D Memory, and FIFO for both source and destination
- DMA chaining for variable length buffer exchanges and high allowable interrupt latency requirement
- Increment, decrement, and no-change support for source and destination addresses
- Each channel is configurable to response to any of the DMA request signals
- Supports 8, 16, or 32-bit FIFO and memory port size data transfers
- DMA burst length configurable up to a maximum of 16 words, 32 half-words, or 64 bytes for each channel
- Bus utilization control for the channel that is not triggered by a DMA request
- Burst time-out errors terminate the DMA cycle when the burst cannot be completed within a programmed time count
- Buffer overflow error terminates the DMA cycle when the internal buffer receives more than 64 bytes of data
- Transfer error terminates the DMA cycle when a transfer error is detected during a DMA burst

3.2 Software Operation

Prior to using a DMA channel, the driver should register an interrupt handler for interrupts generated by the DMA channel in order to receive DMA error or completion events.

The most used scenario of DMA operation is when a device driver wants to transfer a number of bytes to or from a memory buffer located on SDRAM. First, it allocates and initializes a DMA command structure or a list of command structures for multiple transfers. Then it resets the DMA channel and configures the channel registers to point to a command structure for the first DMA transfer. When all the required initialization is done, the DMA channel is started by setting a DMA channel semaphore.

The module provides an API for other drivers to control DMA channels. The DMA software operations are as follows:

- Requesting DMA channel
- Initialization of the channel
- Setting configuration of DMA channel
- Enabling/Disabling DMA
- Getting DMA transfer status
- DMA IRQ handler

3.3 Source Code Structure

The header file, `dmaengine.h`, is available in the directory:

`arch/arm/plat-mxc/include/mach/`

Table 3-1 lists the source files available in the directory, `arch/arm/plat-mxc/ and arch/arm/mach-mx5`.

Table 3-1. DMA API Files

File	Description
`dma-apbh.c`	Parameters of DMA channels
`dmaengine.c`	DMA API functions

3.4 Programming Interface

The module implements custom DMA API. Standard API is not supported. Refer to the doxygen files in the release notes for more information on the methods implemented in the driver.

Chapter 4
MAX17135 Temperature Sensor Driver

The MAX17135 temperature sensor driver provides low-level control of the temperature sensor functionality of the MAX17135 power management IC (PIMC). The MAX17135 is a complete PMIC for E-paper displays, featuring source and gate driver power supplies, a high-speed VCOM amplifier. The MAX17135 PMIC also includes 2 temperature sensors: one for the internal IC temperature and another for the external temperature. The MAX17135 temperature sensor driver adheres to the Linux hardware monitoring framework, using the sysfs interface to query temperature levels. The MAX17135 PMIC is accessed through an I2C interface.

As a product early in the production stages, the MAX17135 has undergone multiple early passes to correct bugs and limitations found in the early hardware. This driver supports the Pass 1 and Pass 2 version of the MAX17135.

4.1 Hardware Operation

The MAX17135 includes a temperature sensor that reads the internal MAX17135 temperature and the external panel temperature with the use of an external temperature sensing diode. Temperature output data is supplied via I2C. An analog-to-digital converter converts the temperature data to 9 bits, two’s complement format and stores the conversion results in separate temperature registers.

Additional details of the MAX17135 hardware operation may be found in the MAX17135 datasheet.

4.2 Driver Features

The MAX17135 PMIC temperature sensor driver is based on the Linux hardware monitoring core driver. It provides the following services for querying temperature:

- Get the PMIC internal temperature
- Get the PMIC external ambient temperature

4.3 Software Operation

The MAX17135 temperature sensor driver is designed as within the hardware monitoring device framework and is accessed through the sysfs file interface. An I2C interface is used for low-level access to the MAX17135 hardware registers. This section details the MAX17135 driver architecture and the sensor API.

4.3.1 Driver Architecture

Figure 4-1 shows the basic architecture for MAX17135 drivers.

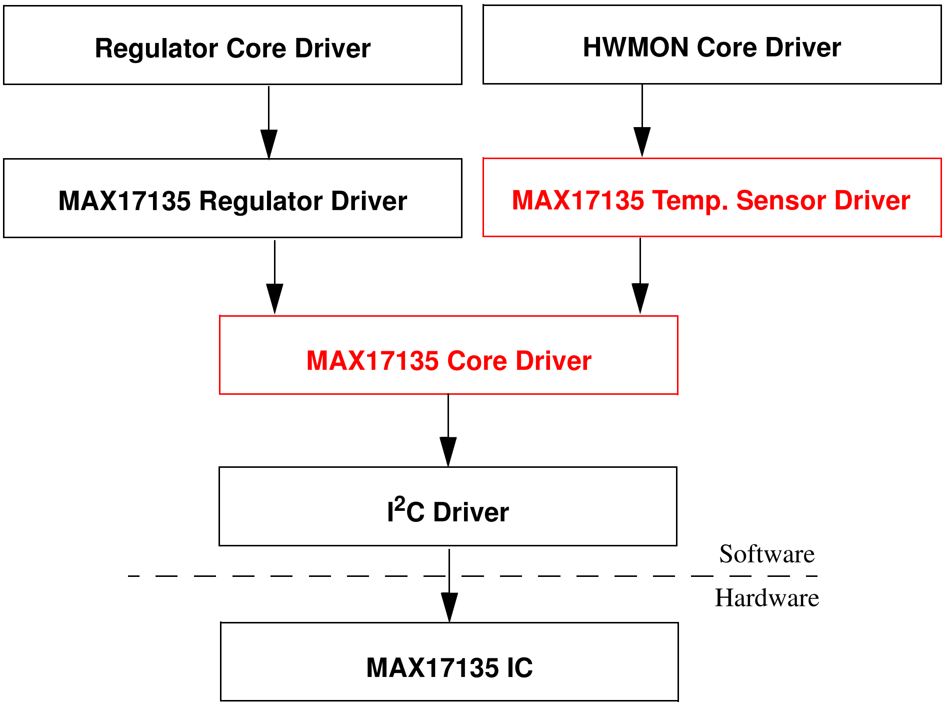

Figure 4-1. MAX17135 Driver Architecture

The MAX17135 Core Driver registers the MAX17135 as an I2C client, establishing the driver's low-level communication API with the MAX17135 hardware (via I2C). Later in the kernel boot-up sequence, the MAX17135 sensor driver is registered. This causes `max17135_sensor_probe`() to execute, which in turn results in calls to `sysfs_create_group`() and to `hwmon_device_register()`, which registers the temperature sensor as a hardware monitoring device. This establishes the driver's high-level API to other drivers and applications (via the sysfs file interface).

4.3.2 Temperature Sensor API

Hardware monitoring device drivers expose entries and data through sysfs files. This provides a simple and consistent way for application programs to scan for sensor data. For additional information about the hardware monitoring devices sysfs interface standard, please see the kernel documentation here:

/documentation/hwmon/sysfs-interface

To find the MAX17135 temperature sensor sysfs files, the correct hwmon device must first be identified. The following directory contains the data for all of the sensor chips supported on a given platform:

/sys/class/hwmon/hwmon*

The MAX17135 may identified by checking the following file for the identifier "max17135_sensor":

/sys/class/hwmon/hwmon*/device/modalias

Once the MAX17135 sensor device directory has been identified, the temperature value can be read from the MAX1735 in the following files:

- temp1_input - Internal MAX17135 temperature (in degrees Celsius)
- temp2_input - External ambient temperature (in degrees Celcius)

4.4 Source Code Structure

The MAX17135 core driver is located in the MFD (Multi-function device drivers) driver directory:

`<ltib_dir>/rpm/BUILD/linux/drivers/mfd.`

Table 4-1. MAX17135 Core Driver Files

File	Description
max17135-core.c	Core support for MAX17135 register access via I2C.
max17135.h	Common header file for all MAX17135 drivers.

The MAX17135 temperature sensor driver is located in the hardware monitoring device driver directory:

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

Table 4-2. MAX17135 Temperature Sensor Driver Files

File	Description
hwmon.c	Core support for hardware monitoring devices.
max17135-hwmon.c	Implementation of the MAX17135 temperature sensor driver

The MAX17135 temperature sensor for the MX50 Armadillo 2 board is registered under `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mx50_arm2.c.`

The MAX17135 temperature sensor for the MX50 EVK board are registered under `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mx50_rdp.c.`

4.5 Menu Configuration Options

The following Linux kernel configurations are provided for the MAX17135 temperature sensor driver. To get to the PMIC power configuration, use the command `./ltib -c` when located in the `<ltib dir>`. On the configuration screen select **Configure Kernel**, exit, and when the next screen appears, choose.

- CONFIG_MFD_MAX17135—This is the configuration option to include MAX17135 core support in the Linux kernel. In menuconfig, this option is available under:

  Device Drivers > Multifunction device drivers > MAX17135 PMIC core

  By default, this option is Y for all architectures.
- CONFIG_SENSORS_MAX17135—This is the configuration option for the MAX17135 temperature sensor driver. In menuconfig, this option is available under:

  Device Drivers > Hardware Monitoring support > Maxim MAX17135

  By default, this option is Y for all architectures.

Chapter 5
MAX17135 Regulator Driver

The MAX17135 regulator driver provides low-level control of the regulator control capability of the MAX17135 PMIC. The MAX17135 PMIC provides power supply regulators for E Ink displays, including source and gate driver power supplies, a high-speed VCOM amplifier and a temperature sensor. The MAX17135 regulator driver is designed within the Linux voltage and current regulator framework, providing a familiar and uniform interface to enable and disable power supply regulators and control voltage levels. Temperature sensor control is provided through a separate temperature sensor driver, details of which may be found in the MAX17135 Temperature Sensor Driver document. The MAX17135 PMIC is accessed through an I2C interface.

As a product early in the production stages, the MAX17135 has undergone multiple early passes to correct bugs and limitations found in the early hardware. This driver supports the Pass 1 and Pass 2 version of the MAX17135.

5.1 Hardware Operation

The MAX17135 provides all of the power management components necessary for the use of an E Ink display: source and gate power driver supplies, a high-speed VCOM amplifier, and a temperature sensor.

A boost converter and an inverting buck-boost converter comprise the source driver power supplies. These supplies generate +15V and -15V, respectively. The positive source driver supply regulation voltage (Vpos) is set through the I2C interface, and the negative source driver supply voltage is tightly regulated to -Vpos within 50mV.

The gate driver power supplies consist of regulated charge pumps that generate +22V and -20V and can deliver up to 20mA each.

The VCOM amplifier output is controlled by an 8-bit digital to analog converter (DAC). This DAC is programmable via an I2C interface and allows small voltage step sizes per DAC step.

Additional details of the MAX17135 hardware operation may be found in the MAX17135 datasheet.

5.2 Driver Features

The MAX17135 PMIC regulator driver is based on the Linux regulator core driver. It provides the following services for regulator control of the PMIC component:

- Switch ON/OFF all voltage regulators
- Switch ON/OFF for the VCOM amplifier
- Set the value for all voltage regulators
- Set the value for the VCOM amplifier

- Get the current value for all voltage regulators
- Get the current value for the VCOM amplifier

5.3 Software Operation

The MAX17135 driver is designed to be accessed through the Linux regulator core driver. An I2C interface is used for low-level access to the MAX17135 hardware registers. This section details the MAX17135 driver architecture and API. Also covered are some important aspect of the driver's operation (how it is initialized) and configuration (key platform-level configuration parameters).

5.3.1 Driver Architecture

Figure 5-1 shows the basic architecture of the drivers that enable the functionality of the MAX17135. regulator driver.

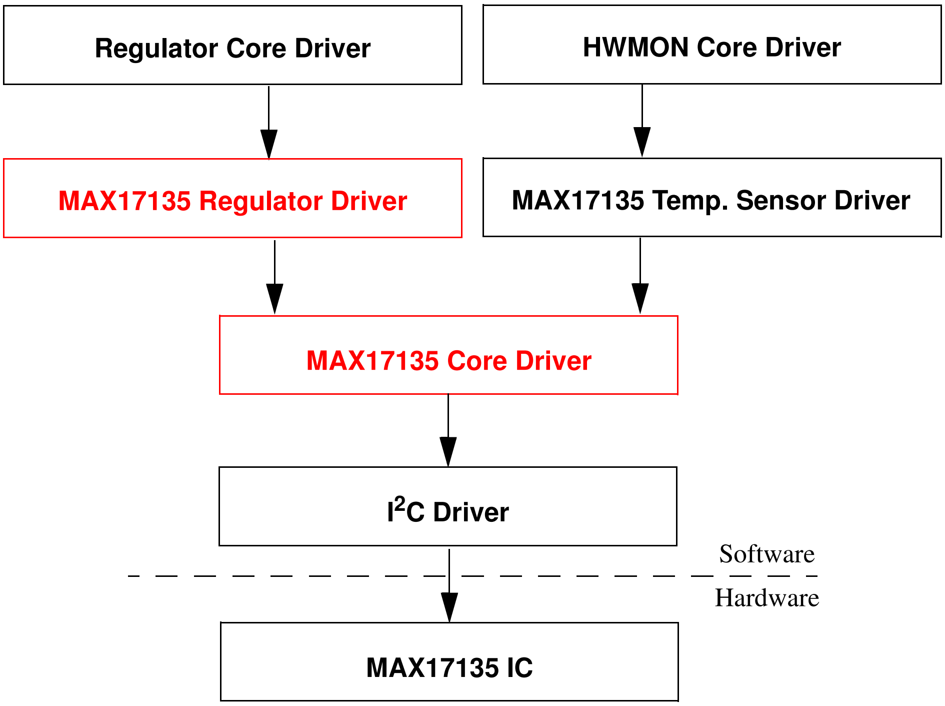

Figure 5-1. MAX17135 Driver Architecture

The MAX17135 Core Driver registers the MAX17135 as an I2C client, establishing the driver's low-level communication API with the MAX17135 hardware (via I2C). The MAX17135 Core Driver then calls the platform-level `Init()` function which initializes the MAX17135 Regulator Driver. This regulator driver registers the individual regulators and establishes the high-level API to other drivers and applications (via the core regulator API).

5.3.2 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—lookup and obtain a reference to a regulator

  `struct regulator *regulator_get(struct device *dev, const char *id);`
- regulator_put—free the regulator source

  `void regulator_put(struct regulator *regulator, struct device *dev);`
- regulator_enable—enable regulator output

  `int regulator_enable(struct regulator *regulator);`
- regulator_disable—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—set regulator output voltage

  `int regulator_set_voltage(struct regulator *regulator, int uV);`
- regulator_get_voltage—get regulator output voltage

  `int regulator_get_voltage(struct regulator *regulator);`

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`.

5.3.3 Driver Configuration

The MAX17135 driver is designed to be ported easily from one platform to another. This goal is facilitated through the `max17135_platform_data` structure, which defines all of the platform-specific information needed by the driver. This structure should be defined in the platform definition file and provided as a parameter for the MAX17135 driver registration.

```
struct max17135_platform_data {
        unsigned int gvee_pwrup;
        unsigned int vneg_pwrup;
        unsigned int vpos_pwrup;
        unsigned int gvdd_pwrup;
        unsigned int gvdd_pwrdn;
        unsigned int vpos_pwrdn;
        unsigned int vneg_pwrdn;
        unsigned int gvee_pwrdn;
        int gpio_pmic_pwrgood;
        int gpio_pmic_vcom_ctrl;
        int gpio_pmic_wakeup;
```

```
               int gpio_pmic_intr;
               int pass_num;
               int vcom_uV;
               struct regulator_init_data *regulator_init;
               int (*init)(struct max17135 *);
      };
```

The `max17135_platform_data` structure consists of a series of voltage supply timing values, a set of GPIO definitions, a number identifying which pass of the MAX17135 is being used, a VCOM value to be used by the driver, and a regulator_init_data structure to define the constraints of the MAX17135 regulators.

`gvee_pwrup`, `vneg_pwrup`, `vpos_pwrup`, and `gvdd_pwrup` provide the time (in milliseconds) required for each regulator to power up. Conversely, `gvee_pwrup`, `vneg_pwrdn`, `vpos_pwrdn`, and `gvdd_pwrdn` provide the time (in milliseconds) required for each regulator to power down.

`gpio_pmic_pwrgood`, `gpio_pmic_vcom_ctrl`, `gpio_pmic_wakeup`, and `gpio_pmic_intr` identify the appropriate GPIO number used for each pin.

`pass_num` identifies which pass of MAX17135 is being used on the current platform. Currently, only pass 1 and pass 2 parts are supported. Specifying the correct pass number is important because the VCOM voltage register setting is configured differently from one MAX17135 pass to another. Specifying the wrong MAX17135 pass number could result in an incorrect VCOM value and subsequently damage the E Ink panel.

The `vcom_uV` parameter specifies the VCOM voltage (in microvolts) that is required by the E Ink panel on the platform. This value is configured one time (the first time that the VCOM is enabled) and should not be changed after that.

A kernel option makes `pass_num` and `vcom_uV` easy to configure at run-time. The option should be specified using the following string:

```
max17135:pass=[pass_num],vcom=[vcom_uV]
```

The `regulator_init` parameter specifies the voltage constraints for the MAX17135 voltage supplies.

The init() function should be defined to fill out the `max17135` structure and call to the MAX17135 Regulator Driver to register the MAX17135 regulators.

5.3.4 Driver Operation Details

5.3.4.1 Driver Initialization

The MAX17135 core driver must be registered at the platform level as an I2C client using the string ID “max17135”. When the I2C driver is registered, `max17135_probe`() is invoked. This function creates the `max17135` structure and fills in the i2c_client field. The `max17135_platform_data init()` function is then invoked, which fills out the rest of the `max17135` structure. Next, `max17135_register_regulator` is invoked for each MAX17135 regulator. This function adds a platform device for each of the regulators defined in `regulator_init`. The probe function for each regulator (`max17135_regulator_probe`) then in turn calls `regulator_register`() to register each regulator and make them accessible via the core regulator APIs.

5.3.4.2 Driver Access

Typical access to the MAX17135 involves enabling and disabling the two key regulators registered by the MAX17135 driver: DISPLAY and VCOM. The DISPLAY regulator, when enabled, turns on all of the voltage supplies to the E Ink display. The VCOM regulator enables the VCOM amplifier and sets the VCOM voltage to the value specified in `vcom_uV`. Disabling the MAX17135 is a key element in power management for a system. Therefore, the DISPLAY and VCOM regulators are typically only enabled when the E Ink display needs to be updated, and are disabled once the E Ink display is no longer being updated.

5.4 Source Code Structure

The MAX17135 core driver is located in the MFD (Multi-function device drivers) driver directory:

`<ltib_dir>/rpm/BUILD/linux/drivers/mfd`.

Table 5-1. MAX17135 Core Driver Files

File	Description
max17135-core.c	Core support for MAX17135 register access via I2C.
max17135.h	Common header file for all MAX17135 drivers.

The MAX17135 regulator driver is located in the regulator device driver directory:

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

Table 5-2. MAX17135 Regulator Driver Files

File	Description
core.c	Linux kernel interface for regulators.
max17135-regulator.c	Implementation of the MAX17135 regulator client driver

The MAX17135 regulators for the MX50 Armadillo 2 board are registered under `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mx50_arm2.c.`

The MAX17135 regulators for the MX50 EVK board are registered under `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mx50_rdp.c.`

5.5 Menu Configuration Options

The following Linux kernel configurations are provided for the MAX17135 Regulator driver. To get to the PMIC power configuration, use the command `./ltib -c` when located in the `<ltib dir>`. On the configuration screen select **Configure Kernel**, exit, and when the next screen appears, choose.

- CONFIG_MFD_MAX17135—This is the configuration option to include MAX17135 core support in the Linux kernel. In menuconfig, this option is available under:

  Device Drivers > Multifunction device drivers > MAX17135 PMIC core

  By default, this option is Y for all architectures.

- CONFIG_REGULATOR_MAX17135—This is the configuration option for the MAX17135 regulator driver. This option is dependent on the CONFIG_MFD_MAX17135 option. In menuconfig, this option is available under:

  Device Drivers > Voltage and Current regulator support > Maxim MAX17135 Regulator Support

  By default, this option is Y for all architectures.

Chapter 6
MC34708 Regulator Driver

The MC34708 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 PMIC protocol driver to access the PMIC hardware control registers.

6.1 Hardware Operation

The MC34708 provides reference and supply voltages for the application processor as well as peripheral devices.The buck converters provide the power supply to processor cores and to other low voltage circuits such as I/O and memory. Dynamic voltage scaling is provided to allow controlled supply rail adjustments for the processor cores and/or other circuitry. Two DVS control pins are provided for pin controlled DVS on the buck switchers targeted for processor core supplies.

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

6.2 Driver Features

The MC34708 PMIC regulator driver is based on the PMIC protocol driver and regulator core driver. It provides the following services for regulator control of the PMIC component:

- Switch ON/OFF all voltage regulators
- Switch ON/OFF all BUCK switchers
- Set the value for all voltage regulators
- Get the current value for all voltage regulators

6.3 Software Operation

The PMIC power management driver and the MC34708 PMIC regulator client driver perform operations by reconfiguring the PMIC hardware control registers. This is done by calling protocol 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, then any changes that use the power management driver and the regulator client driver can affect the operation or stability of the entire system.

6.4 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—lookup and obtain a reference to a regulator
  ```
  struct regulator *regulator_get(struct device *dev, const char *id);
  ```
- regulator_put—free the regulator source
  ```
  void regulator_put(struct regulator *regulator, struct device *dev);
  ```
- regulator_enable—enable regulator output
  ```
  int regulator_enable(struct regulator *regulator);
  ```
- regulator_disable—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—set regulator output voltage
  ```
  int regulator_set_voltage(struct regulator *regulator, int uV);
  ```
- regulator_get_voltage—get regulator output voltage
  ```
  int regulator_get_voltage(struct regulator *regulator);
  ```

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`.

6.5 Driver Architecture

Figure 6-1 shows the basic architecture of the MC34708 regulator driver.

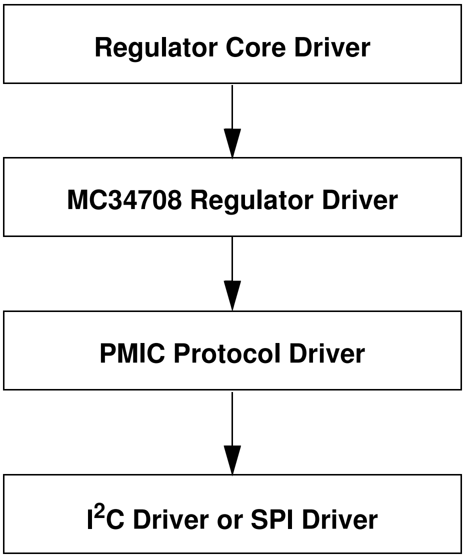

Figure 6-1. MC34708 Regulator Driver Architecture

6.6 Driver Interface Details

Access to the MC34708 regulator is provided through the API of the regulator core driver. The MC34708 regulator driver provides the following regulator controls:

- Buck switch supplies
  - — Five buck switch regulators on normal mode: SWx, where x = 1–5
  - — Five buck switch regulators on standby mode: SWx_ST, where x = 1–5
  - — Five buck switch regulators on DVFS mode: SWx_ST, where x = 1–5
- Linear Regulators

  VGEN1, VPLL, VDAC, VGEN2, VUSB and VUSB2

All of the regulator functions are handled by setting the appropriate PMIC hardware register values. This is done by calling the PMIC protocol driver APIs to access the PMIC hardware registers.

6.7 Source Code Structure

The MC34708 regulator driver is located in the regulator device driver directory:

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

Table 6-1. MC34708 Power Management Driver Files

File	Description
core.c	Linux kernel interface for regulators.
reg-mc34708.c	Implementation of the MC34708 regulator client driver

The MC34708 regulators for MX50 RDP board are registered under `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mx50_rdp_pmic_mc34708.c.`

6.8 Menu Configuration Options

The following Linux kernel configurations are provided for the MC34708 Regulator driver. To get to the PMIC power configuration, use the command `./ltib -c` when located in the `<ltib dir>`. On the configuration screen select **Configure Kernel**, exit, and when the next screen appears, choose.

- Device Drivers > Voltage and Current regulator support > MC34708 Regulator Support.

Chapter 7
MC34708 Digitizer Driver

This chapter describes the Linux PMIC Digitizer Driver that provides low-level access to the PMIC analog-to-digital converters (ADC). This capability includes taking measurements of the X-Y coordinates and contact pressure from an attached touch panel. This device driver uses the PMIC protocol driver to access the PMIC hardware control registers that are associated with the ADC.

The PMIC digitizer driver is used to provide access to and control of the analog-to-digital converter (ADC) that is available with the PMIC. Multiple input channels are available for the ADC, and some of these channels have dedicated functions for various system operations. For example:

- Sampling the voltages on the touch panel interfaces to obtain the (X,Y) position and pressure measurements
- Battery voltage level and current monitoring

The PMIC ADC has a 10-bit resolution and supports either a single channel conversion or automatic conversion of all input channels in succession. The conversion can be triggered by issuing a command.

A hardware interrupt can be generated following the completion of an ADC conversion. Some PMIC chips also provide a pulse generator that is synchronized with the ADC conversion. The pulse generator can enable or drive external circuits in support of the ADC conversion process.

The PMIC ADC components are subject to arbitration rules as documented in the documentation for each PMIC. These arbitration rules determine how requests from both primary and secondary SPI interfaces are handled. SPI bus arbitration configuration and control is not part of this driver because the platform has configured arbitration settings as part of the normal system boot procedure. There is no need to dynamically reconfigure the arbitration settings after the system has been booted.

7.1 Driver Features

The PMIC Digitizer Driveris is a client of the PMIC protocol driver. The PMIC protocol driver provides hardware control register reads and writes through the SPI bus interface and also register/deregister event notification callback functions. The PMIC protocol driver requires access to ADC-specific event notifications.

The PMIC Digitizer Driver supports the following features for supporting a touch panel device:

- Selects either a single ADC input channel or an entire group of input channels to be converted
- Starts an ADC conversion by issuing the appropriate start conversion command
- Enable/disables hardware interrupts for all ADC-related event notifications
- Provides an interrupt handler routine that receives and properly handles all ADC end-of-conversion

- Other device drivers register/deregister additional callback functions to provide custom handling of all ADC-related event notifications
- Provides a read-only device interface for passing touchpanel (X,Y) coordinates and pressure measurements to applications
- Provides the ability to read out one or more ADC conversion results
- Implements the appropriate input scaling equations so that the ADC results are correct
- Specifies the delay between successive ADC conversion operations, if supported by the PMIC. For PMIC chips that do not support this feature, the device driver returns a NOT_SUPPORTED status
- Provides support for a pulse generator that is synchronized with the ADC conversion. For PMIC chips that do not support this feature, returns a NOT_SUPPORTED status
- Provides a complete IOCTL interface to initiate an ADC conversion operation and to return the conversion results
- Provides support for a polling method to detect when the ADC conversion has been completed

This digitizer driver is not responsible for any additional ADC-related activities such as battery level or current. Such functions are handled by other PMIC-related device drivers. Also, this device driver is not responsible for SPI bus arbitration configuration. The appropriate arbitration settings that are required in order for this device driver to work properly are expected to have been set during the system boot process.

7.2 Software Operation

Most of the required operations for this device driver simply involve writing the correct configuration settings to the appropriate PMIC control registers. This can be done by using the APIs that are provided with the PMIC protocol driver.

Once an ADC conversion has been started, suspend the calling thread until the conversion has been completed. Avoid using a busy loop since this negatively impacts processor and overall system performance. Instead, the use of a wait queue offers a much better solution. Therefore, any potentially time-consuming operations results in the calling thread being placed into a wait queue until the operation is completed.

The PMIC ADC conversion can take a significant amount of time. The delay between a start of conversion request and a conversion completed event may even be open ended, if the conversion is not started until the appropriate external trigger signal is received. Therefore, all ADC conversion requests must be placed in a wait queue until the conversion is complete. Once the ADC conversion has completed, the calling thread can be removed from the wait queue and reawakened.

Avoid the use of any polling loops or other thread delay tactics that would negatively impact processor performance. Also, avoid doing anything that prevents hardware interrupts from being handled, because the ADC end-of-conversion event is typically signalled by a hardware interrupt.

7.3 Source Code Structure

Table 7-1 lists the source files for the MC34708-specific version of this driver. These are contained in the following directories:

```
<ltib_dir>/rpm/BUILD/linux/drivers/mxc/pmic/mc34708/mc34708_adc.c
```

```
<ltib_dir>/rpm/BUILD/linux/include/linux/mc34708_adc.h
<ltib_dir>/rpm/BUILD/linux/drivers/input/touchscreen/mxc_ts.c
```

Table 7-1. MC34708 Digitizer Driver Files

File	Description
mc34708_adc.c	Implementation of the MC34708 ADC client driver
mc34708_adc.h	Define names of IOCTL user space interface
mxc_ts.c	Common interface to the input driver system

7.4 Menu Configuration Options

The following Linux kernel configurations are provided. To get to the configurations, use the command ./ltib -c when located in the <ltib dir>. In the screen select **Configure Kernel**, exit, and a new screen appears.

- Choose the MC34708 (MC34708) specific digitizer driver for the PMIC ADC. In menuconfig, this option is available under:

  Device Drivers > MXC Support Drivers > MXC PMIC Support > MC34708 ADC support
- Driver for the MXC touch screen. In menuconfig, this option is available under:

  Device Drivers > Input device support > Touchscreens > MXC touchscreen input driver

Chapter 8
MC34708 RTC Driver

The Linux MC34708 RTC driver provides access to the MC34708 RTC control circuits. This device driver makes use of the MC34708 protocol driver to access the MC34708 hardware control registers. The MC34708 device is used for real-time clock control and wait alarm events.

8.1 Driver Features

The MC34708 RTC driver is a client of the MC34708 protocol driver. It provides the services for real time clock control of MC34708 components. The driver is implemented under the standard RTC class framework.

8.2 Software Operation

The MC34708 RTC driver performs operations by reconfiguring the MC34708 hardware control registers. This is done by calling protocol driver APIs with the required register settings.

8.3 Driver Implementation Details

Configuring the MC34708 RTC driver includes the following parameters:

- Set time of day and day value
- Get time of day and day value
- Set time of day alarm and day alarm value
- Get time of day alarm and day alarm value
- Report alarm event to the client

8.3.1 Driver Access and Control

To access this driver, open the `/dev/rtcN` device to allow application-level access to the device driver using the IOCTL interface, where the `N` is the RTC number. /sys/class/rtc/rtcN sysfs attributes support read only access to some RTC attributes.

8.4 Source Code Structure

Table 8-1 lists the source files for MC34708 RTC driver that are available in the `<ltib_dir>/rpm/BUILD/linux/drivers/rtc` directory.

Table 8-1. MC9S08DZ60 RTC Driver Files

File	Description
rtc-mc13892.c	Implementation of the RTC driver, shares wtih the mc13892

8.5 Menu Configuration Options

The following Linux kernel configurations are provided for this module. To get to the MC34708 RTC configuration, use the command ./ltib -c when located in the `<ltib dir>`. In the screen, select **Configure Kernel**, exit, and a new screen appears.

- Device Drivers > Realtime Clock > Freescale MC34708 Real Time Clock.

Chapter 9
CPU Frequency Scaling (CPUFREQ) Driver

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 VDDGP is changed to the voltage value defined in `cpu_wp_auto[]` . This method can reduce power consumption (thus saving battery power), because the CPU uses less power as the clock speed is reduced.

9.1 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_wp_auto[]`, the CPUFREQ driver changes the CPU frequency to the nearest 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.

Refer to the API document for more information on the functions implemented in the driver (in the doxygen folder of the documentation package).

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 160 MHz) use this command:

```
echo 160000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

The frequency `160000` is in KHz, which is `160` MHz.

The maximum frequency can be checked 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
```

9.2 Source Code Structure

Table 9-1 shows the source files and headers available in the following directory:

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

Table 9-1. CPUFREQ Driver Files

File	Description
cpufreq.c	CPUFREQ functions

9.3 Menu Configuration Options

The following Linux kernel configuration is provided for this module:

- CONFIG_CPU__FREQ—In menuconfig, 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 (userspace)
  - Performance governor
  - Powersave governor
  - Userspace governor for userspace frequency scaling
  - Conservative CPU frequency governor
  - CPU frequency driver for i.MX CPUs

9.3.1 Board Configuration Options

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

Chapter 10
Dynamic Voltage Frequency Scaling (DVFS) Driver

The Dynamic Voltage Frequency Scaling (DVFS) device driver allows simple dynamic voltage frequency scaling. The frequency of the core (CPU) clock domain and the voltage of the core power domain can be changed on the fly with all modules continuing their normal operations. The voltage of the core power domain can be changed through the PMIC. The frequency of the core clock domain can be changed by switching temporarily to an alternate PLL clock, and then returning to the updated PLL, already locked at a specific frequency, or by merely changing the post dividers division factors.

10.1 Hardware Operation

The DVFS core module is a power management module. The purpose of the DVFS module is to detect the appropriate operation frequency for the IC. DVFS core is operated under control of the GPC (General Power Controller) block. The hardware DVFS core interrupt is served by GPC IRQ. The DVFS core domain performance update procedure includes both voltage and frequency changes in appropriate order by the GPC controller (hardware). For more information on the HW DVFS Core block refer to the DVFS chapter in the *Multimedia Applications Processor* documentation.

10.2 Software Operation

The DVFS device driver allows the frequency of the core clock domain and the voltage of the core power domain to be changed on the fly. The frequency of the core clock domain and the voltage of the core power domain are changed by switching between defined freq-voltage operating points. 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 using the clock command in RedBoot.

To Enable the DVFS core use this command:

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

To Disable The DVFS core use this command:

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

10.3 Source Code Structure

Table 10-1 lists the source files and headers available in the following directory:

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

Table 10-1. DVFS Driver Files

File	Description
dvfs_core.c	Linux DVFS functions

10.4 Menu Configuration Options

There are no menu configuration options for this driver. The DVFS core is included by default.

10.4.1 Board Configuration Options

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

Chapter 11
Software Based Peripheral Domain Frequency Scaling

The frequency of the clocks in the peripheral domain can be changed using the software based Bus Frequency Scaling driver. Enabling this driver can significantly lower the power numbers in the LP domain. Depending on the platform, the voltage of the peripheral domain can also be dropped using the on board PMIC.

11.1 Software based Bus Frequency Scaling

The SW will automatically lower the frequency of the various clocks in the peripheral domain based on which drivers are active (it is assumed that the drivers will use the clock API to enable/disable their clocks). Two setpoints are defined for the peripheral bus clock:

> AHB_HIGH_SET_POINT - The module requires the AHB clock to be at the highest frequency (133MHz).
>
> AHB_MED_SET_POINT - The module requires the AHB clock be above 66.5MHz.

The Bus Frequency Scaling driver will take into account the above two associations for the various clocks in the system before changing the peripheral clock.

To enable the SW based Bus Frequency Scaling (***not needed to enter LPAPM mode***) use this command:

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

To disable the SW based Bus Frequency Scaling use this command:

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

Based on which clocks are active, the system can be in any of the three modes specified below:

11.1.1 Low Power Audio Playback Mode (LPAPM)

When all the clocks that need either of the above two mentioned setpoints are disabled, the system can enter an ultra low power mode where the AHB clock and other main clocks in the LP domain are dropped down to 24MHz. On certain platforms and depending on the type of memory used, the DDR frequency is also dropped down to 24MHz. This mode is most commonly entered when the system is idle and the display is turned off. The implementation automatically detects when this mode can be entered and calls into the Bus Frequency driver to change the clocks (and voltages if it can be done) appropriately. On certain platforms, the entire SoC is clocked off the 24MHz oscillator and all PLLs are turned off to save more power.

If any driver that needs the higher AHB clock enables its clock, LPAPM mode will be exited. **Entry and exit from the LPAPM mode does not require the Bus Frequency Scaling driver to be enabled.**

11.1.2 Medium Frequency setpoint

In this mode the AHB and some of the LP domain clocks are divided down such that the AHB clock is above 66.5MHz. In this mode all drivers that require AHB_HIGH_SET_POINT are disabled. Depending on the platform, the voltage can also be dropped.

11.1.3 High Frequency setpoint

In this mode none of the frequencies on the peripheral domain are scaled since drivers that need the AHB_HIGH_SETPOINT are active.

11.2 Source Code Structure

Table 11-1 lists the source files and headers

Table 11-1. Bus Frequency Scaling Driver Files

File	directory	Description
bus_freq.c	arch/arm/mach-mx5	SW bus frequency driver functions

11.3 Menu Configuration Options

There is no option for the SW based Bus Frequency Scaling driver, it included by default.

11.3.1 Board Configuration Options

There are no board configuration options for the Linux Bus Frequency Scaling device driver.

Chapter 12
Low-level Power Management (PM) Driver

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

12.1 Hardware Operation

The i.MX5 supports four low power modes: RUN, WAIT, STOP, and LPSR (low power screen).

Table 12-1 lists the detailed clock information for the different low power modes.

Table 12-1. 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
LPSR	Disable	Disable	Off	On	On

For the detailed information about lower power modes, see the *MCIMX50 Multimedia Applications Processor Reference Manual* (MCIMX50RM).

12.2 Software Operation

The i.MX5 PM driver maps the low-power modes to the kernel power management states as listed below:

- Standby—maps to WAIT mode which offers minimal power saving, while providing a very low-latency transition back to a working system
- Mem (suspend to RAM)—maps to STOP mode which offers significant power saving as all blocks in the system are put into a low-power state, except for memory, which is placed in self-refresh mode to retain its contents
- System idle—maps to WAIT mode

The i.MX5 PM driver performs the following steps to enter and exit low power mode:

1. Enable the gpc_dvfs_clk
2. Allow the Coretex-A8 platform to issue a deep sleep mode request
3. If STOP mode:
   a) Program CCM CLPCR register to set low power control register.
   b) Request switching off ARM/NENO power when pdn_req is asserted.
   c) Request switching off embedded memory peripheral power when pdn_req is asserted.

   d) Program TZIC wakeup register to set wakeup interrupts
4. Call `cpu_do_idle` to execute WFI pending instructions for wait mode
5. If STOP mode, execute `cpu_do_suspend_workaround` in RAM. Change the drive strength of DDR SDCLK as “low” to minum the power leakage in SDCLK. Execute WFI pending instructions for stop mode
6. Generate a wakeup interrupt and exit low power mode. If STOP mode, restore DDR drive strength.
7. Disable gpc_dvfs_clk

12.3 Source Code Structure

Table 12-2 shows the PM driver source files. These files are available in

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/`

Table 12-2. PM Driver Files

File	Description
pm.c	Supports suspend operation
system.c	Supports low-power modes
wfi.S	Assemble file for `cpu_cortexa8_do_idle`
suspend.S	Assemble file for `cpu_do_suspend_workaround`

12.4 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_PM—Build support for power management. In menuconfig, this option is available under

  Power management options > Power Management support

  By default, this option is Y.
- CONFIG_SUSPEND—Build support for suspend. In menuconfig, this option is available under

  Power management options > Suspend to RAM and standby

12.5 Programming Interface

The `mxc_cpu_ip_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.

Chapter 13
ELCDIF Frame Buffer Driver

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 with the one for CLAA WVGA LCD driver.

13.1 Hardware Operation

The frame buffer driver uses the LCDIF API to interact with the hardware.

13.2 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 seeked and `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, by 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 what IOCTLs exist and which data structures they use.

The frame buffer driver implementation for i.MX50 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.

13.3 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_FB_MXC_CLAA_WVGA_SYNC_PANEL [=Y|N|M]
  Configuration option to compile support for the CLAA WVGA(800x480) LCD panel into the kernel (Supported on ARM2 board).
- CONFIG_FB_MXC_SEIKO_WVGA_SYNC_PANEL [=Y|N|M]
  Configuration option to compile support for the SEIKO WVGA(800x480) LCD panel into the kernel (Supported on RDP board).
- CONFIG_FB_MXC_ELCDIF_FB [=Y|N|M]
  Configuration option to compile support for the MXC ELCDIF frame buffer driver into the kernel.

13.4 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 13-1.

Table 13-1. 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 14
Electrophoretic Display Controller (EPDC) Frame Buffer Driver

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 16 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.

14.1 Hardware Operation

The detailed hardware operation of the EPDC is discussed in the *MCIMX50 Multimedia Applications Processor Reference Manual (MCIMX50RM)*.

14.2 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 also through the custom E Ink API extensions. Additionally, panel intialization and framebuffer formats are discussed.

14.2.1 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 more detail in the EPDC Frame Buffer Driver Extensions section.

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.

14.2.2 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 on page 14-6.

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 on page 14-6.

14.2.3 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`.

```
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 *MCIMX50RM* 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.

14.2.3.1 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.

14.2.4 EPDC Panel Initialization

The framebuffer driver will not typically (** - see below) 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.

** 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.

14.2.5 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);
```

14.3 Source Code Structure

Table 13-2 lists the source files associated with the EPDC driver. These files are available in the following directory:

`drivers/video/mxc`

Table 13-3 lists the global header files associated with the EPDC driver. These files are available in the following directory:

`include/linux/`

Table 14-2. 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

14.4 Menu Configuration Options

The following Linux kernel configuration options are provided for the EPDC 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_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—This option enables support for auto-update mode, which provides automated EPD updates through the deferred IO 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—This 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—This 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—This configuration option 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

14.5 Programming Interface

14.5.1 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, 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_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:

int32_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:

__u32 *update_marker*

User-defined value used to identify a particular update (passed as a parameter in MXCFB_SEND_UPDATE IOCTL call). The marker value should be re-used here to wait for the update to complete.

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.

14.5.2 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 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 */
};
```

Chapter 15
Data Co-Processor (DCP) Driver

The Data Co-Processor (DCP) cryptography driver is used to accelerate cryptographic operations in the kernel space and user-space (Linux Crypto API).

15.1 Hardware Operation

The DCP provides support for general encryption functions typically used for security algorithms. The supported modes are:

- Advanced Encryption Standard (AES)
- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- SHA-1
- SHA-256

The driver uses DMA to process the data in place.

15.2 Software Operation

The software implementation of the DCP driver conforms with the Linux Crypto driver model. It registers a number of ciphers and block ciphers. Refer to the documentation located in the folder `Documentation/crypto/` (kernel source tree) for full information about the Linux Crypto API.

The following ciphers are registered in the DCP driver module:

- AES

The following block ciphers are registered in the DCP driver module:

- AES (ECB)
- AES (CBC)

The following hashing algorithms are registered in the DCP driver module:

- SHA-1
- SHA-256

In addition, the DCP can perform 128-bit AES crypto using the OTP KEY0 which is not accessible by software, and therefore not usable through the Crypto APIs. The driver permits a user application to use AES 128-bit/ECB mode but only supports encrypting/decrypting a single 128-bit block. The ROM uses the OTP key during boot. Therefore, it is possible to verify that a bootstream is valid before committing it to Flash. While the bootstream uses AES/CBC mode, it is far simpler to use the ECB mode. The user space access is performed by means of a miscellaneous device character file (under /proc/misc: dcpboot), and two IOCTLs: DBS_ENC for encryption and DBS_DEC for decryption.

Typical usage scenarios for encrypting/decrypting using the OTP key is as follows:

```
uint8_t mac[16];
int r, fd;

fd = open(<character-device-file>, O_RDWR);
/* check fd for successful open call */
/* load vector to mac */
r = ioctl(fd, DBS_ENC, mac); /* encrypt */
/* check r for failure (!=0) */
r = ioctl(fd, DBS_DEC, mac); /* decrypt */
/* check r for failure (!=0) */
close(fd);
```

15.3 Source Code Structure

The DCP cryptography module is implemented in the following files:

```
drivers/crypto/dcp.c
drivers/crypto/dcp.h
drivers/crypto/dcp-bootstream-ioctl.h
```

15.4 Menu Configuration Options

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

- CONFIG_CRYPTO_DEV_DCP [=M|Y]

  Configuration option for the DCP cryptography driver, which is dependent on ARCH_MX50 and automatically selects CRYPTO_ALGAPI and CRYPTO_BLKCIPHER.

  In menuconfig, this option is available under:
  Cryptographic API > Hardware crypto devices > Support for the DCP engine

15.5 Programming Interface

This driver is integrated into the Linux Crypto API.

Chapter 16
Random Number Generator (RNG) Driver

The Random Number Generator (RNG) driver driver is used to generate pseudo-random data in the RNG hardware for the user via the hwrng device framework in Linux. The pseudo-random data is generated by the hardware in a NIST compliant fashion and is intended for use in cryptographic algorithms.

16.1 Hardware Operation

Please refer to the SOC's Reference Manual.

16.2 Software Operation

The RNG driver registers as a hwrng device with the hw_random character driver in Linux. The user can access the random data generated by the RNG simply via a read of the /dev/hwrng node for as many bytes as necessary.

16.3 Source Code Structure

While the RNG driver file is named for the RNGC hardware module, it works equally well for the RNGB module.

The RNG driver is implemented in the following file:

```
drivers/char/hw_random/fsl-rngc.c
```

16.4 Menu Configuration Options

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

- CONFIG_HW_RANDOM_FSL_RNGC [=M|Y]

  Configuration option for the RNG driver, which is dependent on CONFIG_HW_RANDOM, which is set by default through imx5_defconfig, ARCH_HAS_RNGC, which is set by ARCH_MX50. This driver conflicts with the RNG driver found in the menuconfig under the following path: Device Drivers > MXC support drivers > MXC security drivers > MXC RNG driver. Therefore, the config for this driver is also dependent on NOT setting CONFIG_MXC_SECURITY_RNG.

  In menuconfig, this option is available under:
  Device Drivers > Character devices > Freescale RNGC Random Number Generator.

16.5 Unit Test

1. Boot the board & login as root.

2. Run the RNG test of dumping out 100 bytes of data:

```
root:~# dd if =/dev/hwrng bs=1 count=100 | hexdump
```

3. Run the same test again to ensure that the same data is not generated again.

Chapter 17
Video for Linux Two (V4L2) Driver

The V4L2 driver is designed and implemented to comply with version 2.0 of the Video4Linux API (V4L2). The software implements a platform driver for a V4L2 output and output overlay device. The implementation interacts with the PxP DMA-ENGINE driver. The LCD driver is organized in a flexible and extensible manner. It interacts with the frame buffer driver to accomplish frame buffer overlay of the video stream and output to the display. See Chapter 13, “ELCDIF Frame Buffer Driver,” for more details on the frame buffer driver.

17.1 Hardware Operation

Please refer to the SOC’s Reference Manual.

17.2 Software Operation

A V4L2 driver utilizes the V4L2 driver framework to provide functionality by a standard character driver model. The V4L2 API Specification (Revision 0.24) provides complete details of the API exported to user space applications. The following V4L2 features are supported by the driver:

- RGB555, RGB565, RGB24, YUV420 (planar), and YUV422P (planar) input formats
- Programmable input pixel format and size
- Mmap streaming input buffers
- Direct PxP output to the display
- Overlay of the frame buffer on the PxP input stream
- Color keying and alpha blending of the overlay
- Horizontal and vertical flipping of the PxP output
- 90°, 180°, and 270° rotation of the PxP output
- Programmable scaling of YUV420 and YUV422P input formats
- Programmable default background color
- Selection of YCbCr or YUV color space

These features are supported using custom APIs:

- Output to mmap user buffer

The supported V4L2 IOCTLs include the following:

- VIDIOC_QUERYCAP
- VIDIOC_REQBUFS
- VIDIOC_QUERYBUF

- VIDIOC_QBUF
- VIDIOC_DQBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF
- VIDIOC_ENUMOUTPUT
- VIDIOC_G_OUTPUT
- VIDIOC_S_OUTPUT
- VIDIOC_ENUM_FMT
- VIDIOC_TRY_FMT
- VIDIOC_G_FMT
- VIDIOC_S_FMT
- VIDIOC_G_FBUF
- VIDIOC_S_FBUF
- VIDIOC_G_CROP
- VIDIOC_S_CROP
- VIDIOC_QUERYCTRL
- VIDIOC_G_CTRL
- VIDIOC_S_CTRL

The following V4L2 standard controls are implemented by the driver:

- V4L2_CID_HFLIP
- V4L2_CID_VFLIP

The following V4L2 custom controls have been added to the driver:

- V4L2_CID_PRIVATE_BASE—Rotation (0°, 90°, 180°, or 270°)
- V4L2_CID_PRIVATE_BASE + 1—Background Color
- V4L2_CID_PRIVATE_BASE + 2—Set S0 Chromakey
- V4L2_CID_PRIVATE_BASE + 3—Color space (0 - YCbCr, 1 - YUV)

17.3 Menu Configuration Options

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

- VIDEO_MXC_PXP_V4L2 [=M|Y]

  This is a video4linux driver for the Freescale PxP(Pixel Pipeline). This module supports output overlay of the MXC framebuffer on a video stream.

  In menuconfig, this option is available under:

  `Device Drivers -> Multimedia Device -> Video capture adapter -> MXC PxP V4L2 driver`

17.4 Source Code Structure

The V4L2 driver source code is located in `drivers/media/video/mxc/output/mxc_pxp_v4l2.c` and `drivers/media/video/mxc/output/mxc_pxp_v4l2.h`.

Chapter 18
Pixel Pipeline (PxP) DMA-ENGINE Driver

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.

18.1 Hardware Operation

The PxP driver uses PxP registers to interact with the hardware. For detailed hardware operation, please refer to the i.MX50 reference manual.

18.2 Software Operation

18.2.1 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:

ddevice_alloc_chan_resources/ allocate resources and descriptors */*

device_free_chan_resources/ release DMA channel's resources */*

device_is_tx_complete/ 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/ terminate all pending operations */*

The first four functions are used by the DMA Engine Framework, the last two are used by the DMA slave (DMA client). Notably, *dma_async_issue_pending* (which calls *device_issue_pending internally*) 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.

18.2.2 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.

18.2.3 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.

18.2.4 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.

18.2.5 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.

18.3 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

18.4 Source Code Structure

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

drivers/dma/pxp/pxp_dma.c

- PxP DMA driver source file. Implements the DMA Engine API.

drivers/dma/pxp/pxp_device.c

- PxP interface driver for the user-space library to access

drivers/dma/pxp/Makefile

- Makefile to build pxp_dma

include/linux/pxp_dma.h

- Public include file for MXC pxp_dma driver.

drivers/dma/Makefile

- Added MXC pxp_dma to build

drivers/dma/Kconfig

- Added entry for MXC pxp_dma

Chapter 19
Graphics Processing Unit (GPU)

The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D graphics applications. The GPU2D (2D graphics processing unit) is based on the AMD Z160 core, which is an embedded 2D and vector graphics accelerator targeting OpenVG 1.1 graphics API and feature set as well as common 2D acceleration. The GPU driver can be divided into mainly two parts, GPU kernel module and user space libraries. The GPU kernel module is open source, while the user space libraries are delivered as binaries.

19.1 Driver Features

The GPU driver provides following software support with hardware acceleration:

1. EGL (EGL™ is an interface between Khronos rendering APIs such as OpenGL ES or OpenVG and the underlying native platform window system) 1.3 API defined by Khronos Group
2. OpenVG (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.

19.2 Hardware Operation

Refer to the GPU chapter in the *MCIMX50 Multimedia Applications Processor Reference Manual* (MCIMX50RM) for detailed hardware operation and programming information.

19.3 Software Operation

The GPU driver is divided into two parts. The GPU kernel module is the base of the entire stack, which provides essential hardware access, device management, memory management, command stream management, context management and power management. The user space libraries implements the stack logic and provides following APIs to the upper layer applications.

- EGL 1.3 API
- OpenVG 1.1 API

19.4 Source Code and Binaries

The source code of the GPU kernel module is provided in drivers/mxc/amd-gpu directory of the 2.6.31 kernel source tree. Table 19-1 lists the modules and libraries associated with GPU.

Table 19-1. GPU Related File List

File	Description
lib2dz160.so libgsl.so libEGL.so libOpenVG.so	GPU related libraries that are part of the Linux BSP filesystem. These libraries are located in /usr/lib/
2dblt_perf	c2d performance test located in /usr/bin/
c2d_sanity_app_z160	c2d sanity test located in /usr/bin/

19.5 GPU Driver API Reference

Refer to the following web sites for detailed specifications:

- EGL 1.3 API: http://www.khronos.org/egl/ for detailed specifications
- OpenVG 1.1 API: http://www.khronos.org/openvg/

19.6 Menu Configuration Options

The same GPU driver package is used for i.MX50 as i.MX51 and i.MX53.

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

- CONFIG_MXC_AMD_GPU - Configuration option for GPU driver. In the menuconfig this option is available under Device Drivers > MXC support drivers > MXC GPU support > MXC GPU support.

The GPU kernel module is configured as loadable module by default.

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

- Package list > amd-gpu-bin-mx51

  This package provides proprietary binary libraries, and test applications based on framebuffer
- Package list > amd-gpu-x11-bin-mx51

  This package provides proprietary binary libraries, and test applications based on X-Window

Chapter 20
X Windows Acceleration

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

20.1 Hardware Operation

X Windows acceleration utilizes the 2D GPU which is discussed in the Chapter 19, “Graphics Processing Unit (GPU). Acceleration is also dependent on the frame buffer memory.

20.2 Software Operation

X Windows acceleration is supported for X.Org X Server version 1.6.4.

The following list summarizes the types of operations that are accelerated for X Windows. All operations involve frame buffer memory which may be onscreen or offscreen.:

- Solid fill of a rectangle
- 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 list includes additional features supported as part of the X Windows acceleration:

- Allocation of X pixmaps directly in frame buffer memory
- EGL swap buffers where EGL window surface is an X window
- X window can be composited into an X pixmap which can be used directly as any EGL surface

20.2.1 X Windows Acceleration Architecture

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

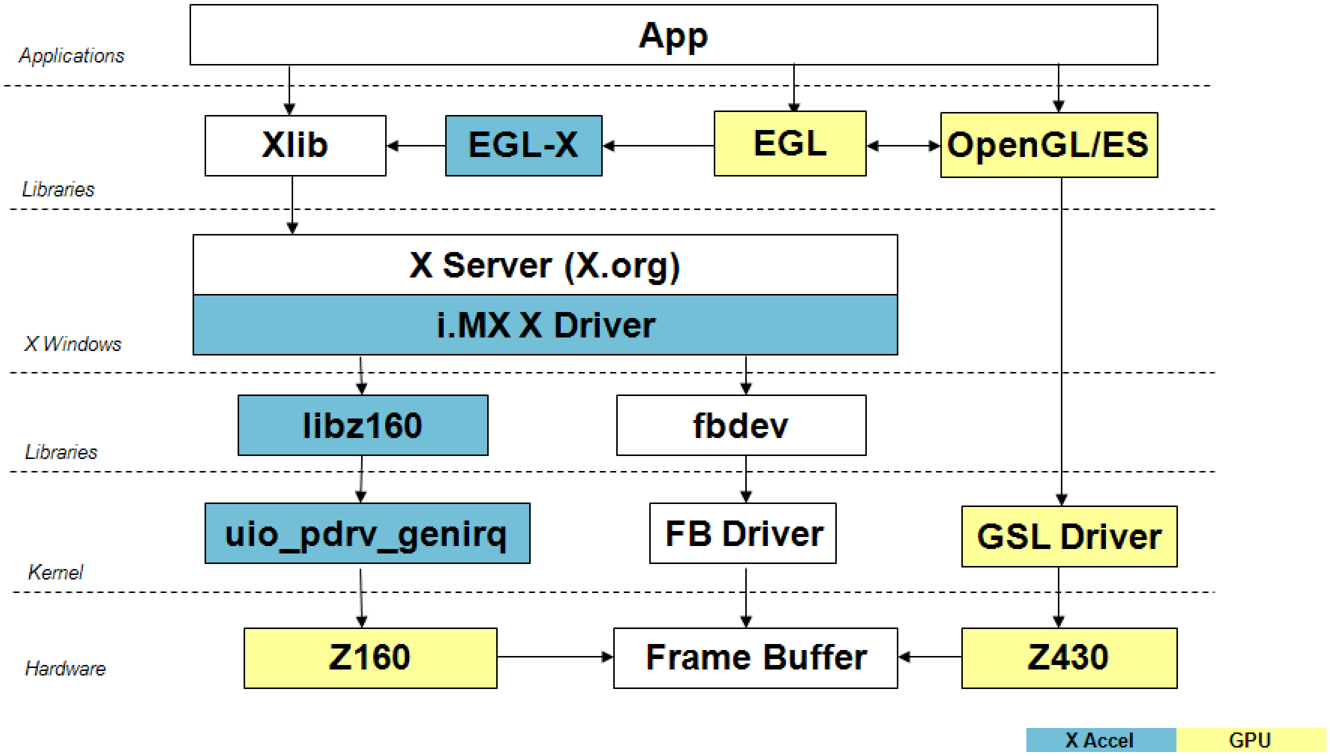

Figure 20-1. X Window Acceleration Block Diagram

The components shown in yellow are those provided as part of the 2D/3D GPU driver support which includes OpenGL/ES and EGL. The components shown in white are the standard components in the X Windows system without acceleration. The components shown in blue are those added to support X Windows acceleration and briefly described here.

The **i.MX X Driver** lbrary module (`imx-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 Z160 2D GPU core. The entire linearly contiguous frame buffer memory in `/dev/fb0` is used for allocating pixmaps for X both onscreen and offscreen. The driver provides supports a custom X extension which allows X clients to query the GPU address of any X pixmap stored in frame buffer memory.

The **libz160** library module (`libz160.so`) contains the register level programming interface to the Z160 GPU module. This includes the storing of register programming commands into packets which can be streamed to the device. The functions in this library are called by the i.MX driver code.

The **uio_pdrv_genirq** loadable kernel module (`uio_pdrv_genirq.ko`*)* is based on the user space I/O driver which provides generic intterrupt handlinig. The base driver handles interrupts received when the 2D GPU is idle. The driver is extended to provide user-space memory mapping of the Z160 registers as well as the memory to use for command stream buffering. The code in this kernel module is indirectly called when the functions in the libz160 library module access the `/dev/uio` device to program the Z160. **Note that when this kernel module is loaded, it is not possible to run OpenVG applications.**

The **EGL-X** library module (`libEGL.so`) contains the X Windows implementation of the low level EGL plarform-specific support functions. This allows X window and X pixmap objects to be used as EGL window and pixmap surfaces. The EGL-X library uses Xlib function calls in its implementation along with the i.MX driver module's X extension for querying the GPU address of X pixmaps stored in frame buffer memory.

20.2.2 i.MX X Driver Details

The i.MX X Driver, referred to as imx-drv, implements the EXA interface of the X server in providing acceleration. The following list mentions details particular to this implementation:

- The EXA solid fill operation is accelerated, except for rectangles containing less than 150 pixels in which case fallback is to software rendering.
- The EXA copy operation is accelerated, except for rectangles containing less than 150 pixels in which case fallback is to software rendering.
- For EXA solid fill and copy operations, only solid plane masks and only `GXcopy` raseter-op operations are accelerated.
- EXA composite allows for many options and combinations of source/mask/target for rendering. Most of the (commonly used) EXA composite operations are accelerated.

  The following types of EXA composite operations are accelerated;

  - Simple source composite with target
  - Simple source-in-mask composite with target
  - Constant source composite with target
  - Constant source and buffer mask composite with target
  - Operations for rectangles containing at least 150 pixels
  - Only these blending functions: SOURCE, OVER, IN, IN-REVERSE, OUT-REVERSE, and ADD (some of these are needed to support component-alpha blending which is accelerate)

  In general, the following types of (less commonly used) EXA composite operations are **not** accelerated:

  - Transformed sources/maskes
  - Sources intended to be repeating patterns
  - Masks with repeating patterns
- All of the memory allocated for `/dev/fb0` is made available to EXA's linear offscreen memory manager. The portion of this memory beyond the screen memory is available for allocation of X pixmaps. Once an X pixmap is allocated in the frame buffer memory, it is never migrated back to system memory. The amount of memory allocated to `/dev/fb0` needs to be larger than the amount needed for the screen; otherwise, no operations can be accelerated because X pixmaps must be in frame buffer memory (as a source and target) in order to be accelerated. This amount of memory needs to be several MB, but that depends on the number of X windows and pixmaps used, possible usage of X pixmaps as texture, and whether X windows are using the XComposite extension.
- An 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 frame buffer memory.
- The buffer pitch alignment for the Z430 is 32 bytes while the buffer pitch alignment for the Z160 is 4 bytes. Because X pixmaps can be allocated from the frame buffer memory and these X pixmaps

could be used in EGL for OpenGL/ES drawing operations (using the Z430), the pitch alignment requirement given to EXA’s offscreen memory manager is the conservative value of 32 bytes.

- Attempts are made to disable the Z160 sub-module clocks whenever the 2D GPU is not being used for a period of time. The idle time period is specified in milliseconds where the default idle period before these clocks are disabled is 1000 milliseconds. This time period value can be modified by adding a “`GPUIdleTimeout`” “`Option`” in the xorg.conf for this imx-drv driver. A time period value of 0 causes the clocks to never be disabled.

20.2.3 EGL-X Details

The EGL-X library implements the low level EGL interface when used in the X Windows system. The following list mentions details particular to this implementation:

- The `eglDisplay` native display type is “`Display*`” in X Windows.
- The `eglWindowSurface` native window surface type is “`Window`” in X Windows.
- The `eglPixmapSurface` native pixmap surface type is “`Pixmap`” in X Windows.
- When an `eglWindowSurface` is created, the back buffers used for double-buffering can have different representations from the window surface (based on the selected `eglConfig`). An attempt is made to create each back buffer using the representation which provides the most efficient blit of the back buffer contents to the window surface when `eglSwapBuffers` is called. Each back buffer is allocated separately using the following approach:
  - — Create an X pixmap of the necessary size. Use the X extension for the imx-drv X acceleration driver module to query the physical frame buffer address for this X pixmap if was allocated in the offscreen frame buffer memory.
  - — If the X pixmap is in the offscreen frame buffer memory and ...
    - – the pixel format of the back buffer matches that of the window, then the `XCopyArea` function is used to blit the back buffer to the window.
    - – the pixel format of the back buffer does **not** match that of the window, then the `XRenderComposite` function is used to blit (and convert) the back buffer to the window.
  - — If the X pixmap is **not** in the offscreen frame buffer memory, then a back buffer is allocated from the GSL memory pool and mapped to an `XImage`.
    - – If the pixel format of the back buffer matches that of the window, then the `XPutImage` function is used to blit the back buffer to the window.
    - – If the pixel format of the back buffer does **not** match that of the window, then the `XPutImage` function is used to first blit the back buffer to the X pixmap (to get image transferred to X server without format convrsion) and then the `XRenderComposite` function is used to blit (and convert) the X pixmap to the window.

20.2.4 Setup X Windows Acceleration

- Verify that the following packages are available and installed:

```
kernel_<kernel-version>-imx_<bsp-version>_armel.deb
```

This package contains the `uio_pdrv_genirq.ko` kernel module and installs it into the folder with all the other loadable kernel modules.

`udev-fsl-rules_<bsp-version>_armel.deb`

Install the `udev-fsl-rules_<bsp-version>_armel.deb` package. This package sets up any necessary permissions for `/dev/uio`.

`libz160-bin_<bsp-version>_armel.deb`

This package contains the `libz160.so` library module and installs it in the `/usr/lib` folder.

`amd-gpu-x11-bin-mx51_<bsp-version>_armel.deb`

This package contains the `libEGL.so` library module and installs it in the `/usr/lib` folder.

`xserver-xorg-video-imx_<bsp-version>_armel.deb`

This package contains the `imx-dev.so` driver module for X acceleration and installs it in the folder with all the other X.org driver modules.

- Verify that the file `/etc/modules` contains the following entries (in the order listed):

```
uio_pdrv_genirq
gpu
```

- Verify that the file `/etc/X11/xorg.conf` contains one “`Device`” section as follows

```
Section "Device"
        Identifier      "i.MX Accelerated Framebuffer Device"
        Driver          "imx"
        Option          "ShadowFB"                  "false"
        Option          "MigrationHeuristic"        "smart"
EndSection
```

- Add "`x_mem=<size>`" to the kernel boot command "`video=`" parameter string. “`<size>`″ should be approximately 15MB. For example:

```
video=mxcepdcfb:E60,bpp=16,x_mem=15M
```

There are a few ways to verify that X Windows acceleration is indeed operating given that the above steps have been performed.

1. Run “`lsmod`” command from any terminal. It should list the `gpu` and `uio_pdrv_genirq` kernel modules, in that order.
2. Edit the file `/var/log/Xorg.0.log` and confirm that the following lines are present:

```
(II) EXA(0): Using custom EXA
(II) IMX(0): IMX EXA acceleration setup successful
```

3. In the same `/var/log/Xorg.0.log` file, search for a line similar to the following:

```
(II) EXA(0): Offscreen pixmap area of 15062K bytes
```

This would indicate the number of bytes available for X pixmaps that can be allocated in the off-screen frame buffer memory. In this example, there is almost 15MB of available memory.

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

This section describes the ASoC driver architecture and implementation. The ASoC architecture is imported to provide a better solution for ALSA kernel drivers. ASoC aims to divide the ALSA kernel driver into machine, platform (CPU), and audio codec components. Any modifications to one component do not impact another components. The machine layer registers the platform and the audio codec device, and sets up the connection between the platform and the audio codec according to the link interface, which is supported both by the platform and the audio codec. More detailed information about ASoC can be found at http://www.alsa-project.org/main/index.php/ASoC.

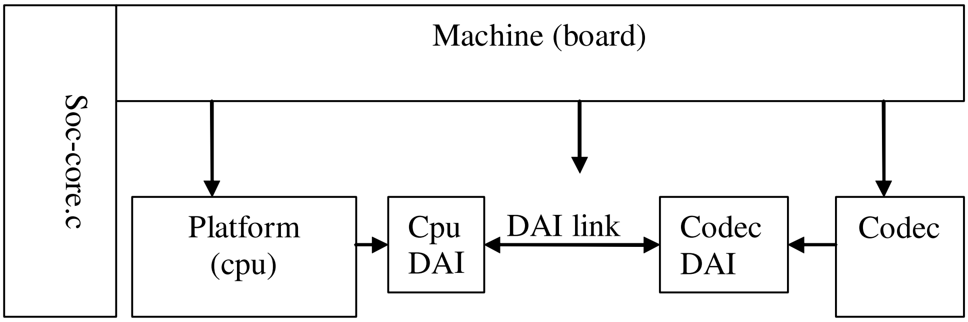

Figure 21-1. ALSA SoC Software Architecture

The ALSA SoC driver has the following components as shown in Figure 21-1:

- 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.

21.1 SoC Sound Card

Currently, the stereo codec (`sgtl5000`), 5.1 codec (`wm8580`), 4-channel ADC codec (ak5702), 7.1 codec(cs42888), built-in ADC/DAC codec, and Bluetooth codec drivers are implemented using SoC architecture. The five sound card drivers are built in independently. The stereo sound card supports stereo playback and mono capture. The 5.1 sound card supports up to six channels of audio playback. The 4-channel sound card supports up to four channels of audio record. The Bluetooth sound card supports

Bluetooth PCM playback and record with Bluetooth devices. The built-in ADC/DAC codec supports stereo playback and record.

NOTE

Only the Stereo Codec is supported on the i.MX50 platform.

21.1.1 Stereo Codec Features

The stereo codec supports the following features:

- Sample rates for playback and capture are 32 KHz, 44.1 KHz, 48 KHz, and 96 KHz
- Channels:
  - Playback: supports two channels. (stereo)
  - Capture: supports two channels. (Only one channel has valid voice data due to hardware connection)
- 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

21.1.2 Sound Card Information

The registered sound card information can be listed as follows using the commands `aplay -l` and `arecord -l`.

```
root@freescale /$ aplay -l
**** List of PLAYBACK Hardware Devices ****
      card 0: imx3stack [imx-3stack], device 0: SGTL5000 SGTL5000-PCM-0 []
       Subdevices: 1/1
       Subdevice #0: subdevice #0
root@freescale /$ arecord -l
**** List of CAPTURE Hardware Devices ****
      card 0: imx3stack [imx-3stack], device 0: SGTL5000 SGTL5000-PCM-0 []
       Subdevices: 1/1
      Subdevice #0: subdevice #0
```

21.2 ASoC Driver Source Architecture

As shown in Figure 21-1, `imx-pcm.c` is shared by the stereo ALSA SoC driver, the 5.1 ALSA SoC driver and the Bluetooth 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. `sgtl5000.c` registers the stereo codec and hifi DAI drivers. The direct hardware operations on the stereo codec are in `sgtl5000.c`. `imx-3stack-sgtl5000.c` is the machine layer code which creates the driver device and registers the stereo sound card.

Figure 21-2 shows the ALSA SoC source file relationship.

Figure 21-2. ALSA Soc Source FIle Relationship

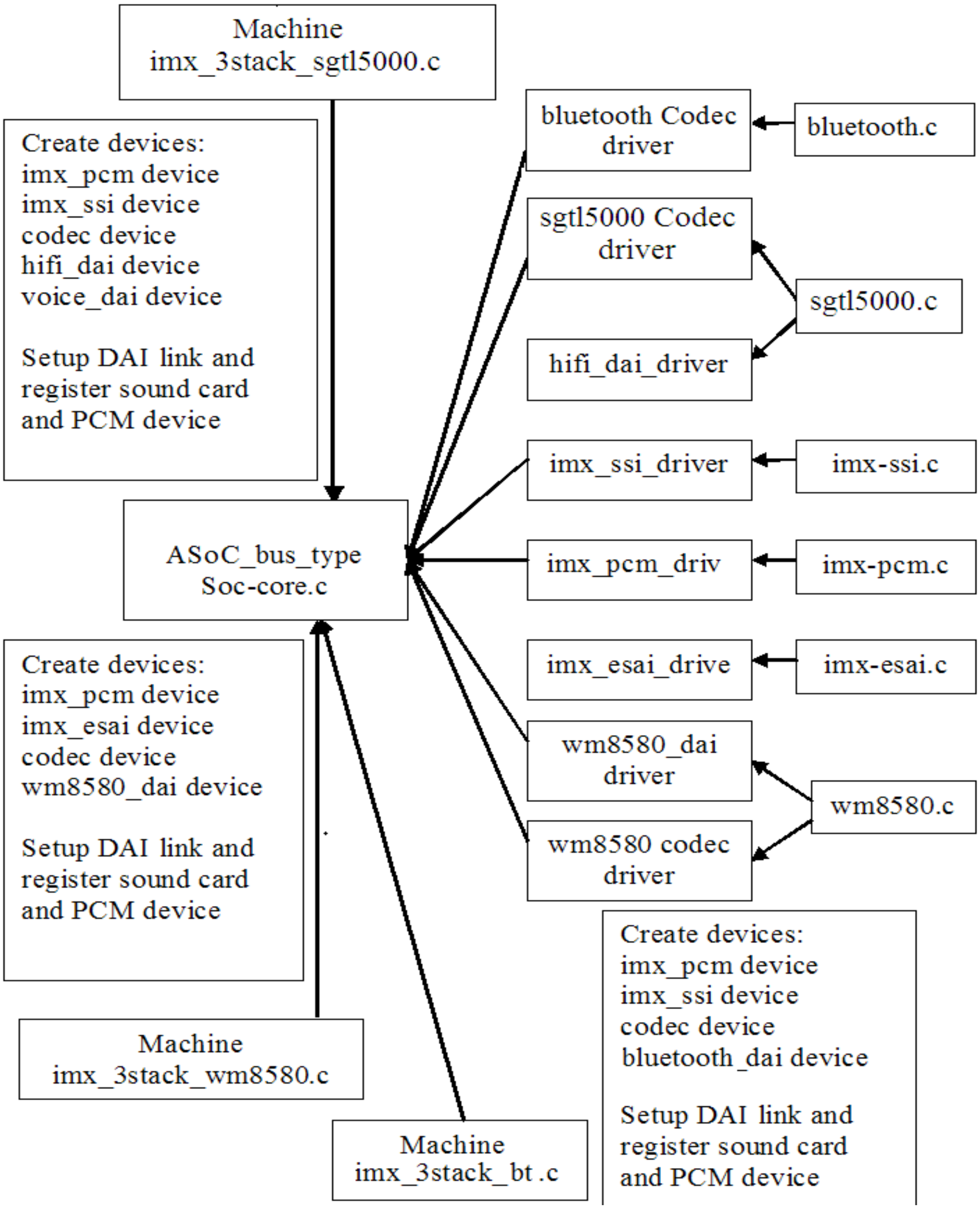

Table 21-1 shows the stereo codec SoC driver source files. These files are under the `<ltib_dir>/rpm/BUILD/linux/sound/soc` directory.

Table 21-1. Stereo Codec SoC Driver Files

File	Description
imx/imx-3stack-sgtl5000.c	Machine layer for stereo codec ALSA SoC
imx/imx-pcm.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
imx/imx-ac97.c	AC97 driver for i.MX chips
codecs/sgtl5000.c	Codec layer for stereo codec ALSA SoC
codecs/sgtl5000.h	Header file for stereo codec driver

21.3 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>`. Select **Configure the Kernel** on the screen displayed and exit. When the next screen appears, select the following options to enable this module:

- SoC Audio support for i.MX SGTL5000. In menuconfig, this 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
- CONFIG_SND_MXC_SOC_IRAM: This config is used to allow audio DMA playback buffers in IRAM. In menuconfig, this option is available under

  Device drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > Locate Audio DMA playback buffers in IRAM

21.4 Hardware Operation

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

21.4.1 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 SGTL5000 ASoC codec driver exports the audio record/playback/mixer APIs according to the ASoC architecture. The ALSA related audio function and the FM loopback function cannot be performed simultaneously.

The codec driver is generic and hardware independent code that configures the codec to provide audio capture and playback. It does not contains 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 SGTL5000 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`. The `io_probe` routine initializes the codec hardware to the desired state.

Headphone insertion/removal can be detected through a MCU interrupt signal. The driver reports the event to user space through sysfs.

21.5 Software Operation

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

21.5.1 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, preallocates 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

21.5.2 Device Open

The ALSA driver:

- 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.

Chapter 22
NAND Flash Driver

The NAND Flash Memory Technology Devices (MTD) driver is used in the Generic-Purpose Media Interface (GPMI) controller on the i.MX50. Only the hardware specific layer has to be implemented for the NAND MTD driver to operate. The rest of the functionality such as Flash read/write/erase is automatically handled by the generic layer provided by the Linux MTD subsystem for NAND devices.

22.1 Hardware Operation

NAND Flash is a nonvolatile storage device used for embedded systems. It does not support random accesses of memory as in the case of RAM or NOR Flash. Reading or writing to NAND Flash must be done through the GPMI. NAND Flash is a sequential access device appropriate for mass storage applications. Code stored on NAND Flash can not be executed from there. Code must be loaded into RAM memory and executed from there. The i.MX50 contains a hardware error-correcting block.

22.2 Software Operation

MTDs in Linux cover all memory devices such as RAM, ROM, and different kinds of NOR/NAND Flashes. The MTD subsystem provides uniform access to all such devices. Above the MTD devices there could be either MTD block device emulation with a Flash file system (JFFS2) or a UBI layer. The UBI layer in turn, can have either UBIFS above the volumes or a Flash Translation Layer (FTL) with a regular file system (FAT, Ext2/3) above it. The hardware specific driver interfaces with the GPMI module on i.MX50. It implements the lowest level operations such as read, write and erase. If enabled, it also provides information about partitions on the NAND device—this information has to be provided by platform code.

The NAND driver is the point where read/write errors can be recovered, if possible. Hardware error correction is performed by ECC8 or BCH blocks and is driven by NAND drivers code.

Detailed information about NAND driver interfaces can be found at http://www.linux-mtd.infradead.org

22.2.1 Basic Operations: Read/Write

The NAND driver exports the following callbacks:

- `mil_ecc_read_page` (with ECC)
- `mil_ecc_write_page` (with ECC)
- `mil_read_byte` (without ECC)
- `mil_read_buf` (without ECC)
- `mil_write_buf` (without ECC)
- `mil_ecc_read_oob` (with ECC)

- `mil_ecc_write_oob` (with ECC)

These functions read the requested amount of data, with or without error correction. In the case of read, the `mil_incoming_buffer_dma_begin` function is called, which creates the DMA chain, submits it to execute, and waits for completion. The write case is a bit more complex: the data to be written is mapped and flushed out by calling `mil_incoming_buffer_dma_begin` before processing the command `NAND_CMD_PAGEPROG`.

22.2.2 Error Correction

When reading or writing data to Flash, some bits can be flipped. This is normal behavior, and NAND drivers utilize various error correcting schemes to correct this. It could be resolved with software or hardware error correction. The GPMI driver uses only a hardware correction scheme with the help of an hardware accelerator-BCH.

For BCH, the page laylout of 2K page is (2k + 64), the page layout of 4K page is (4k + 218).

22.2.3 Boot Control Block Management

During startup, the NAND driver scans the first block for the presence of a NAND Control Block (NCB). Its presence is detected by magic signatures. When a signature is found, the boot block candidate is checked for errors using Hamming code. If errors are found, they are fixed, if possible. If the NCB is found, it is parsed to retrieve timings for the NAND chip.

All boot control blocks are created when formatting the medium using the user space kobs application.

22.2.4 Bad Block Handling

When the driver begins, by default, it builds the bad block table. It is possible to determine if a block is bad, dynamically, but to improve performance it is done at boot time. The badness of the erase block is determined by checking a pattern in the beginning of the spare area on each page of the block. However, if the chip uses hardware error correction, the bad marks falls into the ECC bytes area. Therefore, if hardware error correction is used, the bad block mark should be moved. The driver decides if bad block marks should be moved if there is no NAND control block. Then, to prevent another move of bad block marks, the driver writes the default NCB to the Flash.

The following functions that deal with bad block handling are grouped together in the `gpmi-nfc-mil.c` file:

- `mil_block_bad`
- `mil_scan_bbt`
-

22.2.5 Special NAND supporting

The i.MX50 also supports the TOGGLE NAND and ONFI NAND.

Both the TOGGLE NAND and ONFI NAND support the DDR mode in transmission which can transfer data in double speed of the normal nand.

22.3 Source Code Structure

The NAND driver is located in the `drivers/mtd/nand/gpmi-nfc` directory. The following files are included in the NAND driver:

- `gpmi-nfc-main.c`
- `gpmi-nfc-mil.c`
- `gpmi-nfc-hal-common.c`
- `gpmi-nfc-hal-v0.c`
- `gpmi-nfc-hal-v1.c`
- `gpmi-nfc-hal-v2.c`
- `gpmi-nfc-event-reporting.c`
- `gpmi-nfc-rom-v0.c`
- `gpmi-nfc-rom-v1.c`
- `gpmi-nfc-rom-common.c`
- `gpmi-nfc.h`
- `gpmi-nfc-gpmi-regs-v0.h`
- `gpmi-nfc-gpmi-regs-v2.h`
- `gpmi-nfc-gpmi-regs-v3.h`
- `gpmi-nfc-bch-regs-v0.h`
- `gpmi-nfc-bch-regs-v1.h`
- `gpmi-nfc-bch-regs-v2.h`

22.4 Menu Configuration Options

To enable the NAND driver, the following options must be set:

- CONFIG_MTD_NAND_GPMI_NFC = [Y | M]

In addition, these MTD options must be enabled:

- CONFIG_MTD_NAND = [y | m]
- CONFIG_MTD = y
- CONFIG_MTD_PARTITIONS = y
- CONFIG_MTD_CHAR = y
- CONFIG_MTD_BLOCK = y

In addition, these UBI options must be enabled:

- CONFIG_MTD_UBI=y
- CONFIG_MTD_UBI_WL_THRESHOLD=4096
- CONFIG_MTD_UBI_BEB_RESERVE=1
- CONFIG_UBIFS_FS=y
- CONFIG_UBIFS_FS_LZO=y
- CONFIG_UBIFS_FS_ZLIB=y

Chapter 23
SPI NOR Flash Memory Technology Device (MTD) Driver

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.

23.1 Hardware Operation

On some boards, the SPI NOR - AT45DB321D is equipped, while on some boards M25P32 is equipped. Please 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 accesses 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.

23.2 Software Operation

In a Flash-based embedded Linux system, a number of Linux technologies work together to implement a file system. Figure 23-1 illustrates the relationships between some of the standard components.

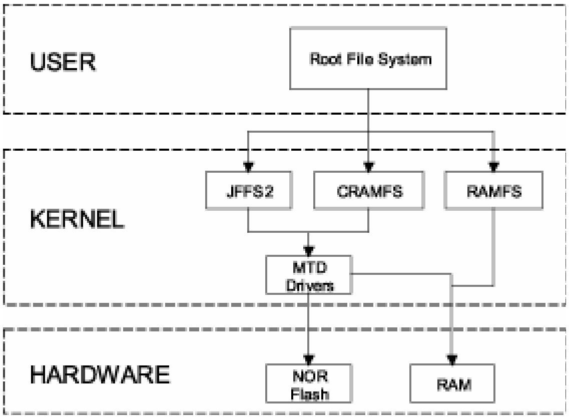

Figure 23-1. Components of a Flash-Based File System

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, erase NOR Flash.

23.3 Driver Features

This NOR MTD implementation supports the following features:

- Provides necessary information for the upper layer MTD driver

23.4 Source Code Structure

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

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

Table 23-1 shows the driver files:

Table 23-1. SPI NOR MTD Driver Files

File	Description
mxc_dataflash.c	Source file
mxc_m25p80.c	Source file

23.5 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_MXC_DATAFLASH: This config enables the access to AT DataFlash chips, using FSL SPI. In menuconfig, this option is available under

  Device Drivers > Memory Technology Device (MTD) support >Self-contained MTD device drivers > Support for AT DataFlash via FSL SPI interface
- CONFIG_MTD_MXC_M25P80: This enables access to most modern SPI flash chips, used program and data storage. Series supported include Atmel AT26DF Spansion S25SL, SST 25VF, ST M25P, and Winbond W25X ... . In menuconfig, this option is available under

  Device Drivers > Memory Technology Device (MTD) support >Self-contained MTD device drivers > Support SPI Flash chips (M25P, ...) via FSL SPI interface

Chapter 24
Low-Level Keypad Driver

The low-level keypad driver interfaces with the Keypad Port Hardware (KPP) in the i.MX device. The keypad driver is implemented as a standard Linux 2.6 keyboard driver, modified for the i.MX device.

The keypad driver supports the following features:

- Interrupt-driven scan code generation for keypress and release on a keypad matrix
- Keypad as a standard input device

The keypad driver can be accessed through the `/dev/input/event0` device file. The numbering of the event node depends on whether other input devices are loaded or not.

24.1 Hardware Operation

The KPP driver supports a keypad matrix with as many as eight rows and eight columns. Any pins that are not being used for the keypad are available as general purpose input/output pins. The actual keypad matrix is dependent on hardware connection.

The keypad port interfaces with a keypad matrix. On a keypress, the intersecting row and column lines are shorted together. The keypad has two mode of operation, Run mode and Low Power mode. In both modes the KPP detects any keypress event, but in low power mode the keypress event is detected even when the MCU clock is not running.

24.2 Software Operation

The keypad driver generates scan-codes for key press and release events on the keypad matrix. The operation is as follows:

1. When a key is pressed on the keypad, the keypad interrupt handler is called
2. In the keypad interrupt handler, the `mxc_kpp_scan_matrix` function is called to scan for key-presses and releases
3. The keypad scan timer function is called every 10 ms to scan for any keypress or release on the keypad
4. The scan-code for the keypress or release is generated by the `mxc_kpp_scan_matrix` function
5. The generated scancodes are converted to input device keycodes using the `mxckpd_keycodes` array

Every keypress or release follows the debounce state machine shown in Figure 24-1. The `mxc_kpp_scan_matrix` function is called for every keypress and release interrupt.

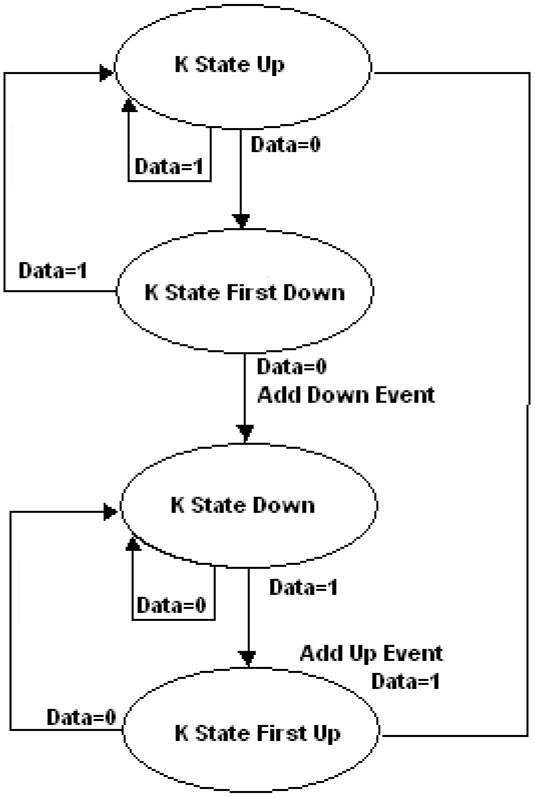

Figure 24-1. Keypad Driver State Machine

The keypad driver registers the input device structure within the `__init` function by calling `input_register_device(&mxckbd_dev)`.

The driver sets input bit fields and conveys all the events that can be generated by this input device to other parts of the input systems. The keypad driver can generate only `EV_KEY` type events. This can be indicated using `__set_bit(EV_KEY, mxckbd_dev.evbit)`.

The keypress key codes are reported by calling `input_event()`. The reported key press/release events are passed to the event interface (`/dev/input/event0`). This event interface is created when the `evdev.c` executable, located in `<ltib_dir>/rpm/BUILD/linux/drivers/input`, is compiled. The event interface is a generic input event interface. It passes the events generated in the kernel to the user space with timestamps. Blocking reads, non-blocking reads and `select()` can be done on `/dev/input/event0`.

The structure of `input_event` is as follows:

```
struct input_event {
        struct timeval time;
        unsigned short type;
        unsigned short code;
        unsigned int value;
        };
```

where:

- *time* is the timestamp at which the key event occurred
- *code* is the i.MX keycode for keypress or release
- *value* equals 0 for key release and 1 for keypress

The functions mentioned in this section are implemented as a low-level interface between the Linux OS and the KPP hardware. They cannot be called from other drivers or from a user application.

The keypress and release scancodes can be derived using the following formula,

```
scancode (press)   = (row × 8) + col;
scancode (release) = (row × 8) + col + 128;
```

24.3 Reassigning Keycodes

The keypad driver takes advantage of the input subsystem's ability to remap key codes. A user space application can use the `EVIOCGKEYCODE` and `EVIOCSKEYCODE` IOCTLs on the device node (for example `/dev/input/event0`) to get and set key codes. Applications such as keyfuzz and input-kbd (from the input-utils package) use these IOCTLs which are handled by the input subsystem. See the *kernel* *`Documentation/input/input-programming.txt`* for details on remapping codes.

24.4 Driver Features

The keypad driver supports the following features:

- Returns the input keycode for every key that is pressed or released
- Interrupt driver for keypress or release
- Blocking and nonblocking reads
- Implemented as a standard input device

24.5 Source Code Structure

Table 24-1 shows the keypad driver source files that are available in the following directories:

```
<ltib_dir>/rpm/BUILD/linux/drivers/input/keyboard
<ltib_dir>/rpm/BUILD/linux/include/linux
```

Table 24-1. Keypad Driver Files

File	Description
mxc_keyb.c	Low-level driver implementation
mxc_keyb.h	Driver structures, control register address definitions
input.h	Generic Linux keycode definitions
arch/arm/mach-mx5/mx50_rdp.c arch/arm/mach-mx5/device.c	Contains the platform-specific keymapping keycode array

24.6 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_MXC_KEYBOARD—MXC Keypad driver used for the MXC KPP. In menuconfig this option is available under

  Device Drivers > Input device support > Keyboards > MXC Keypad Driver.
- CONFIG_INPUT_EVDEV—Enabling this option creates the device node `/dev/input/event0`. In menuconfig, this option is available under

  Device Drivers > Input device support > Event interface.

The following source code configuration options are available for this module:

- Matrix config—The keypad matrix can be configured for up to eight rows and eight columns. The keypad matrix configuration can be done by changing the `rowmax` and `colmax` members in the `keypad_plat_data` structure in the platform specific file (see Table 24-1).
- Debounce delay—The user can configure the debounce delay by changing the variable `KScanRate` defined in `mxc_keyb.c`.

24.7 Programming Interface

User space applications can get information about the keypad driver through the standard proc and sysfs files such as `/proc/bus/input/devices` and the files under `/sys/class/input/event0/`.

24.8 Interrupt Requirements

Table 24-2 lists the keypad interrupt timer requirements.

Table 24-2. Keypad Interrupt Timer Requirements

Parameter	Equation	Typical	Worst-Case
Key scanning interrupt	(X number of instruction/MHz) × 64	(X/MHz) × 64	(X/MHz) × 64
Alarm for key polling	None	10 ms	10 ms

Chapter 25
Fast Ethernet Controller (FEC) Driver

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 or 100 Mbps related 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)
- Transmit 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).

25.1 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 the 10 Mbps-only 7-wire serial network interface (SNI), which uses a subset of the MII pins.

A brief overview of the device functionality is provided here. For details see the FEC chapter of the *MX50Multimedia Applications Processor Reference Manual*.

In MII mode, there are 18 signals defined by the IEEE 802.3 standard and supported by the EMAC. SNI and RMIImode uses a subset of the 18 signals. These signals are listed in Table 25-1.

Table 25-1. Pin Usage in MIIRMII and SNI Modes

Direction	EMAC Pin Name	MII Usage	SNI Usage	RMII Usage
In/Out	FEC_MDIO	Management Data Input/Output	General I/O	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	Data out, bit 0
Out	FEC_TXD[1]	Data out, bit 1	General output	Data out, bit 1
Out	FEC_TXD[2]	Data out, bit 2	General output	Not Used
Out	FEC_TXD[3]	Data out, bit 3	General output	Not Used

Table 25-1. Pin Usage in MIIRMII and SNI Modes (continued)

Direction	EMAC Pin Name	MII Usage	SNI Usage	RMII Usage
Out	FEC_TX_EN	Transmit Enable	Transmit Enable	Transmit Enable
Out	FEC_TX_ER	Transmit Error	General output	Not Used
In	FEC_CRS	Carrier Sense	Not Used	Not Used
In	FEC_COL	Collision	Collision	Not Used
In	FEC_TX_CLK	Transmit Clock	Transmit Clock	Synchronous clock reference (REF_CLK)
In	FEC_RX_ER	Receive Error	General input	Receive Error
In	FEC_RX_CLK	Receive Clock	Receive Clock	Not Used
In	FEC_RX_DV	Receive Data Valid	Receive Data Valid	Not Used
In	FEC_RXD[0]	Data in, bit 0	Data in	Data in, bit 0
In	FEC_RXD[1]	Data in, bit 1	General input	Data in, bit 1
In	FEC_RXD[2]	Data in, bit 2	General input	Not Used
In	FEC_RXD[3]	Data in, bit 3	General input	Not Used

The MII management interface consists of two pins, FEC_MDIO and FEC_MDC. The FEC hardware operation can be divided in the following parts. For detailed information consult the *MX50Multimedia 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, state machine etc.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.

25.2 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

25.3 Source Code Structure

Table 25-2 shows the source files available in the `<ltib_dir>/rpm/BUILD/linux/drivers/net directory.`

Table 25-2. 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.

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

- CONFIG_FEC—Provided for this module. This option is available under
  Device Drivers > Network device support > Ethernet (10 or 100Mbit) > FEC Ethernet controller.

To mount NFS-rootfs through FEC, disable the other Network config in the menuconfig if need.

25.5 Programming Interface

Table 25-2 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.

25.5.1 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 */
};
/*
 *      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 */
```

25.5.2 Getting a MAC Address

.

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.

Chapter 26
Inter-IC (I^2C) Driver

I^2C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices. The I^2C driver for Linux has two parts:

- I^2C bus driver—low level interface that is used to talk to the I^2C bus
- I^2C chip driver—acts as an interface between other device drivers and the I^2C bus driver

26.1 I^2C Bus Driver Overview

The I^2C bus driver is invoked only by the I^2C chip driver and is not exposed to the user space. The standard Linux kernel contains a core I^2C module that is used by the chip driver to access the I^2C bus driver to transfer data over the I^2C bus. The chip driver uses a standard kernel space API that is provided in the Linux kernel to access the core I^2C module. The standard I^2C 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 I^2C 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 I^2C master mode

26.2 I^2C Device Driver Overview

The I^2C device driver implements all the Linux I^2C data structures that are required to communicate with the I^2C 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 I^2C bus. Internally, these API functions use the standard I^2C kernel space API to call the I^2C core module. The I^2C core module looks up the I^2C bus driver and calls the appropriate function in the I^2C 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.

26.3 Hardware Operation

The I^2C module provides the functionality of a standard I^2C master and slave. It is designed to be compatible with the standard Philips I^2C 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

26.4 Software Operation

The I^2C driver for Linux has two parts: an I^2C bus driver and an I^2C chip driver.

26.4.1 I^2C Bus Driver Software Operation

The I^2C 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 I^2C bus. The algorithm structure contains a pointer to a function that is called whenever the I^2C chip driver wants to communicate with an I^2C device.

During startup, the I^2C bus adapter is registered with the I^2C core when the driver is loaded. Certain architectures have more than one I^2C module. If so, the driver registers separate `i2c_adapter` structures for each I^2C module with the I^2C core. These adapters are unregistered (removed) when the driver is unloaded.

After transmitting each packet, the I^2C 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 I^2C bus driver uses wait queues for its operation, other device drivers should be careful not to call the I^2C API methods from an interrupt mode.

26.4.2 I^2C Device Driver Software Operation

The I^2C driver controls an individual I^2C device on the I^2C bus. A structure, `i2c_driver`, describes the I^2C 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 I^2C devices, after the driver is loaded. The `attach_adapter` callback function is called whenever a new I^2C bus driver is loaded in the system. When the I^2C 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.

26.5 Driver Features

The I^2C driver supports the following features:

- I^2C communication protocol
- I^2C master mode of operation

NOTE

The I^2C driver do not support the I^2C slave mode of operation.

26.6 Source Code Structure

Table 26-1 shows the I^2C bus driver source files available in the directory:

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

Table 26-1. I^2C Bus Driver Files

File	Description
mxc_i2c.c	I^2C bus driver source file
mxc_i2c_reg.h	Register definitions

26.7 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 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 > MXC I2C support.

26.8 Programming Interface

The I^2C device driver can use the standard SMBus interface to read and write the registers of the device connected to the I^2C bus. For more information, see `<ltib_dir>/rpm/BUILD/linux/include/linux/i2c.h`.

26.9 Interrupt Requirements

The I^2C module generates many kinds of interrupts. The highest interrupt rate is associated with the transfer complete interrupt as shown in Table 26-2.

Table 26-2. I^2C Interrupt Requirements

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

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 I^2C interface).

Chapter 27
Configurable Serial Peripheral Interface (CSPI) Driver

The CSPI driver implements a standard Linux driver interface to the CSPI controllers. It supports the following features:

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

27.1 Hardware Operation

CSPI is used for fast data communication with fewer software interrupts than conventional serial communications. Each CSPI 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 CSPI includes:

- Master/slave-configurable
- Two chip selects allowing a maximum of four different slaves each for master mode operation
- Up to 32-bit programmable data transfer
- 8 × 32-bit FIFO for both transmit and receive data
- Configurable polarity and phase of the Chip Select (SS) and SPI Clock (SCLK)

27.2 Software Operation

The following sections describe the CSPI software operation.

27.2.1 SPI Sub-System in Linux

The CSPI driver layer is located between the client layer (PMIC and SPI Flash are examples of clients) and the hardware access layer. Figure 27-1 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 complete asynchronously through completion callbacks. There are also some simple synchronous wrappers for those calls, including ones for common transaction types like writing a command and then reading its response.

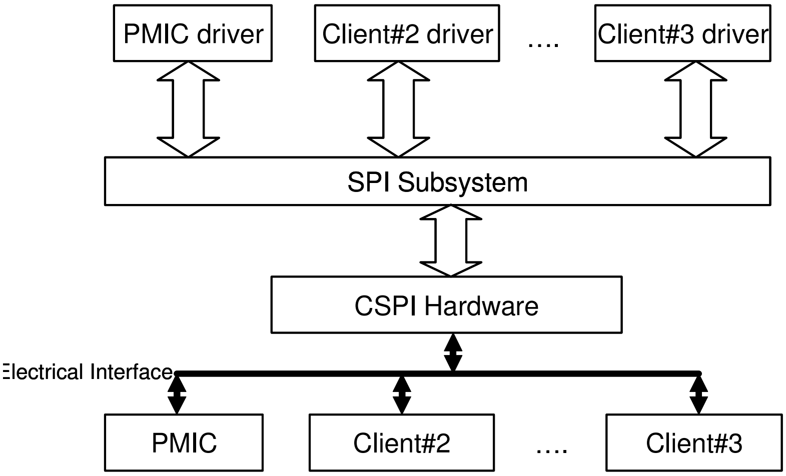

Figure 27-1. SPI Subsystem

All SPI clients must have a protocol driver associated with them and they must all be sharing the same controller driver. Only the controller driver can interact with the underlying SPI hardware module. Figure 27-2 shows how the different SPI drivers are layered in the SPI subsystem.

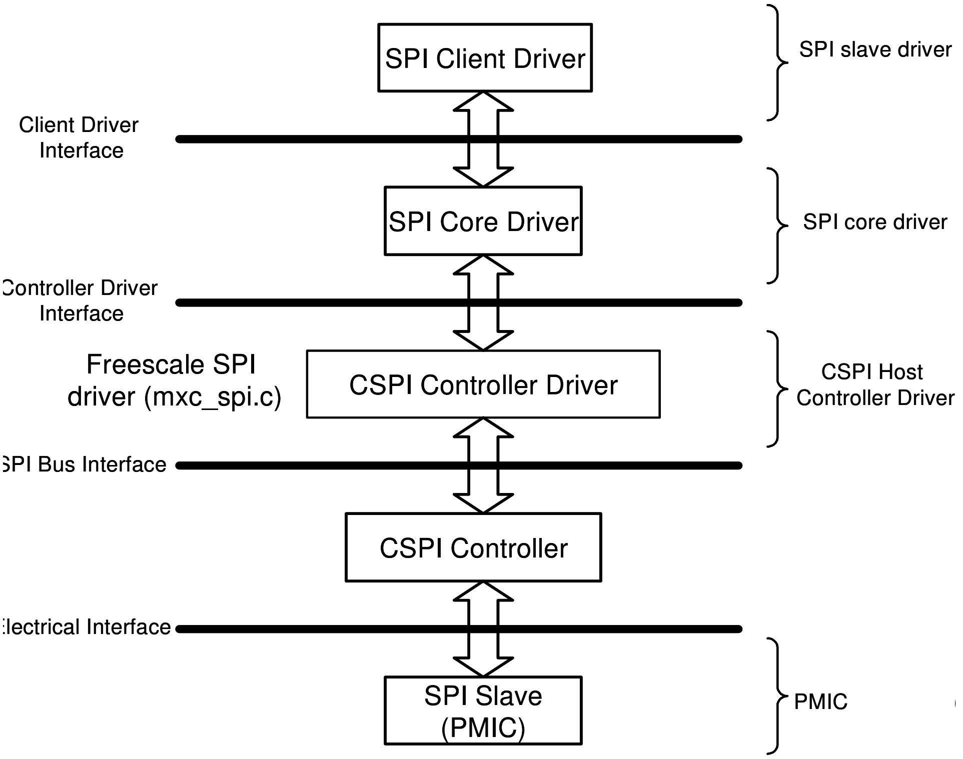

Figure 27-2. Layering of SPI Drivers in SPI Subsystem

27.2.2 Software Limitations

The CSPI 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

27.2.3 Standard Operations

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

- Init function `mxc_spi_init()`—Registers the `device_driver` structure.
- Probe function `mxc_spi_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 CSPI I/O pins, requests for IRQ and resets the hardware.
- Chip select function `mxc_spi_chipselect()`—Configures the hardware CSPI for the current SPI device. Sets the word size, transfer mode, data rate for this device.
- SPI transfer function `mxc_spi_transfer()`—Handles data transfers operations.
- SPI setup function `mxc_spi_setup()`—Initializes the current SPI device.
- SPI driver ISR `mxc_spi_isr()`—Called when the data transfer operation is completed and an interrupt is generated.

27.2.4 CSPI Synchronous Operation

Figure 27-3 shows how the CSPI provides synchronous read/write operations.

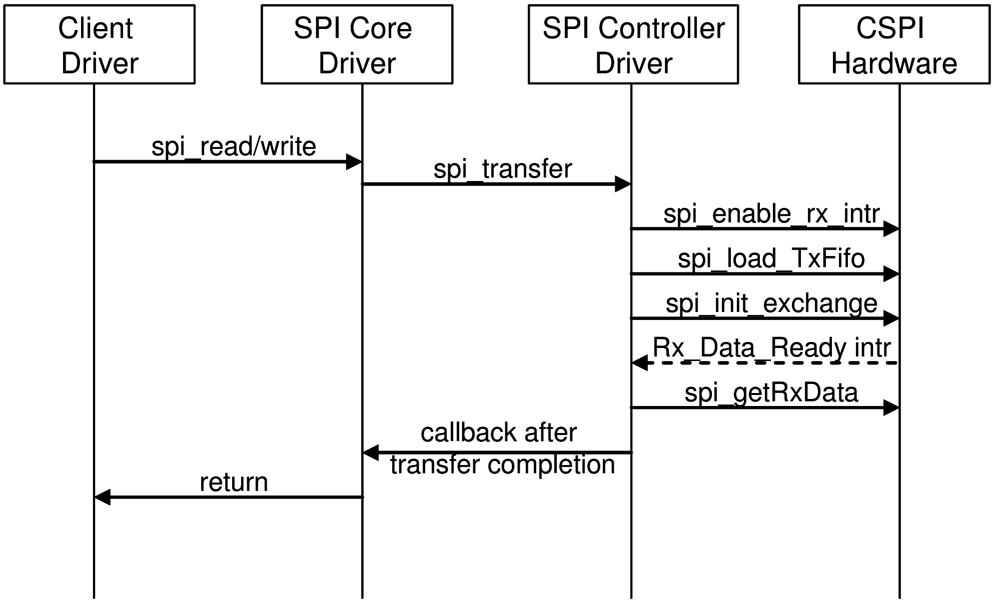

Figure 27-3. CSPI Synchronous Operation

27.3 Driver Features

The CSPI module supports the following features:

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

27.4 Source Code Structure

Table 27-1 shows the source files available in the devices directory:

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

Table 27-1. CSPI Driver Files

File	Description
mxc_spi.c	SPI Master Controller driver

27.5 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>`. 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—Library code that is automatically selected by drivers that need it. SPI_MXC selects it. In menuconfig, this option is available under

  Device Drivers > SPI Support > Utilities for Bitbanging SPI masters.
- CONFIG_SPI_MXC—Implements the SPI master mode for MXC CSPI. In menuconfig, this option is available under

  Device Drivers > SPI Support > MXC CSPI controller as SPI Master.
- CONFIG_SPI_MXC_SELECTn—Selects the CSPI hardware modules into the build (where n = 1 or 2). In menuconfig, this option is available under

  Device Drivers > SPI Support > CSPIn.
- CONFIG_SPI_MXC_TEST_LOOPBACK—To select the enable testing of CSPIs in loop back mode. In menuconfig, this option is available under

  Device Drivers > SPI Support > LOOPBACK Testing of CSPIs.

  By default this is disabled as it is intended to use only for testing purposes.

27.6 Programming Interface

This driver implements all the functions that are required by the SPI core to interface with the CSPI hardware. For more information, see the API document generated by Doxygen (in the doxygen folder of the documentation package).

27.7 Interrupt Requirements

The SPI interface generates interrupts. CSPI interrupt requirements are listed in Table 27-2.

Table 27-2. CSPI 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 28
1-Wire Driver

Each i.MX processor has an integrated 1-Wire interface. This driver is implemented as a character driver and provides a custom user space API that allows a user space application to interact with it.

28.1 Hardware Operation

The 1-Wire interface is used to connect to a battery monitor (Dallas DS2438Z). The 1-Wire interface reads battery current, voltage and other information. The EVK board by default does not support 1-Wire. Refer to the *MC1MX508 Multimedia Applications Processor Reference Manual* (MCIMX508RM for more information.

28.2 Software Operation

- The 1-Wire module software implementation conforms to Linux W1 driver.`/sys/class/power_supply/ interface`.

To avoid the conflict with the SPDIF module, a `w1` command option must be added in the launch command line for the 1-Wire.

28.3 Driver Features

The 1-Wire implementation supports the following features:

- i.MX 1-Wire module
- Reads battery information from DS2640

28.4 Source Code Structure

The 1-Wire master module is implemented in `<ltib_dir>/rpm/BUILD/linux/drivers/w1/masters`.

Table 28-1. 1-Wire Driver Files

File	Description
mxc_w1.c	1-Wire function implementation

The 1-Wire slave driver is located in `<ltib_dir>/rpm/BUILD/linux/drivers/w1/slaves/w1_ds2438.c`.

28.5 Menu Configuration Options

In order to get to the one-wire configuration, 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 1-Wire driver:

- Enable MXC 1-wire master. In menuconfig, this option is available under

  Device Driver > Dallas's 1-wire support > 1-wire Bus Masters > Freescale MXC driver for 1-wire
- Enable 1-Wire slave DS2438 driver. In menuconfig, this option is available under

  Device Driver > Dallas's 1-wire support > 1-wire Slaves > Smart Battery Monitor (DS2438)

Chapter 29
MMC/SD/SDIO Host Driver

The MultiMediaCard (MMC)/ Secure Digital (SD)/ Secure Digital Input Output (SDIO) Host driver implements a standard Linux driver interface to the enhanced MMC/SD host controller (eSDHC). 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 SD and SDIO cards
- 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 mode

29.1 Hardware Operation

The MMC communication is based on an advanced 11-pin serial bus designed to operate in a low voltage range. The eSDHC module support MMC along with SD memory and I/O functions. The eSDHC 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 eSDHC only support the SD bus protocol.

The eSDHC command transfer type and eSDHC command argument registers allow a command to be issued to the card. The eSDHC 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.

The block length register defines the number of bytes in a block (block size). As the Stream mode of MMC is not supported, the block length must be set for every transfer.

There are four 32-bit registers used to store the response from the card in the eSDHC. The eSDHC reads these four registers to get the command response directly. The eSDHC uses a fully configurable 128×32-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 eSDHC 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 eSDHC 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 eSDHC FIFO to system memory by reading the data buffer access register

To transmitting data, the steps are as follows:

1. The eSDHC 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 eSDHC FIFO by writing to the Data Buffer Access Register for a number of pre-defined bytes

The read-only eSDHC Present State and Interrupt Status Registers provide eSDHC 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 eSDHC interrupt status enable and signal enable registers allow the user to control if these interrupts occur.

29.2 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 eSDHC.

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 `sdhci_drv_init()`—Registers the device_driver structure.
- The probe function `sdhci_probe and sdhci_probe_slot()`—Performs initialization and registration of the MMC device specific structure with MMC bus protocol driver. The driver probes for memory and IRQ resources. Configures the IOMUX to enable eSDHC I/O pins and resets the hardware.
- `sdhci_set_ios()`—Sets bus width, voltage level, and clock rate according to core driver requirements.
- `sdhci_request()`—Handles both read and write operations. Sets up the number of blocks and block length. Configures an DMA channel, allocates safe DMA buffer and starts the DMA channel. Configures the eSDHC transfer type register eSDHC command argument register to issue a command to the card. This function starts the SDMA and starts the clock.
- MMC driver ISR `sdhci_cd_irq()`—Called when the MMC/SD card is detected or removed.
- MMC driver ISR `sdhci_irq()`—Interrupt from eSDHC called when command is done or errors like CRC or buffer underrun or overflow occurs.
- DMA completion routine `sdhci_dma_irq()`—Called after completion of a DMA transfer. Informs the MMC core driver of a request completion by calling `mmc_request_done()` API.

Figure 29-1 shows how the MMC-related drivers are layered.

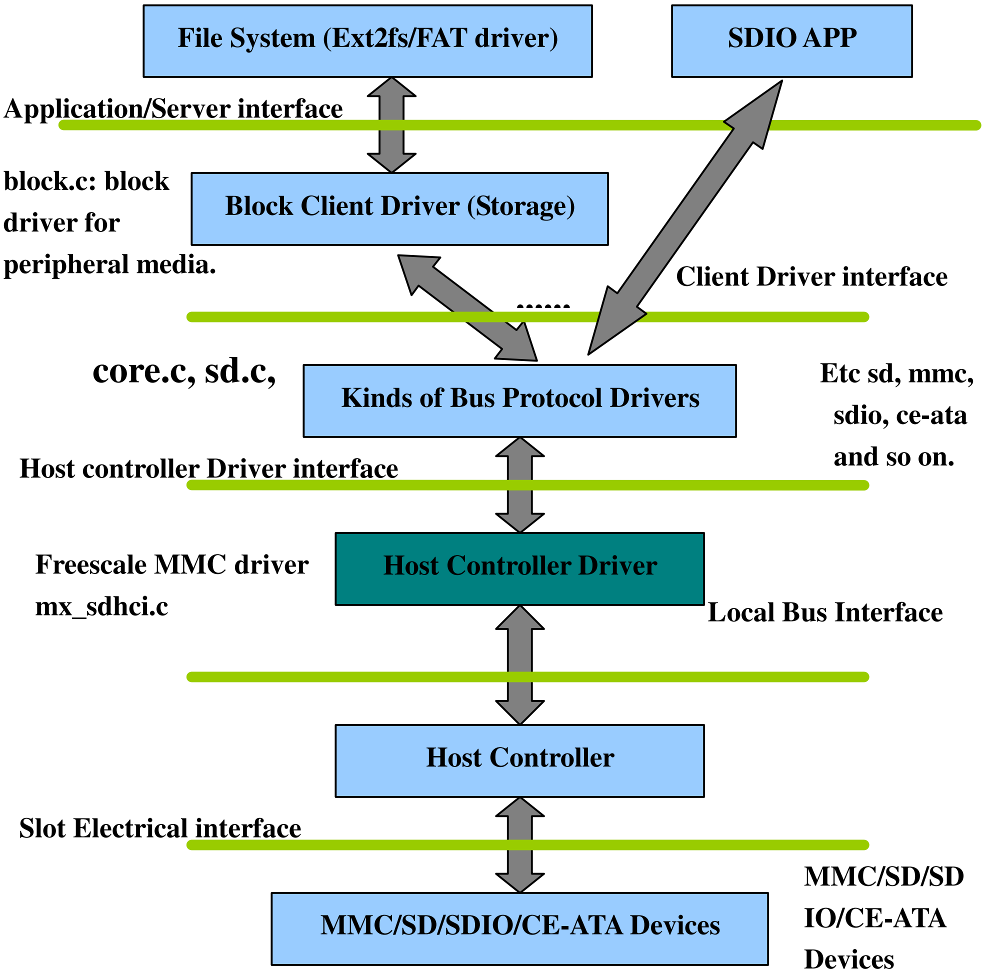
File System (Ext2fs/FAT driver)
SDIO APP
Application/Server interface
block.c: block driver for peripheral media.
Block Client Driver (Storage)
Client Driver interface
......
core.c, sd.c,
Kinds of Bus Protocol Drivers
Etc sd, mmc, sdio, ce-ata and so on.
Host controller Driver interface
Freescale MMC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

Figure 29-1. MMC Drivers Layering

29.3 Driver Features

The MMC driver supports the following features:

- Supports multiple eSDHC modules
- Provides all the entry points to interface with the Linux MMC core driver
- MMC and SD cards
- Recognizes data transfer errors such as command time outs and CRC errors
- Power management

29.4 Source Code Structure

Table 29-1 shows the eSDHC source files available in the source directory: `<ltib_dir>/rpm/BUILD/linux/drivers/mmc/host/.`

Table 29-1. eSDHC Driver FilesMMC/SD Driver Files

File	Description
mx_sdhci.h	Header file defining registers
mx_sdhci.c	eSDHC driver

29.5 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>`. 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—Build 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—Build 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_IMX_ESDHCI—Driver used for the i.MX eSDHC ports. In menuconfig, this option is found under

  Device Drivers > MMC/SD Card Support > Freescale i.MX Secure Digital Host Controller Interface support
- CONFIG_MMC_IMX_ESDHCI_PIO_MODE—Sets i.MX Multimedia card Interface to PIO mode. In menuconfig, this option is found under

  Device Drivers > MMC/SD Card support > Freescale i.MX Secure Digital Host Controller Interface PIO mode

  This option is dependent on CONFIG_MMC_IMX_ESDHCI. By default, this option is not set and DMA mode is used.
- CONFIG_MMC_UNSAFE_RESUME—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 > Allow unsafe resume.

29.6 Programming Interface

This driver implements the functions required by the MMC bus protocol to interface with the i.MX eSDHCmodule. See the *BSP API* document (in the doxygen folder of the documentation package), for additional information.

Chapter 30
ARC USB Driver

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/Full Speed Host Only core (HOST1)
- High speed and Full Speed OTG core
- Host mode—Supports HID (Human Interface Devices), MSC (Mass Storage Class), and PTP (Still Image) drivers
- Peripheral mode—Supports MSC, and CDC (Communication Devices Class) drivers
- Embedded DMA controller

30.1 Architectural Overview

A USB host system is composed of a number of hardware and software layers. Figure 30-1 shows a conceptual block diagram of the building block layers in a host system that support USB 2.0.

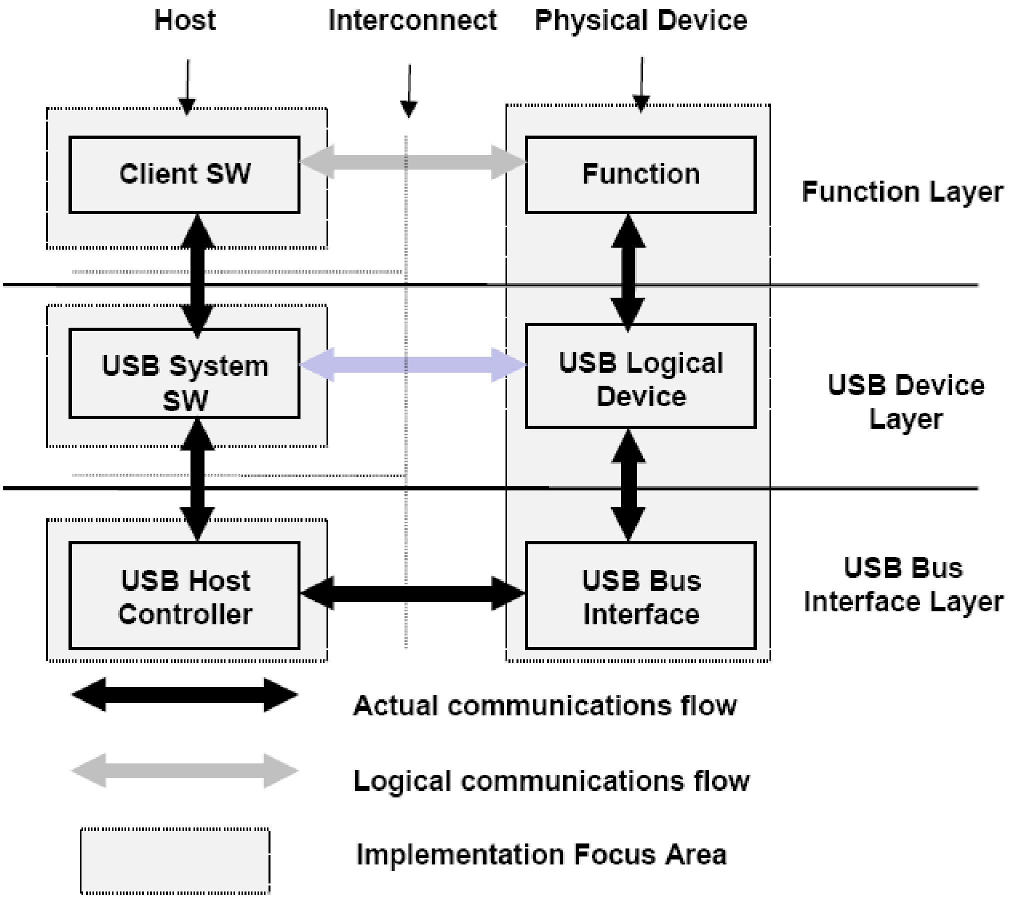

Figure 30-1. USB Block Diagram

30.2 Hardware Operation

For information on hardware operations, refer to the EHCI spec.ehci-r10.pdf available at http://www.usb.org/developers/docs/.

30.3 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, an OTG finish state machine (FSM) is implemented

30.4 Driver Features

The USB stack supports the following features:

- USB device mode
- Mass storage device profile—subclass 8-1 (RBC set)
- USB host mode
- HID host profile—subclasses 3-1-1 and 3-1-2. (USB mouse and keyboard)
- Mass storage host profile—subclass 8-1
- Ethernet USB profile—subclass 2
- DC PTP transfer

30.5 Source Code Structure

Table 30-1 shows the source files available in the source directory, `<ltib_dir>/rpm/BUILD/linux/drivers/usb`.

Table 30-1. 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
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 30-2 shows the platform related source files.

Table 30-2. 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 30-3 shows the platform-related source files in the directory:`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/`

Table 30-3. USB Platform Header Files

File	Description
usb_dr.c	Platform-related initialization

Table 30-4 shows the common platform source files in the directory:

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

Table 30-4. USB Common Platform Files

File	Description
utmixc.c	Internal UTMI transceiver driver
usb_common.c	Common platform related part of USB driver
usb_wakeup.c	Handle usb wakeup events

30.6 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_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 M.

  CONFIG_USB_EHCI_ARC—Build support for selecting the ARC EHCI host. In menuconfig, this option is available underDevice drivers > USB support > Support for Freescale controller.

  By default, this option is Y.
- CONFIG_USB_EHCI_ARC_H1—Build support for selecting the USB Host1. In menuconfig, this option is available underDevice drivers > USB support > Support for Host1 port on 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 > Support for Host-side USB > EHCI HCD (USB 2.0) support > Support for Freescale controller.

  By default, this option is N.
- CONFIG_USB_STATIC_IRAM—Build support for selecting the IRAM usage for host. In menuconfig, this option is available under

  Device drivers > USB support > Use IRAM for USB.

  By default, this option is N.
- CONFIG_USB_EHCI_ROOT_HUB_TT—Build support for OHCI or UHCI companion. 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_FSL && USB_SUPPORT.
- 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_USB_OTG—OTG Support, support dual role with ID pin detection.

  By default, this option is N.
- CONFIG_UTMI_MXC_OTG—USB OTG pin detect support for UTMI PHY, enable UTMI PHY for OTG support.

  By default, this option is N.
- 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.

30.7 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.

30.8 Default USB Settings

Table 30-5 shows the default USB settings.

Table 30-5. Default USB Settings

Platform	OTG HS	OTG FS	Host1	Host2(HS)	Host2(FS)
i.MX50 EVK	enable	N/A	enable (HS)	N/A	N/A

By default, both usb device and host function are build-in kernel, otg port is used for device mode, and host 1 is used for host mode.

The default configuration does not enable OTG port for both device and host mode. To enable USB-OTG for both host and device mode, configure the kernel as follows and rebuild the kernel and modules:

- CONFIG_USB_EHCI_ARC_OTG—Enable support for the USB OTG port in HS/FS Host mode.built as Y
- CONFIG_USB_GADGET—USB Gadget Support: built as y
- CONFIG_USB_OTG —OTG Support: built as Y
- CONFIG_MXC_OTG—USB OTG pin detect support for UTMI PHY: built as Y
- build USB GADGET driver as M, for example:
  CONFIG_USB_ETH CONFIG_USB_FILE_STORAGEthen , if you want to use EVK as mass storage device, insmod g_file_storage.ko file=/dev/mmcblk0p2
  if you want to use the otg as ethernet, insmod g_ether.ko , then you can use ifconfig usb0 to configure the ip

30.9 System WakeUp

- Both host and device connect/disconnect event can be system wakeup source

30.10 USB Wakeup usage

30.10.1 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
```

30.10.2 What kinds of wakeup event usb support

Take USBOTG port as the 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 (such as press usb key at 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 needs to enable hub's wakeup first. for remote wakeup, it needs to do below three steps:

```
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's wakeup,
that device connects at second level hub)
(1-1.1 is the usb device name)
```

30.10.3 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 31
Universal Asynchronous Receiver/Transmitter (UART) Driver

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 through the device files `/dev/ttymxc0` through `/dev/ttymxc4`, where `/dev/ttymxc0` refers to UART 1. Autobaud detection is not supported.

31.1 Hardware Operation

Refer to the *MX50 Multimedia 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.

31.2 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 this 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. The 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.

31.3 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

31.4 Source Code Structure

Table 31-1 shows the UART driver source files that are available in the directory:

`<ltib_dir>/rpm/BUILD/linux/drivers/serial.`

Table 31-1. UART Driver Files

File	Description
mxc_uart.c	Low level driver
serial_core.c	Core driver that is included as part of standard Linux
mxc_uart_reg.h	Register values
mxc_uart_early.c	Source file to support early serial console for UART

Table 31-2 shows the header files associated with the UART driver.

Table 31-2. UART Global Header Files

File	Description
<ltib_dir>/rpm/BUILD/linux/ arch/arm/plat-mxc/include/mach/mxc_uart.h	UART header that contains UART configuration data structure definitions

The source files, `serial.c and serial.h`, are associated with the UART driver that is available in the directory: `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5`. The source file contains UART configuration data and calls to register the device with the platform bus.

31.5 Configuration

This section discusses configuration options associated with Linux, chip configuration options, and board configuration options.

31.5.1 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_SERIAL_MXC—Used for the UART driver for the UART ports. In menuconfig, this option is available under

Device Drivers > Character devices > Serial drivers > MXC Internal serial port support.

By default, this option is Y.

- CONFIG_SERIAL_MXC_CONSOLE—Chooses the Internal UART to bring up the system console. This option is dependent on the CONFIG_SERIAL_MXC option. In the menuconfig this option is available under

  Device Drivers > Character devices > Serial drivers > MXC Internal serial port support > Support for console on a MXC/MX27/MX21 Internal serial port.

  By default, this option is Y.

31.5.2 Source Code Configuration Options

This section details the chip configuration options and board configuration options.

31.5.2.1 Chip Configuration Options

The following chip-specific configuration options are provided in `mxc_uart.h`. The x in `UARTx` denotes the individual UART number. The default configuration for each UART number is listed in Table 31-5.

31.5.2.2 Board Configuration Options

The following board specific configuration options for the driver can be set within

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx/board-mx_.h`:

- UART Mode (`UARTx_MODE`)—Specifies DTE or DCE mode
- UART IR Mode (`UARTx_IR`)—Specifies whether the UART port is to be used for IrDA.
- UART Enable / Disable (`UARTx_ENABLED`)—Enable or disable a particular UART port; if disabled, the UART is not registered in the file system and the user can not access it

For i.MX508, the board specific configuration options for the driver is set within

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/serial.c`

31.6 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.

31.7 Interrupt Requirements

The UART driver interface generates many kinds of interrupts. The highest interrupt rate is associated with transmit and receive interrupt.

The system requirements are listed in Table 31-3.

Table 31-3. UART Interrupt Requirements

Parameter	Equation	Typical	Worst Case
Rate	(BaudRate/(10))*(1/Rxtl + 1/(32–Txtl))	5952/sec	300000/sec
Latency	320/BaudRate	5.6 ms	213.33 µs

The baud rate is set in the `mxcuart_set_termios` function. The typical values are based on a baud rate of 57600 with a receiver trigger level (Rxtl) of one and a transmitter trigger level (Txtl) of two. The worst case is based on a baud rate of 1.5 Mbps (maximum supported by the UART interface) with an Rxtl of one and a Txtl of 31. There is also an undetermined number of handshaking interrupts that are generated but the rates should be an order of magnitude lower.

31.8 Device Specific Information

31.8.1 UART Ports

The UART ports can be accessed through the device files `/dev/ttymxc0`, `/dev/ttymxc1`, and so on, where `/dev/ttymxc0` refers to UART 1. The number of UART ports on a particular platform are listed in Table 31-4.

31.8.2 Board Setup Configuration

Table 31-4. UART General Configuration

Platform	Number of UARTs	Max Baudrate
i.MX508	3	4Mbps

Table 31-5. UART Active/Inactive Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	1	1	1	0	0	0

Table 31-6. UART IRDA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	NO_IRDA	NO_IRDA	NO_IRDA			

Table 31-7. UART Mode Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	MODE_DCE	MODE_DCE	MODE_DCE			

Table 31-8. UART Shared Peripheral Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	-1	-1	-1			

Table 31-9. UART Hardware Flow Control Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	0	0	0			

Table 31-10. UART DMA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	0	0	1			

Table 31-11. UART DMA RX Buffer Size Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	1024	512	1024			

Table 31-12. UART UCR4_CTSTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	16	-1	16			

Table 31-13. UART UFCR_RXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	16	16	16			

Table 31-14. UART UFCR_TXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	16	16	16			

Table 31-15. UART Interrupt Mux Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	INTS_MUXED	INTS_MUXED	INTS_MUXED			

Table 31-16. UART Interrupt 1 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	MXC_INT_UART1	MXC_INT_UART2	MXC_INT_UART3			

Table 31-17. UART Interrupt 2 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	-1	-1	-1			

Table 31-18. UART interrupt 3 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	-1	-1	-1			

31.9 Early UART Support

The kernel starts logging messages on a serial console when it knows where the device is located. This happens when the driver enumerates all the serial devices, which can happen a minute or more after the kernel begins booting.

Linux kernel 2.6.10 and later kernels have an early UART driver that operates very early in the boot process. The kernel immediately starts logging messages, if the user supplies an argument as follows:

```
console=mxcuart,0xphy_addr,115200n8
```

Where `phy_addr` represents the physical address of the UART on which the console is to be used and `115200n8` represents the baud rate supported.

Chapter 32
Secure Real Time Clock (SRTC) Driver

The Secure 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.

32.1 Hardware Operation

The SRTC is a real time clock with enhanced security capabilities. It provides an accurate, 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 pre-set alarm is reached.

32.2 Software Operation

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

32.2.1 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

In addition, the following IOCTLS were added to allow user application such as DRM to track changes in the time, which is user settable. The DRM application needs a way to track how much the time changed by so that it can manage its own secure clock = SRTC + secureclk_offset. The secureclk_offset should be calculated by the DRM application based on changes to the SRTC time counter.

- RTC_READ_TIME_47BIT: allows a read of the 47-bit LP time counter on SRTC
- RTC_WAIT_FOR_TIME_SET: allows user thread to block until 47-bit LP time counter is set. At which point, the user thread is woken up and is provided the SRTC offset (which is the difference between the new and old LP counter)

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
```

32.2.2 Keep Alive in the Power Off State

To keep 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. For example, CKIL and NVCC_SRTC_POW can be connected to the MC34708 CLK32KMCU and VSRTC. Bit 4, DRM, of the MC34708 Power Control 0 Register can be enabled to keep VSRTC and CLK32KMCU on for all states.

Ordinarily, 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 recommend to automatically enable the coin-cell charger so that the SRTC is properly powered.

32.3 Driver Features

The SRTC driver includes the following features:

- Implements all the functions required by Linux to provide the real time clock and alarm interrupt
- Reserves time in power off state
- Alarm wakes up the system from low power modes

32.4 Source Code Structure

The RTC module is implemented in the following directory:

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

Table 32-1 shows the RTC module files.

Table 32-1. RTC Driver Files

File	Description
rtc-mxc_v2.c	SRTC driver implementation file

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

32.5 Menu Configuration Options

To get to the SRTC 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 SRTC driver:

- Device Drivers > Real Time Clock > Freescale MXC Secure Real Time Clock

Chapter 33
Watchdog (WDOG) Driver

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.

33.1 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.

33.2 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.

33.3 Generic WDOG Driver

The generic WGOD driver is implemented in the `<ltib_dir>/rpm/BUILD/linux/drivers/watchdog/mxc_wdt.c` file. It provides functions for various IOCTLs and read/write calls from the user level program to control the WDOG.

33.3.1 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

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

- CONFIG_MXC_WATCHDOG—Enables Watchdog timer module. This option is available under Device Drivers > Watchdog Timer Support > MXC watchdog.

33.3.3 Source Code Structure

Table 33-1 shows the source files for WDOG drivers that are in the following directory:

`<ltib_dir>/rpm/BUILD/linux/drivers/watchdog.`

Table 33-1. WDOG Driver Files

File	Description
mxc_wdt.c	WDOG function implementations
mxc_wdt.h	Header file for WDOG implementation

Watchdog system reset function is located under

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

33.3.4 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 34
Pulse-Width Modulator (PWM) Driver

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 4×16 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.

34.1 Hardware Operation

Figure 34-1 shows the PWM block diagram.

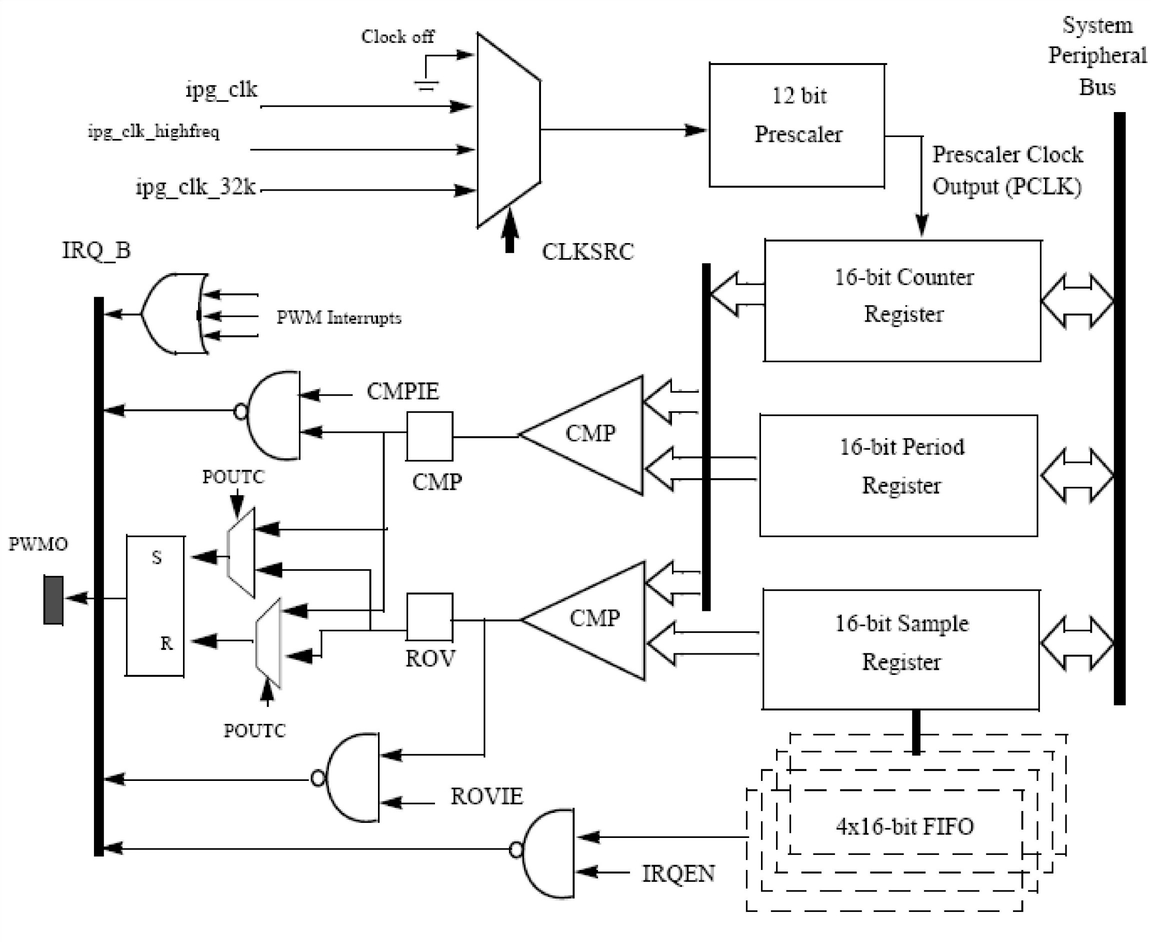

Figure 34-1. PWM Block Diagram

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

34.2 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.

34.3 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 34-1 provides a summary of the interface functions in source code.

Table 34-1. 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.

34.4 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

34.5 Source Code Structure

Table 34-2 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 34-2. PWM Driver Files

File	Description
pwm.h	Functions declaration
pwm.c	Functions definition

34.6 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 35
HDMI Driver

The HDMI module is supported by a SII902x chip on board or a daughter card, it supports:

- HDMI video output
- Display device EDID fetching
- HDMI audio output

Please refer to SII902x's datasheet for detail information.

35.1 Hardware Operation

Normally, the SII902x chip connects with

- IPUv3 DI port for display support
- SoC SPDIF port for audio support
- I2C bus for setup operation and EDID fetching
- A cable detection interrupt line

35.2 Software Operation

The HDMI driver is implemented based on a fb clienct driver, it mainly contains:

- A fb event notifier callback
- A delay workqueue based on interrupt serves for cable detection and EDID fetching

In fb event notifier callback:

- An event as FB_EVENT_MODE_CHANGE will setup SII902x video and audio parameters
- An event as FB_EVENT_BLANK will be resiponsible for SII902x power on/off

For cable detection and EDID fetching:

- An irq handler -- sii902x_detect_handler() is registered
- A delay workqueue will be scheduled when interrupt occurs
- Workqueue callback function -- det_worker() will triger EDID fetching
- A sysfs node -- /sys/devices/platform/sii9022.0/cable_state is provided to show cable state
- An uevent with EVENT=plugin or EVENT=plugout will send out after hdmi cable plugin/out

35.3 Source Code Structure

Table 35-1 lists the TVE driver source files available in the `<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc` directory.

Table 35-1. Camera File List

File	Description
mxcfb_sii902x.c	HDMI Sii902x driver implementation

35.4 Linux Menu Configuration Options

The Linux kernel configuration option, CONFIG_FB_MXC_SII902X, is provided for the module. This is the configuration option for the HDMI SII902x driver. In `menuconfig`, this option is available under

Device Drivers > Graphics support > Si Image SII9022 DVI/HDMI Interface Chip

This option is dependent on the support Synchronous Panel Framebuffer and IPUv3 option. By default, this option is Y.

Chapter 36
Frequently Asked Questions

36.1 NFS Mounting Root File System

1. Assuming the root file system is under, modify the `/etc/exports` file on the Linux host by adding the following line:
   /tmp/fs *(rw,no_root_squash)/tools/rootfs *(rw,sync,no_root_squash)
2. Make sure the NFS service is started on the Linux host machine. To start it on the host machine, issue:

   Install if not already installed.

36.2 Using the Memory Access Tool

The Memory Access Tool is used to access kernel memory space from user space. The tool can be used to dump registers or write registers for debug purposes.

To use this tool, run the executable file `memtool` located in `/unit_test`:

- Type `memtool` without any arguments to print the help information
- Type `memtool [-8 | -16 | -32] addr count` to read data from a physical address
- Type `memtool [-8 | -16 | -32] addr=value` to write data to a physical address

If a size parameter is not specified, the default size is 32-bit access. All parameters are in hexadecimal.

Chapter 37
OProfile

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.

37.1 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.

37.2 Features

The features of the OProfile are as follows:

- 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 Cortex-A8 based platforms.

37.3 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.

37.4 Software Operation

37.4.1 Architecture Specific Components

If 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.

37.4.2 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.

37.4.3 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.

37.4.4 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).

37.4.5 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.

37.5 Requirements

The requirements of OProfile are as follows:

37.6 Source Code Structure

Oprofile platform-specific source files are available in the directory:

`<ltib_dir>/rpm/BUILD/linux/arch/arm/oprofile/`

Table 37-1. 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/`

37.7 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)

37.8 Programming Interface

This driver implements all the methods required to configure and control PMU and L2 cache EVTMON counters. Refer to the doxygen documentation for more information (in the doxygen folder of the documentation package).

37.9 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.

37.10 Example Software Configuration

The following steps show and example of how to configure the OProfile:

1. Use the command `./ltib -c` from the <ltib dir>. On the screen, first go to Package list and select oprofile. The current version in ltib is 0.9.5.
2. Then return to the first screen and select `Configure Kernel`, follow the instruction from Section 37.7, “Menu Configuration Options,” to enable Oprofile in the kernel space.
3. Save the configuration and start to build.
4. Copy Oprofile binaries to target rootfs. Copy vmlinux to `/boot` directory and run Oprofile

```
root@ubuntu:/boot# opcontrol --separate=kernel --vmlinux=/boot/vmlinux
root@ubuntu:/boot# opcontrol --reset
Signalling daemon... done
root@ubuntu:/boot# opcontrol --setup --event=CPU_CYCLES:100000
root@ubuntu:/boot# opcontrol --start
Profiler running.
root@ubuntu:/boot# opcontrol --dump
root@ubuntu:/boot# opreport
Overflow stats not available
CPU: ARM V7 PMNC, speed 0 MHz (estimated)
Counted CPU_CYCLES events (Number of CPU cycles) with a unit mask of 0x00 (No un
it mask) count 100000
CPU_CYCLES:100000|
  samples|      %|
------------------
        4 22.2222 grep
        CPU_CYCLES:100000|
          samples|      %|
        ------------------
                4 100.000 libc-2.9.so
        2 11.1111 cat
        CPU_CYCLES:100000|
          samples|      %|
        ------------------
                1 50.0000 ld-2.9.so
                1 50.0000 libc-2.9.so
...
root@ubuntu:/boot# opcontrol --stop
Stopping profiling.
```