i.MX53 EVK Linux

Reference Manual

Part Number: 924-76374
Rev.10.10.01
10/2010

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

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

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

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

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

Asia/Pacific:
Freescale Semiconductor China Ltd.
Exchange Building 23F
No. 118 Jianguo Road
Chaoyang District
Beijing 100022
China
+86 10 5879 8000
support.asia@freescale.com

For Literature Requests Only:
Freescale Semiconductor
Literature Distribution Center
1-800 441-2447 or
+1-303-675-2140
Fax: +1-303-675-2150
LDCForFreescaleSemiconductor
@hibbertgroup.com

ARM POWERED®

freescale™ semiconductor

Contents

About This Book

Chapter 1 Introduction

Chapter 2 Architecture

Chapter 3
Machine Specific Layer (MSL)

Chapter 4
Smart Direct Memory Access (SDMA) API

Chapter 3
MC13892 Regulator Driver

Chapter 4
MC13892 RTC Driver

Chapter 5
MC13892 Digitizer Driver

Chapter 6
CPU Frequency Scaling (CPUFREQ) Driver

Chapter 7
Low-level Power Management (PM) Driver

Chapter 8
Dynamic Voltage Frequency Scaling (DVFS) Driver

Chapter 9
Software Based Peripheral Domain Frequency Scaling

Chapter 10
Image Processing Unit (IPU) Drivers

Chapter 11
Video for Linux Two (V4L2) Driver

Chapter 12
LVDS Display Bridge(LDB) Driver

Chapter 13
i.MX5 Dual Display

Chapter 14
Video Processing Unit (VPU) Driver

Chapter 15
Graphics Processing Unit (GPU)

Chapter 16
TV Decoder (TV-In) Driver

Chapter 17
Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver

Chapter 18
The Sony/Philips Digital Interface (S/PDIF) Tx Driver

Chapter 19
SPI NOR Flash Memory Technology Device (MTD) Driver

Chapter 20
NAND Flash Memory Technology Device (MTD) Driver

Chapter 21
SATA Driver

Chapter 22
Low-Level Keypad Driver

Chapter 23
Fast Ethernet Controller (FEC) Driver

Chapter 24
Inter-IC (I2C) Driver

Chapter 25
Configurable Serial Peripheral Interface (CSPI) Driver

Chapter 26
MMC/SD/SDIO Host Driver

Chapter 27
Universal Asynchronous Receiver/Transmitter (UART) Driver

Chapter 28
ARC USB Driver

Chapter 29
Secure Real Time Clock (SRTC) Driver

Chapter 30
Watchdog (WDOG) Driver

Chapter 31
Pulse-Width Modulator (PWM) Driver

Chapter 32
FlexCAN Driver

Chapter 33
Media Local Bus Driver

Chapter 34
OProfile

Chapter 35
Frequently Asked Questions

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

Chapter 1
Introduction

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

- i.MX53 Applications Processor

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

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

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

1.1 Software Base

The i.MX BSP is based on version 2.6.35.3 of the Linux kernel from the official Linux kernel web site (http://www.kernel.org). It is enhanced with the features provided by Freescale.

1.2 Features

Table 1-1 describes the features supported by the Linux BSP for specific platforms.

Table 1-1. Linux BSP Supported Features

Feature	Description	Chapter Source	Applicable Platform
Machine Specific Layer			
MSL	Machine Specific Layer (MSL) supports interrupts, Timer, Memory Map, GPIO/IOMUX, SPBA, SDMA. • Interrupts (AITC/AVIC): The Linux kernel contains common ARM code for handling interrupts. The MSL contains platform-specific implementations of functions for interfacing the Linux kernel to the interrupt controller. • Timer (GPT): The General Purpose Timer (GPT) is set up to generate an interrupt as programmed to provide OS ticks. Linux facilitates timer use through various functions for timing delays, measurement, events, alarms, high resolution timer features, and so on. Linux defines the MSL timer API required for the OS-tick timer and does not expose it beyond the kernel tick implementation. • GPIO/EDIO/IOMUX: The GPIO and EDIO components in the MSL provide an abstraction layer between the various drivers and the configuration and utilization of the system, including GPIO, IOMUX, and external board I/O. The IO software module is board-specific, and resides in the MSL layer as a self-contained set of files. I/O configuration changes are centralized in the GPIO module so that changes are not required in the various drivers. • SPBA: The Shared Peripheral Bus Arbiter (SPBA) provides an arbitration mechanism among multiple masters to allow access to the shared peripherals. The SPBA implementation under MSL defines the API to allow different masters to take or release ownership of a shared peripheral.	Chapter 3, “Machine Specific Layer (MSL)”	All
SDMA API	The Smart Direct Memory Access (SDMA) API driver controls the SDMA hardware. It provides an API to other drivers for transferring data between MCU, DSP and peripherals. The SDMA controller is responsible for transferring data between the MCU memory space, peripherals, and the DSP memory space. The SDMA API allows other drivers to initialize the scripts, pass parameters and control their execution. SDMA is based on a microRISC engine that runs channel-specific scripts.	Chapter 4, “Smart Direct Memory Access (SDMA) API”	i.MX53
MC13892 Regulator	MC13892 regulator driver provides the low-level control of the power supply regulators, setting voltage level and enable/disable regulators.	Chapter 3, “MC13892 Regulator Driver”	i.MX53
MC13892 RTC	MC13892 RTC driver for Linux provides the access to PMIC RTC control circuits	Chapter 4, “MC13892 RTC Driver”	i.MX53

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

Feature	Description	Chapter Source	Applicable Platform
MC13892 Digitizer Driver	MC13892 digitizer driver for Linux that provides low-level access to the PMIC analog-to-digital converters	Chapter 5, “MC13892 Digitizer Driver”	i.MX53
Power Management Drivers			
Low-level PM Drivers	The low-level power management driver is responsible for implementing hardware-specific operations to meet power requirements and also to conserve power on the development platforms. Driver implementations are often different for different platforms. It is used by the DPM layer.	”Chapter 7, “Low-level Power Management (PM) Driver”	i.MX53
CPU Frequency Scaling	The CPU frequency scaling device driver allows the clock speed of the CPUs to be changed on the fly.	Chapter 6, “CPU Frequency Scaling (CPUFREQ) Driver”	i.MX53
DVFS	The Dynamic Voltage Frequency Scaling (DVFS) device driver allows simple dynamic voltage frequency scaling. The frequency of the core clock domain and the voltage of the core power domain can be changed on the fly with all modules continuing their normal operations.	Chapter 8, “Dynamic Voltage Frequency Scaling (DVFS) Driver	i.MX53
Multimedia Drivers			
IPU	The Image Processing Unit (IPU) is designed to support video and graphics processing functions and to interface with video/still image sensors and displays. The IPU driver is a self-contained driver module in the Linux kernel. It contains a custom kernel-level API to manipulate logical channels. A logical channel represents a complete IPU processing flow. The IPU driver includes a frame buffer driver, a V4L2 device driver, and low-level IPU drivers.	Chapter 10, “Image Processing Unit (IPU) Drivers”	i.MX53
TV-IN (ADV7180)	The ADV7180 TV-IN driver is designed under Linux V4L2 architecture. It implements the V4L2 capture interface.	Chapter 16, “TV Decoder (TV-In) Driver”	i.MX53
V4L2 Output	The Video for Linux 2 (V4L2) output driver uses the IPU post-processing functions for video output. The driver implements the standard V4L2 API for output devices.	Chapter 11, “Video for Linux Two (V4L2) Driver”	i.MX53
V4L2 Capture	The Video for Linux 2 (V4L2) capture device includes two interfaces: the capture interface and the overlay interface. The capture interface records the video stream. The overlay interface displays the preview video.	Chapter 11, “Video for Linux Two (V4L2) Driver”	i.MX53
VPU	The Video Processing Unit (VPU) is a multi-standard video decoder and encoder that can perform decoding and encoding of various video formats.	Chapter 14, “Video Processing Unit (VPU) Driver”	i.MX53
AMD GPU	The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D/3D graphics applications.	Chapter 15, “Graphics Processing Unit (GPU)”	i.MX53

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

Feature	Description	Chapter Source	Applicable Platform
Sound Drivers			
ALSA Sound	The Advanced Linux Sound Architecture (ALSA) is a sound driver that provides ALSA and OSS compatible applications with the means to perform audio playback and recording functions. ALSA has a user-space component called ALSAlib that can extend the features of audio hardware by emulating the same in software (user space), such as resampling, software mixing, snooping, and so on. The ASoC Sound driver supports stereo codec playback and capture through SSI.	Chapter 17, “Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver”	i.MX53
S/PDIF	The S/PDIF driver is designed under the Linux ALSA subsystem. It implements one playback device for Tx and one capture device for Rx.	Chapter 18, “The Sony/Philips Digital Interface (S/PDIF) Tx Driver”	i.MX53
Memory Drivers			
SPI NOR MTD	The SPI NOR MTD driver provides the support to the Atmel data Flash using the SPI interface.	Chapter 19, “SPI NOR Flash Memory Technology Device (MTD) Driver”	i.MX53
NAND MTD	The NAND MTD driver interfaces with the integrated NAND controller. It can support various file systems, such as . The driver implementation supports the lowest level operations on the external NAND Flash chip, such as block read, block write and block erase as the NAND Flash technology only supports block access. Because blocks in a NAND Flash are not guaranteed to be good, the NAND MTD driver is also able to detect bad blocks and feed that information to the upper layer to handle bad block management.	Chapter 20, “NAND Flash Memory Technology Device (MTD) Driver	i.MX53
SATA	The SATA AHCI driver is based on the LIBATA layer of the block device infrastructure of the Linux kernel	Chapter 21, “SATA Driver	i.MX53
Input Device Drivers			
Keypad	The keypad driver interfaces Linux to the keypad controller (KPP). The software operation of the keypad driver follows the Linux keyboard architecture.	Chapter 22, “Low-Level Keypad Driver”	i.MX53
Touch Screen	The touch screen driver with MC13892 ADC is designed as a standard Linux input device driver.	Chapter 5, “MC13892 Digitizer Driver	i.MX53
Networking Drivers			
FEC	The 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 adaptor 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.	Chapter 23, “Fast Ethernet Controller (FEC) Driver”	i.MX53

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

Feature	Description	Chapter Source	Applicable Platform
Bus Drivers			
MLB	MediaLB is an on-PCB or inter-chip communication bus, specifically designed to standardize a common hardware interface and software API library.	Chapter 33, "Media Local Bus Driver"	i.MX53
I^2C	The I^2C bus driver is a low-level interface that is used to interface with the I^2C bus. This driver is invoked by the I^2C chip driver; it 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 bus driver to transfer data over the I^2C bus. This bus driver supports: • Compatibility with the I^2C bus standard • Bit rates up to 400 Kbps • Standard I^2C master mode • Power management features by suspending and resuming I^2C.	Chapter 24, "Inter-IC (I2C) Driver"	i.MX53
CSPI	The low-level Configurable Serial Peripheral Interface (CSPI) driver interfaces a custom, kernel-space API to both CSPI modules. It supports the following features: • Interrupt-driven transmit/receive of SPI frames • Multi-client management • Priority management between clients • SPI device configuration per client	Chapter 21, "SPI Bus Driver"	i.MX53
MMC/SD/SDIO - eSDHC	The MMC/SD/SDIO Host driver implements the standard Linux driver interface to eSDHC.	Chapter 26, "MMC/SD/SDIO Host Driver"	i.MX53
UART Drivers			
MXC UART	The Universal Asynchronous Receiver/Transmitter (UART) driver interfaces the Linux serial driver API to all of the UART ports. A kernel configuration parameter gives the user the ability to choose the UART driver and also to choose whether the UART should be used as the system console.	Chapter 27, "Universal Asynchronous Receiver/Transmitter (UART) Driver"	i.MX53
General Drivers			
USB	The USB driver implements a standard Linux driver interface to the ARC USB-OTG controller.	Chapter 28, "ARC USB Driver"	i.MX53
FlexCAN	The FlexCAN driver is designed as a network device driver. It provides the interfaces to send and receive CAN messages. The CAN protocol was primarily designed to be used as a vehicle serial data bus, meeting the specific requirements of this field: real-time processing, reliable operation in the EMI environment of a vehicle, cost-effectiveness and required bandwidth.	Chapter 32, "FlexCAN Driver"	i.MX53
SRTC	The SRTC driver is designed to support MXC Secure RTC module to keep the time and date	Chapter 29, "Secure Real Time Clock (SRTC) Driver"	i.MX53

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

Feature	Description	Chapter Source	Applicable Platform
WatchDog	The Watchdog Timer module protects against system failures by providing an escape from unexpected hang or infinite loop situations or programming errors. This WDOG implements the following features: • Generates a reset signal if it is enabled but not serviced within a predefined time-out value • Does not generate a reset signal if it is serviced within a predefined time-out value	Chapter 26, “Watchdog (WDOG) Driver”	i.MX53
MXC PWM driver	The MXC PWM driver provides the interfaces to access MXC PWM signals	Chapter 31, “Pulse-Width Modulator (PWM) Driver”	i.MX53
Bootloaders			
uBoot	uBoot is an open source boot loader.	See uBoot User guide	i.MX53
GUI			
OProfile	OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.	Chapter 34, “OProfile”	i.MX53

Chapter 3
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 4, “Smart Direct Memory Access (SDMA) API.”

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

3.1 Interrupts

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

3.1.1 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.

3.1.2 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.

3.1.3 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)

3.1.4 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 3-1 lists the source files for interrupts.

Table 3-1. Interrupt Files

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

3.1.5 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.

3.2 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.

3.2.1 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.

3.2.2 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 3.2, “Timer.” Another function provides the time elapsed as the last timer interrupt.

3.2.3 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.

3.2.4 Timer Source Code Structure

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

3.3 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.

3.3.1 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.

3.3.2 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.

3.3.3 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.

3.3.4 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 3-2 lists the source file for the memory map.

Table 3-2. Memory Map Files

File	Description
mm.c	Memory map definition file

3.3.5 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.

3.4 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.

3.4.1 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.

3.4.2 IOMUX Software Operation

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

3.4.3 IOMUX Features

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

3.4.4 IOMUX Source Code Structure

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

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

Table 3-3. IOMUX Files

File	Description
iomux-v3.c	IOMUX function implementation
include/mach/iomux-mx53.h	Pin definitions in the iomux pins

3.4.5 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.

3.5 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.

3.5.1 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.

3.5.1.1 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.

3.5.2 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

3.5.3 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 3-4. GPIO Files

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

3.5.4 GPIO Programming Interface

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

Chapter 4
Smart Direct Memory Access (SDMA) API

4.1 Overview

The Smart Direct Memory Access (SDMA) API driver controls the SDMA hardware. It provides an API to other drivers for transferring data between MCU memory space and the peripherals. It supports the following features:

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

4.2 Hardware Operation

The SDMA controller is responsible for transferring data between the MCU 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.

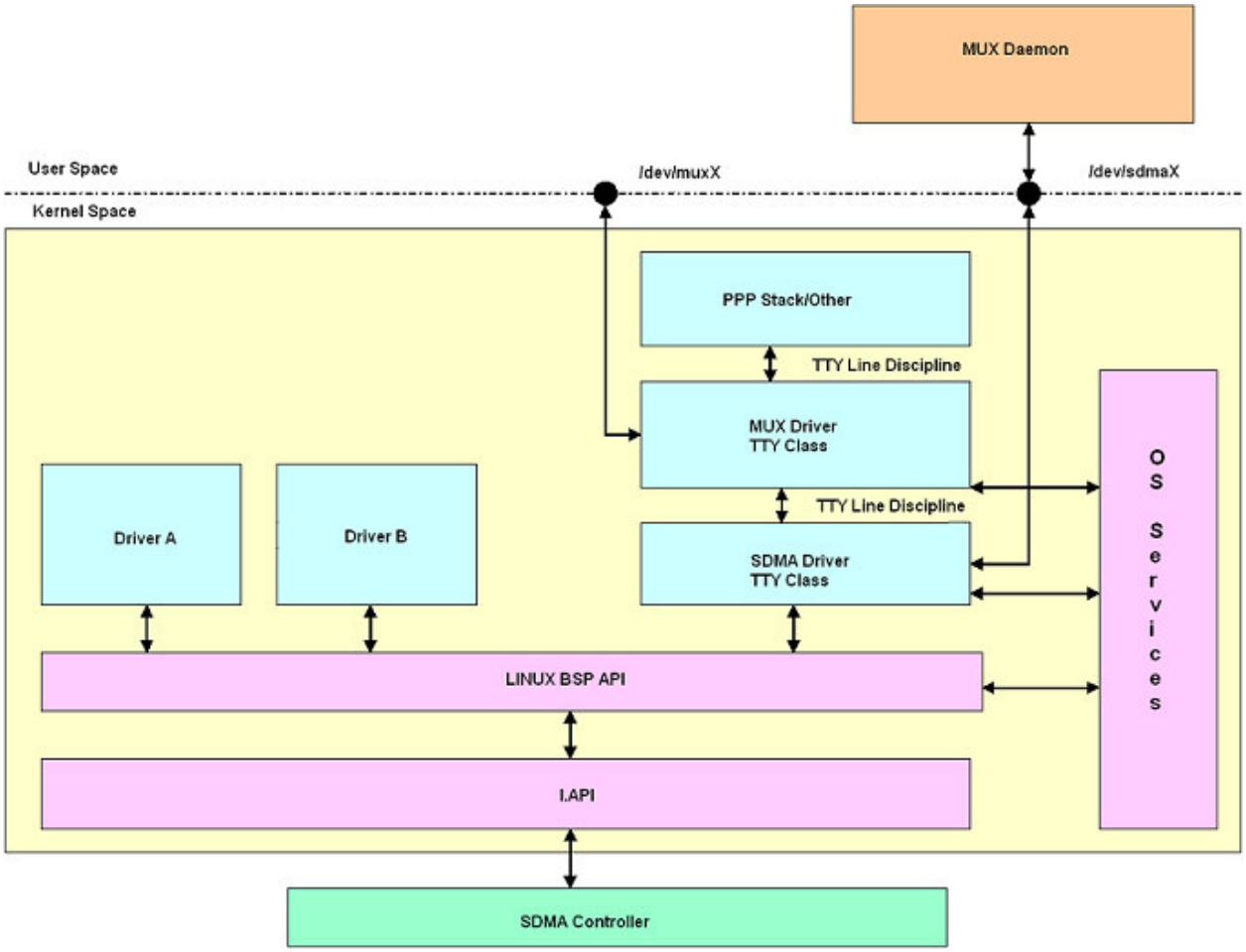
4.3 Software Operation

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 4-1):

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

Figure 4-1. 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 4-1 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 4-1. SDMA Channel Usage

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

4.4 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 4-2 shows the source files available in the directory, `/<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/sdma`.

Table 4-2. 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 4-3 shows the header files available in the directory, `/<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/.`

Table 4-3. SDMA Script Files

File	Description
sdma_script_code_mx53.h	SDMA RAM scripts

4.5 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.

4.6 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).

4.7 Usage Example

Refer to one of the drivers from Table 4-1 that uses the SDMA API driver for a usage example.

Chapter 3
MC13892 Regulator Driver

The MC13892 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.

3.1 Hardware Operation

The MC13892 provides reference and supply voltages for the application processor as well as peripheral devices. Four buck (step down) converters and two boost (step up) converters are included. 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. General Purpose Outputs (GPO) can be used for enabling external functions or supplies, thermistor biasing, and/or a muxed ADC input.

3.2 Driver Features

The MC13892 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 for GPO regulators
- Set the value for all voltage regulators
- Get the current value for all voltage regulators

3.3 Software Operation

The PMIC power management driver and the MC13892 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.

3.4 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`.

3.5 Driver Architecture

Figure 3-1 shows the basic architecture of the MC13892 regulator driver.

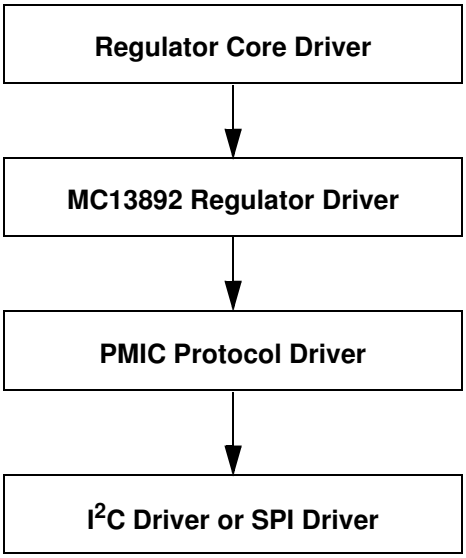


Figure 3-1. MC13892 Regulator Driver Architecture

3.6 Driver Interface Details

Access to the MC13892 regulator is provided through the API of the regulator core driver. The MC13892 regulator driver provides the following regulator controls:

- Buck switch supplies
  - Four buck switch regulators on normal mode: SWx, where x = 1–4
  - Four buck switch regulators on standby mode: SWx_ST, where x = 1–4
  - Four buck switch regulators on DVFS mode: SWx_ST, where x = 1–4
- Linear Regulators
  VVIDEO, VAUDIO, VCAM, VSD, VGEN1, VGEN2, and VGEN3
- Power gating controls
  PWGT1 and PWGT2
- General purpose outputs
  GPOx, where x = 1–4

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.

3.7 Source Code Structure

The MC13892 regulator driver is located in the regulator device driver directory:

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

Table 3-1. MC13892 Power Management Driver Files

File	Description
core.c	Linux kernel interface for regulators.
reg-mc13892.c	Implementation of the MC13892 regulator client driver

The MC13892 regulators for MX53 EVK board are registered under
`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mx53_evk_pmic_mc13892.c`.

The MC13892 regulators for MX50 ARM2 board are registered under
`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mx50_arm2_pmic_mc13892.c`.

The MC13892 regulators for MX53 EVK board are registered under
`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mx53_evk_pmic_mc13892.c`.

The MC13892 regulators for MX50 RDP board are registered under
`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mx50_rdp_pmic_mc13892.c`.

3.8 Menu Configuration Options

The following Linux kernel configurations are provided for the MC13892 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 > MC13892 Regulator Support.

Chapter 4
MC13892 RTC Driver

The Linux MC13892 RTC driver provides access to the MC13892 RTC control circuits. This device driver makes use of the MC13892 protocol driver to access the MC13892 hardware control registers. The MC13892 device is used for real-time clock control and wait alarm events.

4.1 Driver Features

The MC13892 RTC driver is a client of the MC13892 protocol driver. It provides the services for real time clock control of MC13892 components. The driver is implemented under the standard RTC class framework.

4.2 Software Operation

The MC13892 RTC driver performs operations by reconfiguring the MC13892 hardware control registers. This is done by calling protocol driver APIs with the required register settings.

4.3 Driver Implementation Details

Configuring the MC13892 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

4.3.1 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.

4.4 Source Code Structure

Table 4-1 lists the source files for MC13892 RTC driver that are available in the `<ltib_dir>/rpm/BUILD/linux/drivers/rtc` directory.

Table 4-1. MC9S08DZ60 RTC Driver Files

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

4.5 Menu Configuration Options

The following Linux kernel configurations are provided for this module. To get to the MC13892 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 MC13892 Real Time Clock.

Chapter 5
MC13892 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 monitoring
- Measurement of the voltage on the USB ID line to differentiate between mini-A and mini-B plugs

Some of these functions (for example the battery monitoring and USB ID functions) are handled separately by other PMIC device drivers.

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 also be triggered by issuing a command or by detecting the rising edge on a special signal line.

A hardware interrupt can be generated following the completion of an ADC conversion. A hardware interrupt can also be generated if the ADC conversion results are outside of previously defined high and low level thresholds. 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.

5.1 Driver Features

The PMIC Digitizer Driveris 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
- Specifies high and low level thresholds for each ADC conversion
- Starts an ADC conversion by issuing the appropriate start conversion command
- Starts an ADC conversion immediately following the rising edge of the ADTRIG input line or after a predefined delay following the rising edge
- 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 or exceeded high/low level threshold event notifications
- 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 USB ID handling. 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.

5.2 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.

5.3 Source Code Structure

Table 5-1 lists the source files for the MC13892-specific version of this driver. These are contained in the following directories:

`<ltib_dir>/rpm/BUILD/linux/drivers/mxc/pmic/mc13892/pmic_adc.c`

`<ltib_dir>/rpm/BUILD/linux/include/linux/pmic_adc.h`

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

Table 5-1. MC13892 Digitizer Driver Files

File	Description
pmic_adc.c	Implementation of the MC13892 ADC client driver
pmic_adc.h	Define names of IOCTL user space interface
mxc_ts.c	Common interface to the input driver system

5.4 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 MC13892 (MC13892) specific digitizer driver for the PMIC ADC. In menuconfig, this option is available under:

  Device Drivers > MXC Support Drivers > MXC PMIC Support > MC13892 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 6
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.

6.1 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
```

6.2 Source Code Structure

Table 6-1 shows the source files and headers available in the following directory:

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

Table 6-1. CPUFREQ Driver Files

File	Description
cpufreq.c	CPUFREQ functions

6.3 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

6.3.1 Board Configuration Options

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

Chapter 7
Low-level Power Management (PM) Driver

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

7.1 Hardware Operation

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

Table 7-1 lists the detailed clock information for the different low power modes.

Table 7-1. 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).MX53 IC spec.

7.2 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. For MX53, ARM power down is disabled to workaround stop failure.

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

7.3 Source Code Structure

Table 7-2 shows the PM driver source files. These files are available in

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

Table 7-2. 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`

7.4 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

7.5 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 8
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.

8.1 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.

8.2 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.

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
```

8.3 Source Code Structure

Table 8-1 lists the source files and headers available in the following directory:

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

Table 8-1. DVFS Driver Files

File	Description
dvfs_core.c	Linux DVFS functions

8.4 Menu Configuration Options

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

8.4.1 Board Configuration Options

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

Chapter 9
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.

9.1 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:

9.1.1 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.**

9.1.2 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.

9.1.3 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.

9.2 Source Code Structure

Table 9-1 lists the source files and headers

Table 9-1. Bus Frequency Scaling Driver Files

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

9.3 Menu Configuration Options

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

9.3.1 Board Configuration Options

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

Chapter 10
Image Processing Unit (IPU) Drivers

The image processing unit (IPU) is designed to support video and graphics processing functions and to interface with video and still image sensors and displays. The IPU driver provides a kernel-level API to manipulate logical channels. A logical channel represents a complete IPU processing flow. For example, a complete IPU processing flow (logical channel) might consist of reading a YUV buffer from memory, performing post-processing, and writing an RGB buffer to memory. A logical channel maps one to three IDMA channels and maps to either zero or one IC tasks. A logical channel can have one input, one output, and one secondary input IDMA channel. The IPU API consists of a set of common functions for all channels. Its functions are to initialize channels, set up buffers, enable and disable channels, link channels for auto frame synchronization, and set up interrupts.

Typical logical channels include:

- CSI direct to memory
- CSI to viewfinder pre-processing to memory
- Memory to viewfinder pre-processing to memory
- Memory to viewfinder rotation to memory
- CSI to encoder pre-processing to memory
- Memory to encoder pre-processing to memory
- Memory to encoder rotation to memory
- Memory to post-processing to memory
- Memory to post-processing rotation to memory
- Memory to synchronous frame buffer background
- Memory to synchronous frame buffer foreground
- Memory to synchronous frame buffer DC
- Memory to synchronous frame buffer mask

The IPU API has some additional functions that are not common across all channels, and are specific to an IPU sub-module. The types of functions for the IPU sub-modules are as follows:

- Synchronous frame buffer functions
  - — Panel interface initialization
  - — Set foreground and background plane positions
  - — Set local/global alpha and color key
  - — Set backlight level
- CSI functions
  - — Sensor interface initialization

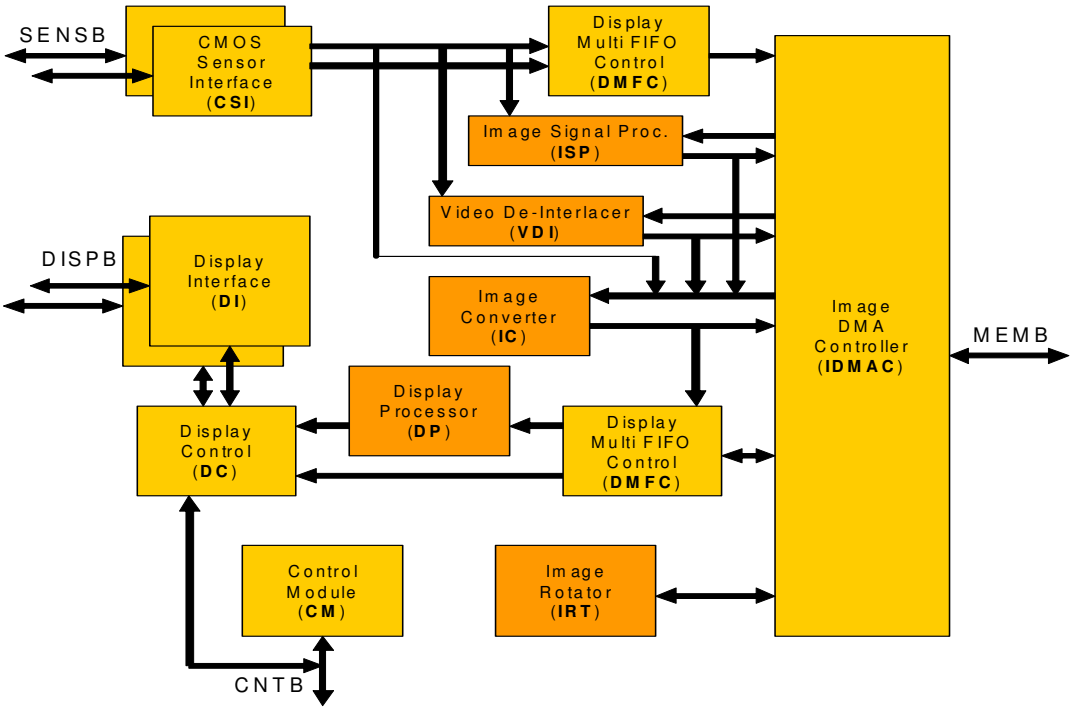
— Set sensor clock
— Set capture size

The higher level drivers are responsible for memory allocation, chaining of channels, and providing user-level API.

10.1 Hardware Operation

The detailed hardware operation of the IPU is discussed in the . shows the IPU hardware modules.

10.2 Software Operation

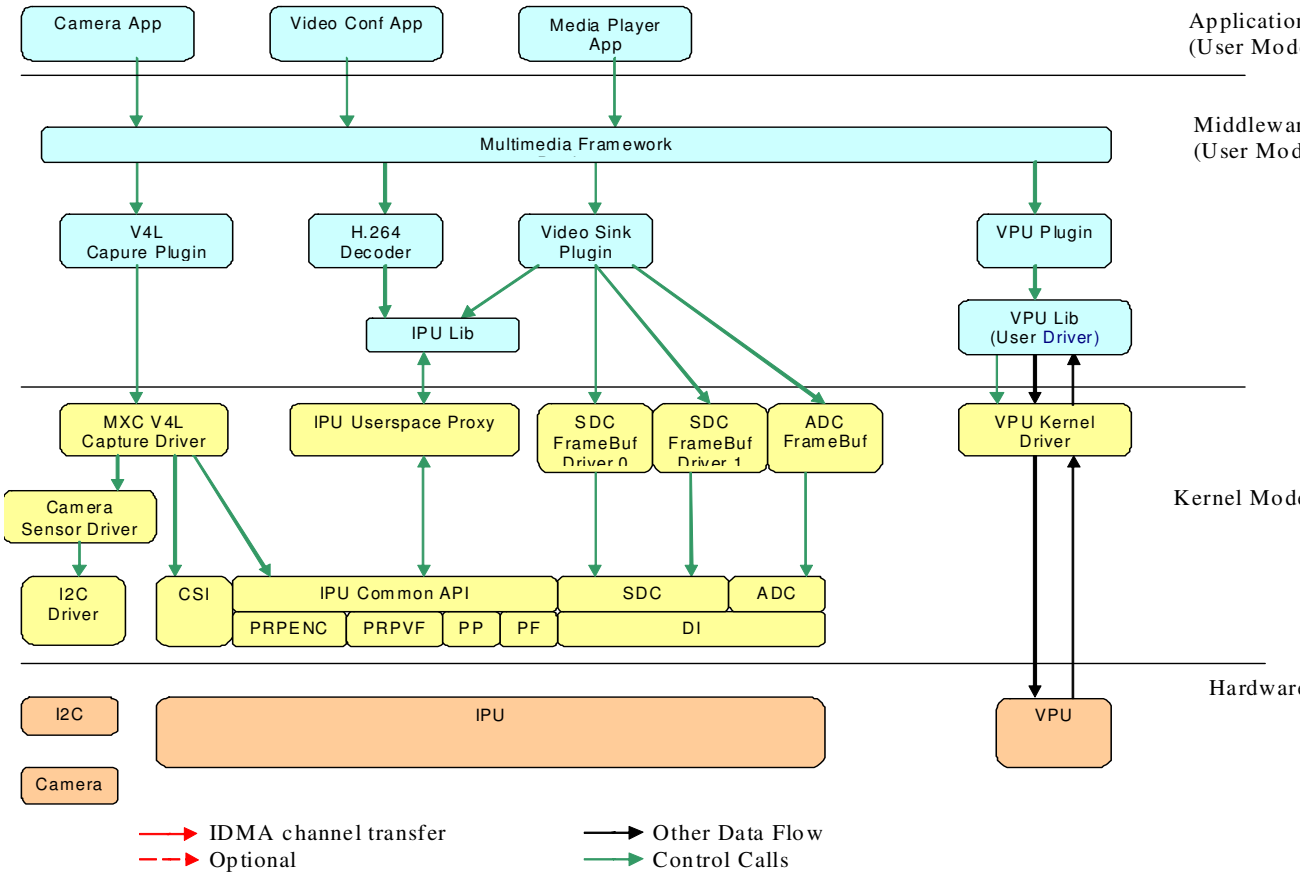


The IPU driver is a self-contained driver module in the Linux kernel. It consists of a custom kernel-level API for the following blocks:

- Synchronous frame buffer driver
- Display Interface (DI)
- Image DMA Controller (IDMAC)
- CMOS Sensor Interface (CSI)
- Image Converter (IC)

shows the interaction between the different graphics/video drivers and the IPU.

The IPU drivers are sub-divided as follows:



- Device drivers—include the frame buffer driver for the synchronous frame buffer, the frame buffer driver for the displays, V4L2 capture drivers for IPU pre-processing, and the V4L2 output driver for IPU post-processing. The frame buffer device drivers are available in the

  `<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc`

  directory of the Linux kernel. The V4L2 device drivers are available in the

  `<ltib_dir>/rpm/BUILD/linux/drivers/media/video`

  directory of the Linux kernel.
- Low-level library routines—interface to the IPU hardware registers. They take input from the high-level device drivers and communicate with the IPU hardware. The low-level libraries are available in the

  `<ltib_dir>/rpm/BUILD/linux/drivers/mxc/ipu3` directory of the Linux kernel.

10.2.1 IPU Frame Buffer Drivers 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 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. This 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.

Other than the physical memory allocation and LCD panel configuration, the common kernel video API is utilized for setting colors, palette registration, image blitting, and memory mapping. The IPU reads the raw pixel data from the frame buffer memory and sends it to the panel for display.

10.2.1.1 IPU Frame Buffer Hardware Operation

The frame buffer interacts with the IPU hardware driver module.

10.2.1.2 IPU Frame Buffer Software Operation

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.

`/dev/fb*` also interacts with several IOCTLs, which allows users to query and set information about the hardware. The color map is also handled through IOCTLs. For more information on what IOCTLs exist and which data structures they use, see `<ltib_dir>/rpm/BUILD/linux/include/linux/fb.h`. The following are a few of the IOCTLs functions:

- Request general information about the hardware, such as name, organization of the screen memory (planes, packed pixels, and so on), and address and length of the screen memory.
- Request and change variable information about the hardware, such as visible and virtual geometry, depth, color map format, timing, and so on. The driver suggests values to meet the hardware capabilities (the hardware returns EINVAL if that is not possible) if this information is changed.
- Get and set parts of the color map. Communication is 16 bits-per-pixel (values for red, green, blue, transparency) to support all existing hardware. The driver does all the calculations required to apply the options to the hardware (round to fewer bits, possibly discard transparency value).

The hardware abstraction makes the implementation of application programs easier and more portable. The only thing that must be built into the application programs is the screen organization (bitplanes or chunky pixels, and so on), because it works on the frame buffer image data directly.

The MXC frame buffer driver (`<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc/mxc_ipuv3_fb.c`) interacts closely with the generic Linux frame buffer driver (`<ltib_dir>/rpm/BUILD/linux/drivers/video/fbmem.c`).

10.2.1.3 Synchronous Frame Buffer Driver

The synchronous frame buffer screen driver implements a Linux standard frame buffer driver API for synchronous LCD panels or those without memory. The synchronous frame buffer screen driver is the top level kernel video driver that interacts with kernel and user level applications. This is enabled by selecting the Synchronous Panel Frame buffer option under the graphics support device drivers in the kernel configuration. To supplement the frame buffer driver, the kernel builder may also include support for fonts and a startup logo. This depends on the VT console for switching from serial to graphics mode.

Except for physical memory allocation and LCD panel configuration, the common kernel video API is utilized for setting colors, palette registration, image blitting and memory mapping. The IPU reads the raw pixel data from the frame buffer memory and sends it to the panel for display.

The frame buffer driver supports different panels as a kernel configuration option. Support for new panels can be added by defining new values for a structure of panel settings.

The frame buffer interacts with the IPU driver using custom APIs that allow:

- Initialization of panel interface settings
- Initialization of IPU channel settings for LCD refresh
- Changing the frame buffer address for double buffering support

The following features are supported:

- Configurable screen resolution
- Configurable RGB 16, 24 or 32 bits per pixel frame buffer
- Configurable panel interface signal timings and polarities
- Palette/color conversion management
- Power management
- LCD power off/on

User applications utilize the generic video API (the standard Linux frame buffer driver API) to perform functions with the frame buffer. These include the following:

- Obtaining screen information, such as the resolution or scan length
- Allocating user space memory using `mmap` for performing direct blitting operations

A second frame buffer driver supports a second video/graphics plane.

10.3 Source Code Structure

Table 10-1 lists the source files associated with the IPU, Sensor, V4L2 and Panel drivers. These files are available in the following directories:

```
<ltib_dir>/rpm/BUILD/linux/drivers/mxc/ipu3
<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc
<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc
<ltib_dir>/rpm/BUILD/linux/drivers/video/backlight
```

Table 10-1. IPU Driver Files

File	Description
ipu_capture.c	Asynchronous frame buffer configuration driver
ipu_common.c	Configuration functions for asynchronous and synchronous frame buffers
ipu_device.c	IPU driver device interface and fops functions
ipu_disp.c	IPU display functions
ipu_ic.c	IPU library functions
mxcfb.c	Driver for synchronous frame buffer
mxcfb_epson_vga.c	Driver for synchronous framebuffer for VGA
mxcfb_claa_wvga.c	Driver for synchronous frame buffer for WVGA
mxcfb_modedb.c	Parameter settings for Framebuffer devices

Table 10-2 lists the global header files associated with the IPU and Panel drivers. These files are available in the following directories:

```
<ltib_dir>/rpm/BUILD/linux/drivers/mxc/ipu3/
<ltib_dir>/rpm/BUILD/linux/include/linux/
<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/
```

Table 10-2. IPU Global Header Files

File	Description
ipu_param_mem.h	Helper functions for IPU parameter memory access
ipu_prv.h	Header file for Pre-processing drivers
ipu_regs.h	IPU register definitions
mxc_pf.h	Header file for Post filtering driver
mxcfb.h	Header file for the synchronous framebuffer driver

10.4 Menu Configuration Options

The following Linux kernel configuration options are provided for the IPU 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_IPU—Includes support for the Image Processing Unit. In menuconfig, this option is available under:

Device Drivers > MXC support drivers > Image Processing Unit Driver

By default, this option is Y for all architectures.

- CONFIG_MXC_CAMERA_MICRON_111—Option for both the Micron mt9v111 sensor driver and the use case driver. This option is dependent on the MXC_IPU option. In menuconfig, this option is available under:

  Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > Micron mt9v111 Camera support

  Only one sensor should be installed at a time.
- CONFIG_MXC_CAMERA_OV2640—Option for both the OV2640 sensor driver and the use case driver. This option is dependent on the MXC_IPU option. In menuconfig, this option is available under:

  Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > OmniVision ov2640 camera support

  Only one sensor should be installed at a time.
- CONFIG_MXC_CAMERA_OV3640—Option for both the OV3640 sensor driver and the use case driver. This option is dependent on the MXC_IPU option. In menuconfig, this option is available under:

  Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > OmniVision ov3640 camera support

  Only one sensor should be installed at a time.
- CONFIG_MXC_IPU_PRP_VF_SDC—Option for the IPU (here the > symbols illustrates data flow direction between HW blocks):

  CSI > IC > MEM MEM > IC (PRP VF) > MEM

  Use case driver for dumb sensor or

  CSI > IC(PRP VF) > MEM

  for smart sensors. In menuconfig, this option is available under:

  Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > Pre-Processor VF SDC library

  By default, this option is M for all.
- CONFIG_MXC_IPU_PRP_ENC—Option for the IPU:

  Use case driver for dumb sensors

  CSI > IC > MEM MEM > IC (PRP ENC) > MEM

  or for smart sensors

  CSI > IC(PRP ENC) > MEM.

  In menuconfig, this option is available under:

  Device Drivers > Multimedia Devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > Pre-processor Encoder library

  By default, this option is set to M for all.

- CONFIG_VIDEO_MXC_CAMERA—This is configuration option for V4L2 capture Driver. This option is dependent on the following expression:

  VIDEO_DEV && MXC_IPU && MXC_IPU_PRP_VF_SDC && MXC_IPU_PRP_ENC

  In menuconfig, this option is available under:

  Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera

  By default, this option is M for all.
- CONFIG_VIDEO_MXC_OUTPUT—This is configuration option for V4L2 output Driver. This option is dependent on VIDEO_DEV && MXC_IPU option. In menuconfig, this option is available under:

  Device Drivers > Multimedia devices > Video capture adapters > MXC Video for Linux Video Output

  By default, this option is Y for all.
- 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_FB_MXC_SYNC_PANEL—This is the configuration option that chooses the synchronous panel framebuffer. This option is dependent on the CONFIG_FB_MXC option. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer

  By default this option is Y for all architectures.
- CONFIG_FB_MXC_EPSON_VGA_SYNC_PANEL—This is the configuration option that chooses the Epson VGA panel. This option is dependent on CONFIG_FB_MXC_SYNC_PANEL option. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > Epson VGA Panel
- CONFIG_FB_MXC_CLAA_WVGA_SYNC_PANEL —This is the configuration option that chooses the CLAA WVGA panel. This option is dependent on CONFIG_FB_MXC_SYNC_PANEL option. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > CLAA WVGA Panel.
- CONFIG_FB_MXC_TVOUT_CH7024 —This configuration option selects the CH7024 TVOUT encoder. This option is dependent on the CONFIG_FB_MXC_SYNC_PANEL option. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > CH7024 TV Out Encoder

- CONFIG_FB_MXC_TVOUT —This configuration option selects the FS453 TVOUT encoder. This option is dependent on CONFIG_FB_MXC_SYNC_PANEL option. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > FS453 TV Out Encoder

10.5 Programming Interface

For more information, see the *API Documents* for the programming interface.

Chapter 11
Video for Linux Two (V4L2) Driver

The Video for Linux Two (V4L2) drivers are plug-ins to the V4L2 framework that enable support for camera and preprocessing functions, as well as video and post-processing functions. The V4L2 camera driver implements support for all camera related functions. The V4l2 capture device takes incoming video images, either from a camera or a stream, and manipulates them. The output device takes video and manipulates it, then sends it to a display or similar device. The V4L2 Linux standard API specification is available at http://v4l2spec.bytesex.org/spec/.

The features supported by the V4L2 driver are as follows:

- Direct preview and output to SDC foreground overlay plane (with no processor intervention and synchronized to LCD refresh)
- Direct preview to graphics frame buffer (with no processor intervention, but not synchronized to LCD refresh)
- Color keying or alpha blending of frame buffer and overlay planes
- Simultaneous preview and capture
- Streaming (queued) capture from IPU encoding channel
- Direct (raw Bayer) still capture (sensor dependent)
- Programmable pixel format, size, frame rate for preview and capture
- Programmable rotation and flipping using custom API
- RGB 16-bit, 24-bit, and 32-bit preview formats
- Raw Bayer (still only, sensor dependent), RGB 16, 24, and 32-bit, YUV 4:2:0 and 4:2:2 planar, YUV 4:2:2 interleaved, and JPEG formats for capture
- Control of sensor properties including exposure, white-balance, brightness, contrast, and so on
- Plug-in of different sensor drivers
- Linking post-processing resize and CSC, rotation, and display IPU channels with no ARM processing of intermediate steps
- Streaming (queued) input buffer
- Double buffering of overlay and intermediate (rotation) buffers
- Configurable 3+ buffering of input buffers
- Programmable input and output pixel format and size
- Programmable scaling and frame rate
- RGB 16, 24, and 32-bit, YUV 4:2:0 and 4:2:2 planar, and YUV 4:2:2 interleaved input formats
- TV output

The driver implements the standard V4L2 API for capture, output, and overlay devices. The command `modprobe mxc_v4l2_capture` must be run before using these functions.

11.1 V4L2 Capture Device

The V4L2 capture device includes two interfaces:

- Capture interface—uses IPU pre-processing ENC channels to record the YCrCb video stream
- Overlay interface—uses the IPU pre-processing VF channels to display the preview video to the SDC foreground panel without ARM processor interaction.

V4L2 capture support can be selected during kernel configuration. The driver includes two layers. The top layer is the common Video for Linux driver, which contains chain buffer management, stream API and other `ioctl` interfaces. The files for this device are located in `<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture/`.

The V4L2 capture device driver is in the `mxc_v4l2_capture.c` file. The lowest layer is in the `ipu_prp_enc.c` file.

This code (`ipu_prp_enc.c`) interfaces with the IPU ENC hardware, `ipu_prp_vf_sdc_bg.c` interfaces with the IPU VF hardware, and `ipu_still.c` interfaces with the IPU CSI hardware. Sensor frame rate control is handled by VIDIOC_S_PARM `ioctl`. Before the frame rate is set, the sensor turns on the AE and AWB turn on. The frame rate may change depending on light sensor samples.

Drivers for specific cameras can be found in `<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture/`

11.1.1 V4L2 Capture IOCTLs

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

- VIDIOC_QUERYCAP
- VIDIOC_G_FMT
- VIDIOC_S_FMT
- VIDIOC_REQBUFS
- VIDIOC_QUERYBUF
- VIDIOC_QBUF
- VIDIOC_DQBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF
- VIDIOC_OVERLAY
- VIDIOC_G_FBUF
- VIDIOC_S_FBUF
- VIDIOC_G_CTRL
- VIDIOC_S_CTRL

- VIDIOC_CROPCAP
- VIDIOC_G_CROP
- VIDIOC_S_CROP
- VIDIOC_S_PARM
- VIDIOC_G_PARM
- VIDIOC_ENUMSTD
- VIDIOC_G_STD
- VIDIOC_S_STD
- VIDIOC_ENUMOUTPUT
- VIDIOC_G_OUTPUT
- VIDIOC_S_OUTPUT

V4L2 control code has been extended to provide support for rotation. The ID is V4L2_CID_PRIVATE_BASE. Supported values include:

- 0—Normal operation
- 1—Vertical flip
- 2—Horizontal flip
- 3—180° rotation
- 4—90° rotation clockwise
- 5—90° rotation clockwise and vertical flip
- 6—90° rotation clockwise and horizontal flip
- 7—90° rotation counter-clockwise

Figure 11-1 shows a block diagram of V4L2 Capture API interaction.

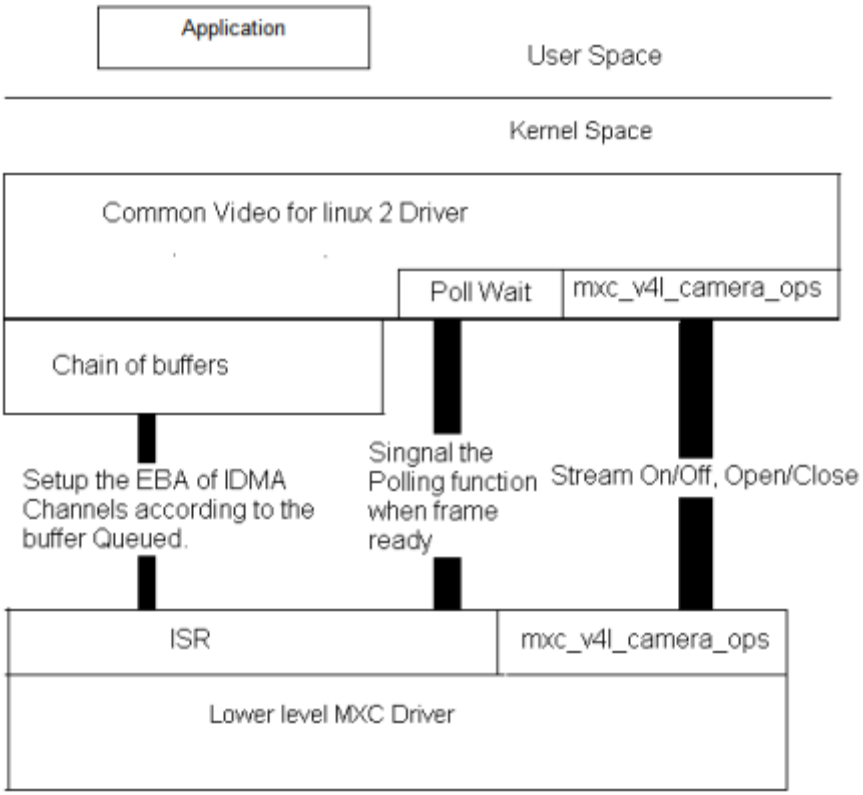


Figure 11-1. Video4Linux2 Capture API Interaction

11.1.2 Use of the V4L2 Capture APIs

This section describes a sample V4L2 capture process. The application completes the following steps:

1. Sets the capture pixel format and size by IOCTL VIDIOC_S_FMT.
2. Sets the control information by IOCTL VIDIOC_S_CTRL for rotation usage.
3. Requests a buffer using IOCTL VIDIOC_REQBUFS. The common V4L2 driver creates a chain of buffers (currently the maximum number of frames is 3).
4. Memory maps the buffer to its user space.
5. Queues buffers using the IOCTL command VIDIOC_QBUF.
6. Starts the stream using the IOCTL VIDIOC_STREAMON. This IOCTL enables the IPU tasks and the IDMA channels. When the processing is completed for a frame, the driver switches to the buffer that is queued for the next frame. The driver also signals the semaphore to indicate that a buffer is ready.
7. Takes the buffer from the queue using the IOCTL VIDIOC_DQBUF. This IOCTL blocks until it has been signaled by the ISR driver.
8. Stores the buffer to a YCrCb file.
9. Replaces the buffer in the queue of the V4L2 driver by executing VIDIOC_QBUF again.

For the V4L2 still image capture process, the application completes the following steps:

1. Sets the capture pixel format and size by executing the IOCTL VIDIOC_S_FMT.
2. Reads one frame still image with YUV422.

FOr the V4L2 overlay support use case, the application completes the following steps:

1. Sets the overlay window by IOCTL VIDIOC_S_FMT.
2. Turns on overlay task by IOCTL VIDIOC_OVERLAY.
3. Turns off overlay task by IOCTL VIDIOC_OVERLAY.

11.2 V4L2 Output Device

The V4L2 output driver uses the IPU post-processing functions for video output. The driver implements the standard V4L2 API for output devices. V4L2 output device support can be selected during kernel configuration. The driver is available at
`<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/output/mxc_v4l2_output.c`.

11.2.1 V4L2 Output IOCTLs

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

- VIDIOC_QUERYCAP
- VIDIOC_REQBUFS
- VIDIOC_G_FMT
- VIDIOC_S_FMT
- VIDIOC_QUERYBUF
- VIDIOC_QBUF
- VIDIOC_DQBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF
- VIDIOC_G_CTRL
- VIDIOC_S_CTRL
- VIDIOC_CROPCAP
- VIDIOC_G_CROP
- VIDIOC_S_CROP
- VIDIOC_S_PARM
- VIDIOC_G_PARM

The V4L2 control code has been extended to provide support for rotation. For this use, the ID is V4L2_CID_PRIVATE_BASE. Supported values include the following:

- 0—Normal operation
- 1—Vertical flip
- 2—Horizontal flip

- 3—Horizontal and vertical flip
- 4—90° rotation
- 5—90° rotation and vertical flip
- 6—90° rotation and horizontal flip
- 7—90° rotation with horizontal and vertical flip

11.2.2 Use of the V4L2 Output APIs

This section describes a sample V4L2 capture process that uses the V4L2 output APIs. The application completes the following steps:

1. Sets the capture pixel format and size using IOCTL VIDIOC_S_FMT.
2. Sets the control information using IOCTL VIDIOC_S_CTRL, for rotation.
3. Requests a buffer using IOCTL VIDIOC_REQBUFS. The common V4L2 driver creates a chain of buffers (currently the maximum number of frames is 3).
4. Memory maps the buffer to its user space.
5. Executes the IOCTL VIDIOC_DQBUF.
6. Passes the data that requires post-processing to the buffer.
7. Queues the buffer using the IOCTL command VIDIOC_QBUF.
8. Starts the stream by executing IOCTL VIDIOC_STREAMON.
9. VIDIOC_STREAMON and VIDIOC_OVERLAY cannot be enabled simultaneously.

11.3 Source Code Structure

Table 11-1 lists the source and header files associated with the V4L2 drivers. These files are available in the following directory:

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

Table 11-1. V2L2 Driver Files

File	Description
capture/mxc_v4l2_capture.c	V4L2 capture device driver
output/mxc_v4l2_output.c	V4L2 output device driver
capture/mxc_v4l2_capture.h	Header file for V4L2 capture device driver
output/mxc_v4l2_output.h	Header file for V4L2 output device driver
capture/ipu_prp_enc.c	Pre-processing encoder driver
capture/ipu_prp_vf_adc.c	Pre-processing view finder (asynchronous) driver
capture/ipu_prp_vf_sdc.c	Pre-processing view finder (synchronous foreground) driver
capture/ipu_prp_vf_sdc_bg.c	Pre-processing view finder (synchronous background) driver
capture/ipu_still.c	Pre-processing still image capture driver

Drivers for specific cameras can be found in
`<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture/`

11.4 Menu Configuration Options

The Linux kernel configuration options are provided in the chapter on the IPU module. See Section 10.4, "Menu Configuration Options."

11.5 V4L2 Programming Interface

For more information, see the *V4L2 Specification* and the *API Documents* for the programming interface. The API Specification is available at http://v4l2spec.bytesex.org/spec/.

Chapter 12
LVDS Display Bridge(LDB) Driver

This section describes the LVDS Display Bridge(LDB) driver which controls LDB module to connect with external display devices with LVDS interface.

12.1 Hardware Operation

The purpose of the LDB is to support flow of synchronous RGB data from the IPU to external display devices through LVDS interface. This support covers all aspects of these activities:

1) Connectivity to relevant devices - Displays with LVDS receivers.

2) Arranging the data as required by the external display receiver and by LVDS display standards.

3) Synchronization and control capabilities.

For the detailed information about LDB, see the .

12.2 Software Operation

The LDB driver will be functional if the driver is built-in and the user add ‘ldb’ option to boot-up command line. Adding more options with ‘ldb=’ prefixed can configure the LDB when the device is probed, including the LVDS channel mapping mode and bit mapping mode of LDB.

When the LDB device is probed properly, the driver will configure the LDB’s reference resistor mode and LDB’s regulator by using platform data information. The LDB driver probe function will also try to match video modes for external display devices with LVDS interface. The display signals’ polarities control bits of LDB will be set according to the matched video modes, and, LVDS channel mapping mode and bit mapping mode of LDB will be set according to the bootup LDB option chosen by user if there is any, otherwise, an appropriate LDB setting will be chosen by the driver if the video mode can be found in local video mode database. If no video mode is matched, nothing will be done in probe function and the user can set up the LDB later by using ioctrls. LDB will be fully enabled in probe function if the driver finds that one display device with LVDS interface is the primary display device.

The steps the driver takes to enable a LVDS channel are:

1. Set ldb_di_clk’s parent clk and the parent clk’s rate.
2. Set ldb_di_clk’s rate.
3. Enable both ldb_di_clk and its parent clk.
4. Set the LDB in a proper mode, including display signals’ polarities, LVDS channel mapping mode, bit mapping mode, reference resistor mode.
5. Enable related LVDS channels.

The LDB driver also defines several ioctrls. Each ioctrl controls a LDB unit setting so that users may set LDB in various modes as they want.

12.3 Source Code Structure

The source code is available in

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

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 as build-in status to enable this module:

> Device Drivers -> Graphics support -> MXC Framebufer support -> Synchronous Panel Framebuffer -> MXC LDB

12.5 Programming Interface

The APIs in the mxc_ldb_ioctl() function controls every other LDB unit setting. The user may call these APIs to set LDB modes or enable/disable LDB.

Chapter 13
i.MX5 Dual Display

This section describes how to setup dual-display on i.MX53 EVK platform.

13.1 Hardware Operation

i.MX53 multimedia application processes incorporate the Image Processing Unit(IPUv3) hardware image processing accelerator. There are two Display Interfaces(DI) within IPUv3, which provide connection to external display devices and related devices. The external display devices can be a LCD display panel, which connects with DI directly. The related devices may be embedded in chip or integrated on EVK boards. i.MX53 chips embed the LVDS Display Birdge(LDB) module so that external display devices with LVDS interfaces can be connected with the chips directly. TVE is not supported in i.MX53 chips now. There is a connector on i.NX53 EVK platform which leads the legacy parallel signals of DI0 out so that a LCD display panel can be connected to it directly or a connected DVI convertor can provide an interface for a DVI monitor.

As there are two DIs within IPUv3, we can support dual-display feature, i.e., each of the two DIs can support an externel display device simultanously. As long as the hardware bandwidth is not exceeded, MX5 EVK platform can drive every possible dual-display feature provided the board design.

Table 8.1 shows all the external display devices can be connected with i.MX51 EVK platform and i.MX53 EVK platform:

Table 13-1.

EVK Platform	DI Number	External display device
i.MX53	0	1) DVI connector 2) CLAA WVGA display panel 3) LVDS display panel(driven by LDB)
	1	1) LVDS display panel(driven by LDB)

For the detailed information about the external display devices on MX5 EVK platform, see the relevant EVK board schematics.

13.2 Software Operation

The user should setup a correct bootup command line if he or she wants to enable dual-display feature. The user may follow these steps to set the bootup command line for display related options:

1) Add ‘tve’ or ‘ldb’ to bootup command line if the use case involves TVE or LDB, otherwise, the options should not be added.

2) Add ‘di1_primary’ to bootup command line if the device connected with DI1 is the primary device, i.e., /dev/fb0 will be mapped to this device after the system boots up. If the device connected with DI0 is the primary device, no specific option is needed.

3) For each of the devices connected with DI, provide a specific video mode in bootup command line in this format: video=mxcdi*x*fb:*DI_pixel_format*, *video_mode*,bpp=*bits_per_pixel_of_frame_buffer*.

The ‘x’ stands for DI number.

The ‘DI_pixel_format’ stands for the output pixel format of the related DI. Usually, ‘RGB565’ is used for CLAA WVGA LCD display panel, ‘RGB24’ is used for DVI monitor and VGA and ‘YUV444’ is used for TVout.

The ‘bits_per_pixel_of_frame_buffer’ stands for the pixel format of the related framebuffer.

The ‘video_mode’ can be found here:

1) DVI connector and VGA: The video mode is in this format:

```
<xres>x<yres>[M][-<bpp>][@<refresh>
```

with <xres>, <yres>, <bpp> and <refresh> decimal numbers and <name> a string. If 'M' is present after yres (and before refresh/bpp if present), the framebuffer driver will compute the timings using VESA(tm) Coordinated Video Timings (CVT).

Note, if the dislplay resolution is 720P, then ‘720P60’ should be used as ‘video_mode’, and if the display resolution is UXGA(only supported on i.MX53 EVK platform), then ‘UXGA’ should be used as ‘video_mode’.

2) LVDS display panel: Use ‘XGA’ for XGA LVDS display panel and use ‘1080P60’ for 1080P LVDS display panel.

3) CLAA WVGA LCD display panel: Use ‘CLAA-WVGA’.

4) TV: Use ‘720P60’ for 720P TVout, use ‘TV-PAL’ for PAL TVout and use ‘TV-NTSC’ for NTSC TVout.

As the primary display device will be unblanked automatically after the system boots up but the secondary is still blank, the user needs to unblank the secondary by himself or herself either with framebuffer ioctrls or command line. Here is the command line the user may use on PDK to unblank the secondary display device in an ordinary case:

```
echo 0 > /sys/class/graphics/fb1/blank
```

The user may also swith the primary display device and secondary display device by command lines on PDK. For example, fb0 is the primary device’s framebuffer and fb1 is the secondary device’s framebuffer. To switch the primary display device and secondary display device, the user may use these command lines:

```
1)echo 1 > /sys/class/graphics/fb0/blank
2)echo 1 > /sys/class/graphics/fb1/blank
3)echo 1 > /sys/class/graphics/fb2/blank
4)echo 1-layer-fb > /sys/class/graphics/fb0/fsl_disp_property
```

```
5)echo 0 > /sys/class/graphics/fb0/blank
6)echo 0 > /sys/class/graphics/fb1/blank
```

To switch them back, ther user may use these command lines:

```
1)echo 1 > /sys/class/graphics/fb0/blank
2)echo 1 > /sys/class/graphics/fb1/blank
3)echo 1 > /sys/class/graphics/fb2/blank
4)echo 1-layer-fb > /sys/class/graphics/fb1/fsl_disp_property
5)echo 0 > /sys/class/graphics/fb0/blank
6)echo 0 > /sys/class/graphics/fb1/blank
```

13.3 Examples

Examples for i.MX53 EVK platform:

1) DI0:CLAA-WVGA LCD display panel, DI1:XGA LVDS display panel(primary)

```
video=mxcdi0fb:RGB565,CLAA-WVGA video=mxcdi1fb:RGB24,XGA di1_primary ldb
```

2) DI0: XGA DVI monitor(primary) DI1:XGA LVDS display panel

```
video=mxcdi0fb:RGB24,1024x768M-16@60 video=mxcdi1fb:RGB24,XGA ldb
```

Chapter 14
Video Processing Unit (VPU) Driver

14.1 Hardware Operation

The VPU hardware performs all of the codec computation and most of the bitstream parsing/packeting. Therefore, the software takes advantage of less control and effort to implement a complex and efficient multimedia codec system.

The VPU hardware data flow is shown in the MPEG4 decoder example in Figure 14-1.

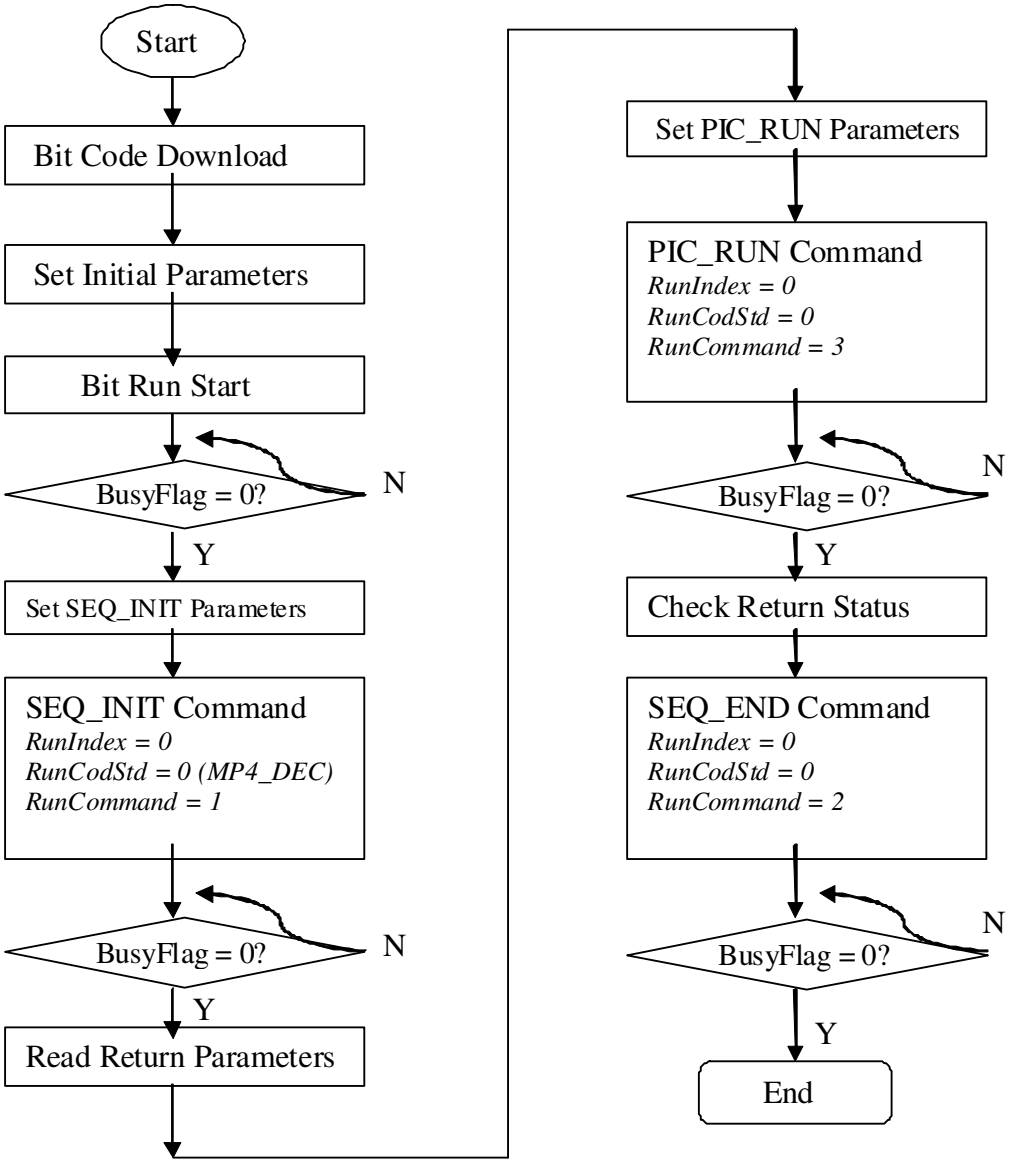


Figure 14-1. VPU Hardware Data Flow

14.2 Software Operation

The VPU software can be divided into two parts: the kernel driver and the user-space library as well as the application in user space. The kernel driver takes responsibility for system control and reserving resources (memory/IRQ). It provides an IOCTL interface for the application layer in user-space as a path to access system resources. The application in user-space calls related IOCTLs and codec library functions to implement a complex codec system.

The VPU kernel driver include the following functions:

- Module initialization—Initializes the module with the device specific structure
- Device initialization—Initializes the VPU clock and hardware, and request the IRQ
- Interrupt servicing routine—Supports events that one frame has been finished
- File operation routines— Provides the following interfaces to user space
  - File open
  - File release
  - File synchronization
  - File IOCTL to provide interface for memory allocating and releasing
  - Memory map for register and memory accessing in user space
- Device Shutdown—Shutdowns the VPU clock and hardware, and release the IRQ

The VPU user space driver has the following functions:

- Codec lib
  - Downloads executable bitcode for hardware
  - Initializes codec system
  - Sets codec system configuration
  - Controls codec system by command
  - Reports codec status and result
- System I/O operation
  - Requests and frees memory
  - Maps and unmaps memory/register to user space
  - Device management

14.3 Source Code Structure

Table 14-1 lists the kernel space source files available in the following directories:

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

Table 14-1. VPU Driver Files

File	Description
mxc_vpu.h	Header file defining IOCTLs and memory structures
mxc_vpu.c	Device management and file operation interface implementation

Table 14-2 lists the user-space library source files available in the `<ltib_dir>/rpm/BUILD/imx-lib-10.10.01/vpu` directory:

Table 14-2. VPU Library Files

File	Description
vpu_io.c	Interfaces with the kernel driver for opening the VPU device and allocating memory
vpu_io.h	Header file for IOCTLs
vpu_lib.c	Core codec implementation in user space
vpu_lib.h	Header file of the codec
vpu_reg.h	Register definition of VPU
vpu_util.c	File implementing common utilities used by the codec
vpu_util.h	Header file

Table 14-3 lists the firmware files available in the following directories:

`<ltib_dir>/rpm/BUILD/firmware-imx-10.10.01/lib/firmware/vpu/` directory

Table 14-3. VPU firmware Files

File	Description
vpu_fw_xxx.bin	VPU firmware

NOTE

To get the to files in Table 14-2, run the command: `./ltib -m prep -p imx-lib` in the console

14.4 Menu Configuration Options

To get to the VPU 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 VPU driver:

- CONFIG_MXC_VPU—Provided for the VPU driver. In menuconfig, this option is available under

  Device Drivers > MXC support drivers > MXC VPU (Video Processing Unit) support

14.5 Programming Interface

There is only a user-space programming interface for the VPU module. A user in the application layer cannot access the kernel driver interface directly. The VPU library access the kernel driver interface for users.

The codec library APIs are listed below:

```
RetCode vpu_EncOpen(EncHandle* pHandle, EncOpenParam* pop);
RetCode vpu_EncClose(EncHandle encHandle);
RetCode vpu_EncGetInitialInfo(EncHandle encHandle, EncInitialInfo* initialInfo);
RetCode vpu_EncRegisterFrameBuffer(EncHandle encHandle, FrameBuffer* pBuffer, int num,
```

```
                                    int stride);
RetCode vpu_EncGetBitstreamBuffer(EncHandle handle, PhysicalAddress* prdPrt,
                                    PhysicalAddress* pwrPtr, Uint32* size);
RetCode vpu_EncUpdateBitstreamBuffer(EncHandle handle, Uint32 size);
RetCode vpu_EncStartOneFrame(EncHandle encHandle, EncParam* pParam);
RetCode vpu_EncGetOutputInfo(EncHandle encHandle, EncOutputInfo* info);
RetCode vpu_EncGiveCommand (EncHandle pHandle, CodecCommand cmd, void* pParam);
RetCode vpu_DecOpen(DecHandle* pHandle, DecOpenParam* pop);
RetCode vpu_DecClose(DecHandle decHandle);
RetCode vpu_DecGetBitstreamBuffer(DecHandle pHandle, PhysicalAddress* pRdptr,
                                    PhysicalAddress* pWrptr, Uint32* size);
RetCode vpu_DecUpdateBitstreamBuffer(DecHandle decHandle, Uint32 size);
RetCode vpu_DecSetEscSeqInit(DecHandle pHandle, int escape);
RetCode vpu_DecGetInitialInfo(DecHandle decHandle, DecInitialInfo* info);
RetCode vpu_DecRegisterFrameBuffer(DecHandle decHandle, FrameBuffer* pBuffer, int num,
                                    int stride, DecBufInfo* pBufInfo);
RetCode vpu_DecStartOneFrame(DecHandle handle, DecParam* param);
RetCode vpu_DecGetOutputInfo(DecHandle decHandle, DecOutputInfo* info);
RetCode vpu_DecBitBufferFlush(DecHandle handle);
RetCode vpu_DecClrDispFlag(DecHandle handle, int index);
RetCode vpu_DecGiveCommand(DecHandle pHandle, CodecCommand cmd, void* pParam);
int vpu_WaitForInt(int timeout_in_ms);
RetCode vpu_SWReset(DecHandle handle, int index);
```

System I/O operations are listed below:

```
int IOSystemInit(void);
int IOSystemShutdown(void);
int IOGetPhyMem(vpu_mem_desc* buff);
int IOFreePhyMem(vpu_mem_desc* buff);
int IOGetVirtMem (vpu_mem_desc* buff);
int IOFreeVirtMem(vpu_mem_desc* buff);
```

14.6 Defining an Application

The most important definition for an application is the codec memory descriptor. It is used for `request`, `free`, `mmap and munmap` memory as follows:

```
typedef struct vpu_mem_desc
{
        int size;                   /*request memory size*/
        unsigned long phy_addr;     /*physical memory get from system*/
        unsigned long cpu_addr;     /*address for system usage while freeing, user doesn't need
                                            to handle or use it*/
        unsigned long virt_uaddr; /*virtual user space address*/
} vpu_mem_desc;
```

Chapter 15
Graphics Processing Unit (GPU)

The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D/3D graphics applications. The GPU3D (3D graphics processing unit) is based on the AMD Z430 core, which is an embedded engine that accelerates user level graphics APIs (Application Programming Interface) such as OpenGL ES 1.1 and 2.0. The GPU2D (2D graphics processing unit) is based on the AMD Z160 core, which is an embedded 2D and vector graphics accelerator targeting the OpenVG 1.1 graphics API and feature set. The GPU driver kernel module source is in kernel source tree, but the libs are delivered as binary only.

15.1 Driver Features

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

- EGL (EGL™ is an interface between Khronos rendering APIs such as OpenGL ES or OpenVG and the underlying native platform window system) 1.3 API defined by Khronos Group
- OpenGL ES (OpenGL® ES is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems) 1.1 API defined by Khronos Group
- OpenGL ES 2.0 API defined by Khronos Group
- OpenVG (OpenVG™ is a royalty-free, cross-platform API that provides a low-level hardware acceleration interface for vector graphics libraries such as Flash and SVG) 1.1 API defined by Khronos Group

15.2 Hardware Operation

Refer to the GPU chapter in the *MCIMX51 Multimedia Applications Processor Reference Manual* (MCIMX51RM) for detailed hardware operation and programming information.

15.3 Software Operation

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

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

15.4 Source Code Structure

Table 15-1 lists GPU driver kernel module source structure:

```
<ltib_dir>/rpm/BUILD/linux/drivers/mxc/amd-gpu
```

Table 15-1. GPU Driver Files

File	Description
Kconfig Makefile	kernel configure file and makefile
include	header files
common	common and core code
os	os specific code
platform	platform specific code

15.5 API References

Refer to the following web sites for detailed specifications:

- OpenGL ES 1.1 and 2.0 API: http://www.khronos.org/opengles/
- EGL 1.3 API: http://www.khronos.org/egl/
- OpenVG 1.1 API: http://www.khronos.org/openvg/

15.6 Menu Configuration Options

The following Linux kernel configurations are provided for GPU driver:

- 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.

To get to the 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 kernel modules, libraries, and test code built from the GPU for framebuffer
- Package list > amd-gpu-x11-bin-mx51

  This package provides proprietary binary kernel modules, libraries, and test code built from the GPU for X-Window

Chapter 16
TV Decoder (TV-In) Driver

The ADV7180 is a versatile one-chip multi-format video decoder that automatically detects and converts PAL, NTSC, and SECAM standards in the form of composite, S-video, and component video into a digital ITU-R BT.656 format.

The TV-In driver is located under the Linux V4L2 architecture. It is based on the V4L2 capture interface. Applications cannot use the TV-In driver directly; instead, the applications use the V4L2 capture driver to open and close the TV-In for starting the video preview.

16.1 Hardware Operation

The ADV7180 is programmed through a 2-wire, serial, bidirectional port (I^2C compatible). It works as an I^2C client and the IPU does not control the TV-In chip. The function has to be performed by the MCU through the I^2C interface and GPIO pins connected to the TV-In decoder chip. The ADV7180 output digital signal format is ITU-R BT.656. This video protocol uses an embedded timing syntax to replace the VSYNC and HSYNC signals.

Refer to the analog device ADV7180 datasheet to get more information for the video decoder. Refer to the datasheet of the platform to get more information of CSI and IPU.

16.2 Software Operation

The TV-In driver implements the V4L2 capture interface and applications use V4L2 capture interface to operate the TV-In chip.

16.3 Source Code Structure Configuration

Table 16-1 describes the source files associated with the TVIN driver, which are available in the directory `<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture`.

Table 16-1. TV-In Driver Source File

File	Description
adv7180.c	Source file for TV-In driver

16.4 Linux Menu Configuration Options

The Linux kernel provides the configuration option for the TV-In driver. In the `menuconfig`, this option is available under

Device Drivers > Multimedia device > Video Capture Adapters > MXC Camera/V4L2 PRP Features support.

This option is dependent on the CONFIG_MXC_TVIN_ADV7180 option. By default, this option is M.

NOTE

The TV-In and Camera share the same CSI hardware interface; therefore, the TV-In and Camera modules cannot be built-in into the kernel at the same time.

Chapter 17
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.

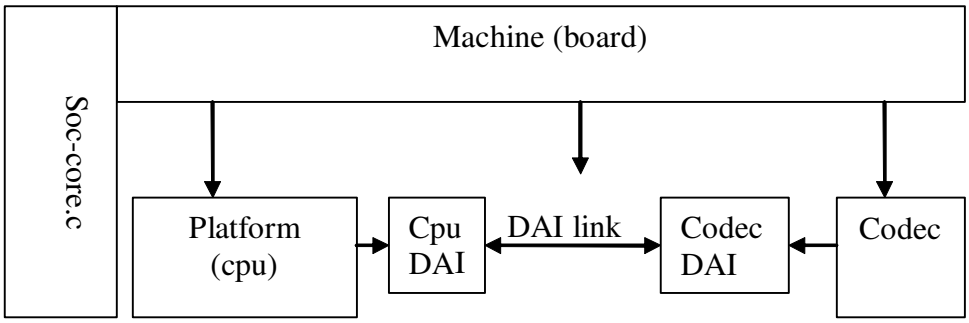


Figure 17-1. ALSA SoC Software Architecture

The ALSA SoC driver has the following components as shown in Figure 17-1:

- 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.

17.1 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

The Stereo Codec and multiple-channel codec are supported on i.MX53 platform.

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

17.1.1 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

17.1.2 Multi-channel Codec Feature

- Sample rates for playback and capture are 44.1kHz, 88.2kHz and 176.4kHz, as there is only a 22.579MHz Osc on the board. If playback the multiple of 48kHz bit streams, the ALSA plugin is needed to convert the sample rate.
- Channels:
  - Playback: supports 6 channels. (5.1)
  - Capture: supports 4 channels.
- Audio formats:
  - Playback:
    - SNDRV_PCM_FMTBIT_S16_LE
    - SNDRV_PCM_FMTBIT_S24_LE
  - Capture
    - SNDRV_PCM_FMTBIT_S16_LE

– SNDRV_PCM_FMTBIT_S24_LE

17.1.3 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
      card 1: imx3stack_1 [imx-3stack], device 0: cs42888 cs42888-0 []
       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
      card 1: imx3stack_1 [imx-3stack], device 0: cs42888 cs42888-0 []
       Subdevices: 1/1
      Subdevice #0: subdevice #0
```

17.2 ASoC Driver Source Architecture

As shown in Figure 17-1, `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.

The multi-channel codec is connected to the CPU through the ESAI interface. `imx-esai.c` registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip ESAI interface. `cs42888.c` registers the multi-channel codec and hifi DAI drivers. The direct hardware operations on the multi-channel codec are in `cs42888.c`. `imx-3stack-cs42888.c` is the machine layer code which creates the driver device and registers the stereo sound card.

Figure 17-2 shows the ALSA SoC source file relationship.

Figure 17-2. ALSA Soc Source FIle Relationship

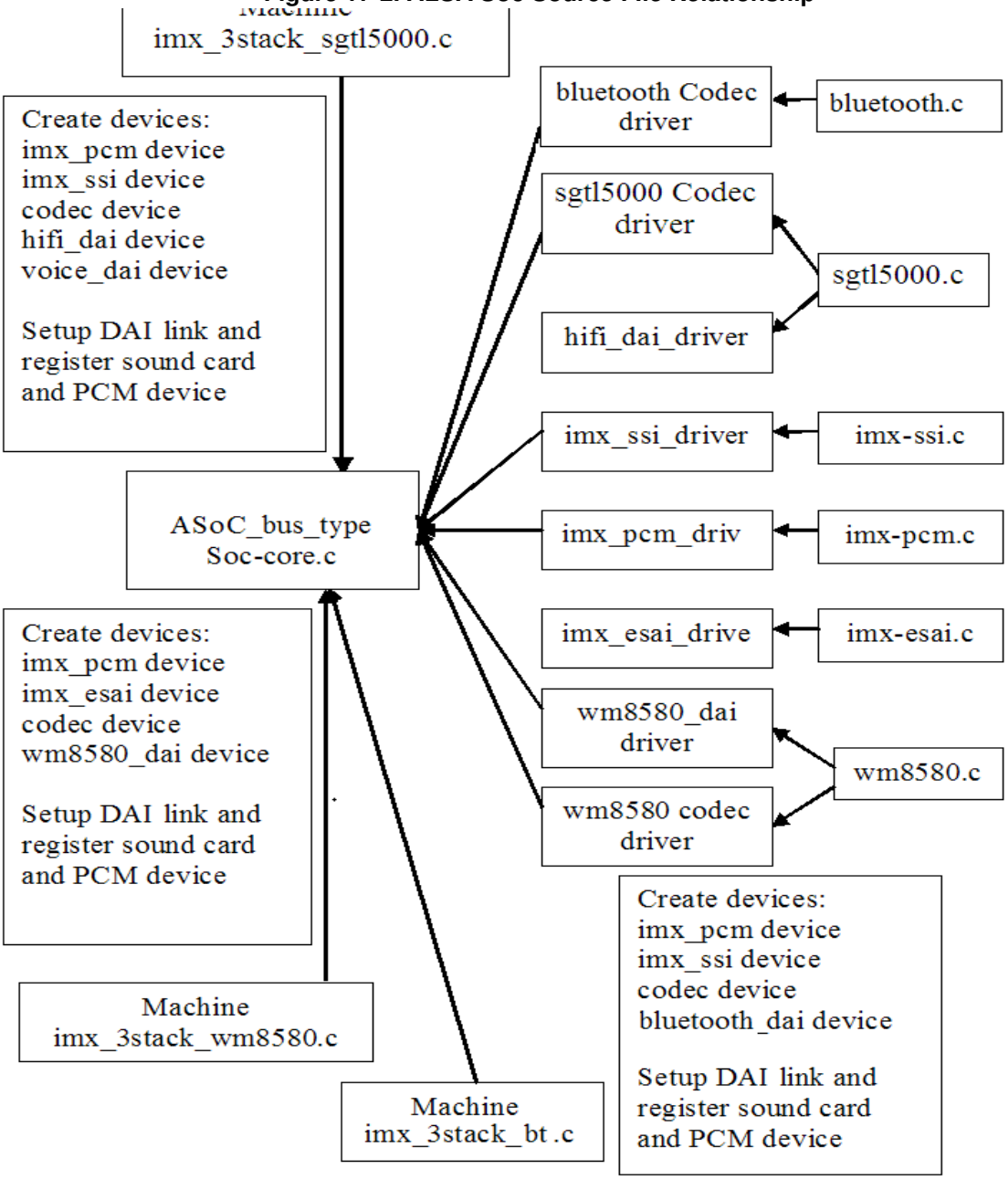


Table 17-1 shows the stereo codec SoC driver source files. These files are under the `<ltib_dir>/rpm/BUILD/linux/sound/soc` directory.

Table 17-1. 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

Table 17-2 shows the multiple-channel ADC SoC driver source files. These files are also under the `<ltib_dir>/rpm/BUILD/linux/sound/soc` directory.

Table 17-2. CS42888 ASoC Driver Source File

File	Description
imx-3stack-cs42888.c	Machine layer for mutliple-channel ADC ALSA SoC
imx/imx-pcm.c	Platform layer for mutliple-channel ADC ALSA SoC
imx/imx-pcm.h	Header file for pcm driver
imx/imx-esai.c	Platform DAI link for mutliple-channel ADC ALSA SoC
imx/imx-esai.h	Header file for platform DAI link
codecs/cs42888.c	codec layer for mutliple-channel ADC ALSA SoC
codecs/cs42888.j	Header file for mutliple-channel ADC driver

17.3 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

- SoC Audio supports for i.MX cs42888. In menuconfig, this is option is available under Device drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio support for IMX - CS42888
- CONFIG_MXC_SSI_DUAL_FIFO: This config is used to enable 2 ssi fifo for audio transfer. In menuconfig, this is option is avaiable under Device drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > MXC SSI enable dual fifo.

17.4 Hardware Operation

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

MX53 EVK boards need re-work, due to the conflict between the FEC PHY and ESAI.

17.4.1 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.

17.5 Software Operation

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

17.5.1 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

17.5.2 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 18
The Sony/Philips Digital Interface (S/PDIF) Tx Driver

The Sony/Philips Digital Interface (S/PDIF) audio module is a stereo transceiver that allows the processor to receive and transmit digital audio. The S/PDIF transceiver allows the handling of both S/PDIF channel status (CS) and User (U) data and includes a frequency measurement block that allows the precise measurement of an incoming sampling frequency.

18.1 S/PDIF Overview

Figure 18-1 shows the block diagram of the S/PDIF interface.

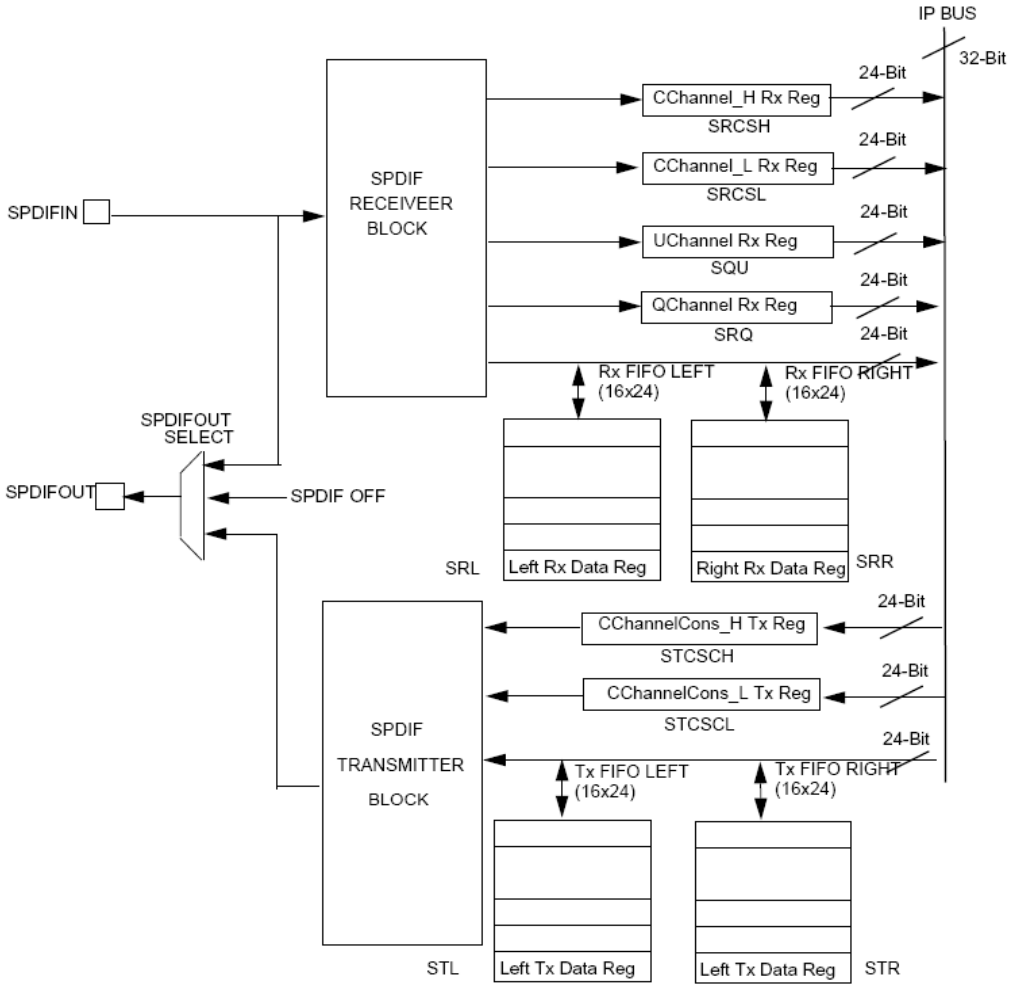


Figure 18-1. S/PDIF Transceiver Data Interface Block Diagram

18.1.1 Hardware Overview

The S/PDIF is composed of two parts:

- The S/PDIF receiver extracts the audio data from each S/PDIF frame and places the data in the S/PDIF Rx left and right FIFOs. The Channel Status and User Bits are also extracted from each frame and placed in the corresponding registers. The S/PDIF receiver provides a bypass option for direct transfer of the S/PDIF input signal to the S/PDIF transmitter.
- For the S/PDIF transmitter, the audio data is provided by the processor through the SPDIFTxLeft and SPDIFTxRight registers. The Channel Status bits are provided through the corresponding registers. The S/PDIF transmitter generates a S/PDIF output bitstream in the biphase mark format (IEC958), which consists of audio data, channel status and user bits.

In the S/PDIF transmitter, the IEC958 biphase bit stream is generated on both edges of the S/PDIF Transmit clock. The S/PDIF Transmit clock is generated by the S/PDIF internal clock generate module and the sources are from outside of the S/PDIF block. For the S/PDIF receiver, it can recover the S/PDIF Rx clock. Figure 18-1 shows the clock structure of the S/PDIF transceiver. MX53 supports S/PDIF Rx and Tx. But MX53 EVK board can only support S/PDIF Tx.

18.1.2 Software Overview

The S/PDIF driver is designed under Linux ALSA subsystem. It provides hardware access ability to support the ALSA driver. The ALSA driver for S/PDIF provides one playback device for Tx and one capture device for Rx. The playback output audio format can be linear PCM data or compressed data with 16-bit default, up to 24-bit expandable support and the allowed sampling bit rates are 44.1, 48 or 32 KHz. The capture input audio format can be linear PCM data or compressed data with 16-bit or 24-bit and the allowed sampling bit rates are from 16 to 96 KHz. The driver provides the same interface for PCM and compressed data transmission.

18.2 S/PDIF Tx Driver

The S/PDIF Tx driver supports the following features:

- 32, 44.1 and 48 KHz sample rates. MX53 EVK only support 44.1KHZ sample rate. To support 48K and 32KHZ sample rate require to connect 24.576MHZ OSC to CKIH2.
- Signed 16 and 24-bit little Endian sample format. Due to S/PDIF SDMA feature, the 24-bit output sample file must have 32-bits in one channel per frame, and only the 24 LSBs are valid

  In the ALSA subsystem, the supported format is defined as S16_LE and S24_LE.
- Two channels
- Driver installation and information query

  By default, the driver is built as a kernel module, run modprobe to install it:

  ```
  #modprobe snd-spdif
  ```

  After the module had been installed, the S/PDIF ALSA driver information can be exported to user by /sys and /proc file system
  - — Get card ID and name

    For example:

```
#cat /proc/asound/cards
0 [imx3stack      ]: SGTL5000 - imx-3stack
         imx_3stack (SGTL5000)
1 [TXRX           ]: MXC_SPDIF - MXC SPDIF TX/RX
         MXC Freescale with SPDIF
```

The number at the beginning of the MXC_SPDIF line is the card ID. The string in the square brackets is the card name

— Get Playback PCM device info

```
#cat /proc/asound/TXRX/pcm[card id]p/info
```

- Software operation

  The ALSA utility provides a common method for user spaces to operate and use ALSA drivers

```
#aplay -D "hw:2,0" -t wav audio.wav
```

NOTE

The -D parameter of `aplay` indicates the PCM device with card ID and PCM device ID: `hw:[card id],[pcm device id]`

18.2.1 Driver Design

Before S/PDIF playback, the configuration, interrupt, clock and channel registers should be initialized. Clock settings are the same for specific hardware connections. During S/PDIF playback, the channel status bits are fixed. The resync, underrun/overrun, empty interrupt and DMA transmit request should be enabled. S/PDIF has 16 TX sample FIFOs on Left and Right channel respectively. When both FIFOs are empty, an empty interrupt is generated if the empty interrupt is enabled. If no data are refilled in the 20.8 μs (1/48000), an underrun interrupt is generated. Overrun is avoided if only 16 sample FIFOs are filled for each channel every time. If auto re-synchronization is enabled, the hardware checks if the left and right FIFO are in sync, and if not, it sets the filling pointer of the right FIFO to be equal to the filling pointer of the left FIFO and an interrupt is generated.

18.2.2 Provided User Interface

The S/PDIF transmitter driver provides one ALSA mixer sound control interface to the user besides the common PCM operations interface. It provides the interface for the user to write S/PDIF channel status codes into the driver so they can be sent in the S/PDIF stream. The input parameter of this interface is the IEC958 digital audio structure shown below, and only status member is used:

```
struct snd_aes_iec958 {
         unsigned char status[24];       /* AES/IEC958 channel status bits */
         unsigned char subcode[147];     /* AES/IEC958 subcode bits */
         unsigned char pad;              /* nothing */
         unsigned char dig_subframe[4];  /* AES/IEC958 subframe bits */
};
```

-

18.3 Source Code Structure

Table 18-1 lists the source file that is available in the directory:

`<ltib_dir>/rpm/BUILD/linux/sound/arm/.`

Table 18-1. S/PDIF Driver Files

File	Description
mxc-alsa-spdif.c	Source file for S/PDIF ALSA driver

18.4 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_SND—Configuration option for the Advanced Linux Sound Architecture (ALSA) subsystem. This option is dependent on CONFIG_SOUND option. In the menuconfig this option is available under

  Device Drivers > Sound card support > Advanced Linux Sound Architecture

  By default, this option is Y.
- CONFIG_SND_MXC_SPDIF—Configuration option for the S/PDIF driver. This option is dependent on CONFIG_SND option. In the menuconfig this option is available under

  Device Drivers > Sound card support > Advanced Linux Sound Architecture > ARM sound devices > MXC SPDIF sound card support

  By default, this option is M.

Chapter 19
SPI NOR Flash Memory Technology Device (MTD) Driver

The SPI NOR Flash Memory Technology Device (MTD) driver provides the support to the Atmel data Flash though the SPI interface. By default, the SPI NOR Flash MTD driver creates static MTD partitions to support Atmel data Flash. If RedBoot partitions exist, they have higher priority than static partitions, and the MTD partitions can be created from the RedBoot partitions.

19.1 Hardware Operation

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.

Unlike conventional Flash memories that are accessed randomly, the AT45DB321D accesses data sequentially. The AT45DB321D operates from a single 2.7–3.6 V power supply for program and read operations. The AT45DB321D is enabled through a chip select pin and accessed through a three-wire interface: Serial Input, Serial Output, and Serial Clock.

19.2 Software Operation

In a Flash-based embedded Linux system, a number of Linux technologies work together to implement a file system. Figure 19-1 illustrates the relationships between some of the standard components.

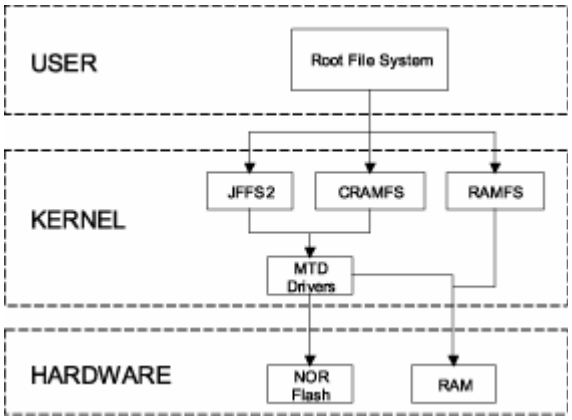


Figure 19-1. Components of a Flash-Based File System

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.

19.3 Driver Features

This NOR MTD implementation supports the following features:

- Provides necessary information for the upper layer MTD driver

19.4 Source Code Structure

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

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

Table 19-1 shows the driver files:

Table 19-1. SPI NOR MTD Driver Files

File	Description
mxc_dataflash.c	Source file

19.5 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:

- 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

Chapter 20
NAND Flash Memory Technology Device (MTD) Driver

20.1 Overview

The NAND Flash MTD driver is for the NAND Flash Controller (NFC) on the i.MX series processor. For the NAND MTD driver to work, only the hardware specific layer has to be implemented. 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.

20.1.1 Hardware Operation

NAND Flash is a non-volatile storage device used for embedded systems. It does not support random access of memory as in the case of RAM or NOR Flash. Reading or writing to NAND Flash has to be through the NFC in the i.MX processors. It uses a multiplexed I/O interface with some additional control pins. It is a sequential access device appropriate for mass storage applications. Code stored on NAND Flash cannot be executed from the NAND Flash. It must be loaded into RAM memory and executed from there.

The NFC in the i.MX processors implements the interface to standard NAND Flash devices. It provides access to both 8-bit and 16-bit NAND Flash. The NAND Flash Control block of the NFC generates all the control signals that control the NAND Flash. The NFC hardware versions vary across i.MX platforms.

20.1.2 Software Operation

The Linux MTD covers all memory devices, such as RAM, ROM, and different kinds of NOR and NAND Flash devices. The MTD subsystem provides a unified and uniform access to the various memory devices.

There are three layers of NAND MTD drivers:

- MTD driver
- Generic NAND driver
- Hardware specific driver

The MTD driver provides a mount point for the file system. It can support various file systems, such as YAFFS2, UBIFS, CRAMFS and JFFS2.

The hardware specific driver interfaces with the integrated NFC on the i.MX processors. It implements the lowest level operations on the external NAND Flash chip, such as read and write. It defines the static partitions and registers it to the kernel. This partition information is used by the upper filesystem layer. It initializes the `nand_chip` structure to be used by the generic layer.

The generic layer provides all functions, which are necessary to identify, read, write and erase NAND Flash. It supports bad block management, because blocks in a NAND Flash are not guaranteed to be good. The upper layer of the file system uses this feature of bad block management to manage the data on the NAND Flash. NAND MTD driver is part of the kernel image. For detailed information on the NAND MTD driver architecture and the NAND API documentation refer to http://www.linux-mtd.infradead.org/.

20.2 Requirements

This NAND Flash MTD driver implementation meets the following requirements:

- Provides necessary hardware-specific information to the generic layer of the NAND MTD driver
- Provides software Error Correction Code (ECC) support
- Supports both 16-bit and 8-bit NAND Flash
- Conforms to the Linux coding standard

20.3 Source Code Structure

Table 20-1 shows the source files available for the NAND MTD driver. These files are under the `<ltib_dir>/rpm/BUILD/linux/drivers/mtd/nand` directory.

20.4 Linux 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:

The following options are available under Device Driver > Memory Technology Device (MTD) support > NAND Device Support > MXC NAND Version 3 support.

20.5 Programming Interface

The generic NAND driver `nand_base.c` provides all functions that are necessary to identify, read, write, and erase NAND Flash. The hardware-dependent functions are provided by the hardware driver `mxc_nd.c/mxc_nd2.c` depending on the NFC version. It mainly provides the hardware access information and functions for the generic NAND driver. Refer to the API documents for the programming interface.

Chapter 21
SATA Driver

21.1 Hardware Operation

The detailed hardware operation of SATA is detailed in the Synopsys DesignWare Cores SATA AHCI documentation, named SATA_Data_Book.pdf.

21.2 Software Operation

The details about the libata APIs, see the libATA Developer's Guide named libata.pdf pulished by Jeff Gazik.

The SATA AHCI driver is based on the LIBATA layer of the block device infrastructure of the Linux kernel . FSL integrated AHCI linux driver combined the standard AHCI drivers handle the details of the integrated freescale's SATA AHCI controller, while the LIBATA layer understands and executes the SATA protocols. The SATAdevice, such as a hard disk, is exposed to the application in user space by the /dev/sda* interface. Filesystems are built upon the block device. The AHCI specified integrated DMA engine, which assists the SATA controller hardware in the DMA transfer modes.

21.3 Source Code Structure Configuration

The source codes of freescale's AHCI sata driver is integrated into the i.MX53 platform related files. `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/mx53_evk.c.`

The standard AHCI and AHCI platform drivers are used to do the actual sata operations.

The source codes of the standard AHCI and AHCI platform drivers are located in `drivers/ata/` folder, named as ahci.c and ahci-platform.c.

21.4 Linux Menu Configuration Options

The following Linux kernel configurations are provided for SATA driver:

- CONFIG_SATA_AHCI_PLATFORM: Configure options for SATA driver. In the menuconfig this option is available under "Device Drivers --->Serial ATA (prod) and Parallel ATA (experimental) drivers -> Platform AHCI SATA support".

In busybox, enable "fdisk" under "Linux System Utilities".

21.5 Board Configuration Options

With the power off, install the SATA cable and hard drive.

21.6 Programming Interface

The application interface to the SATA driver is the standard POSIX device interface (for example: open, close, read, write, and ioctl) on `/dev/sda*`.

21.7 Usage Example

NOTE

There would be a known error message when the SATA disk is initialized, such as:

ata1.00: serial number mismatch '090311PB0300QKG3TB1A' != ''

ata1.00: revalidation failed (errno=-19)

pls ignore that.

1. After building the kernel and the SATA AHCI driver and deploying, boot the target, and log in as root.
2. Make sure that the AHCI and AHCI paltform drivers are built in kernel or loaded into kernel. Use the following commands to load the drivers into kernel.

```
# insmod libata.ko
# insmod libahci.ko
# insmod ahci-platform.ko
```

You should see messages similar to the following:

```
ahci: SSS flag set, parallel bus scan disabled
ahci ahci.0: AHCI 0001.0100 32 slots 1 ports 3 Gbps 0x1 impl platform mode
ahci ahci.0: flags: ncq sntf stag pm led clo only pmp pio slum part ccc
scsi0 : ahci
ata1: SATA max UDMA/133 irq_stat 0x00000040, connection status changed irq 28
ata1: SATA link up 1.5 Gbps (SStatus 113 SControl 300)
ata1.00: ATA-8: Hitachi HTS545032B9A300, PB3OC60G, max UDMA/133
ata1.00: 625142448 sectors, multi 0: LBA48 NCQ (depth 31/32)
ata1.00: serial number mismatch '090311PB0300QKG3TB1A' != ''
ata1.00: revalidation failed (errno=-19)
ata1: limiting SATA link speed to 1.5 Gbps
ata1.00: limiting speed to UDMA/133:PIO3
ata1: SATA link up 1.5 Gbps (SStatus 113 SControl 310)
ata1.00: configured for UDMA/133
scsi 0:0:0:0: Direct-Access     ATA      Hitachi HTS54503 PB3O PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 625142448 512-byte logical blocks: (320 GB/298 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO or
FUA
 sda: sda1 sda2 sda3
sd 0:0:0:0: [sda] Attached SCSI disk
```

You may use standard Linux utilities to partition and create a file system on the drive (for example: fdisk and mke2fs) to be mounted and used by applications.

The device nodes for the drive and its partitions appears under `/dev/sda*`. For example, to check basic kernel settings for the drive, execute `hdparm /dev/sda`.

21.8 Usage Example

Create Partitons

The following command can be used to find out the capacities of the hard disk. If the hard disk is pre-formatted, this command shows the size of the hard disk, partitions, and filesystem type:

```
$fdisk -l /dev/sda
```

If the hard disk is not formatted, create the partitions on the hard disk using the following command:

```
$fdisk /dev/sda
```

After the partition, the created files resemble `/dev/sda[1-4]`.

Block Read/Write Test:

The command, `dd`, is used for for reading/writing blocks. Note this command can corrupt the partitions and filesystem on Hard disk.

To clear the first 5 KB of the card, do the following:

```
$dd if=/dev/zero of=/dev/sda1 bs=1024 count=5
```

The response should be as follows:

5+0 records in

5+0 records out

To write a file content to the card enter the following text, substituting the name of the file to be written for `file_name`, do the following:

```
$dd if=file_name of=/dev/sda1
```

To read 1KB of data from the card enter the following text, substituting the name of the file to be written for `output_file`, do the following:

```
$dd if=/dev/sda1 of=output_file bs=1024 count=1
```

Files System Tests

Format the hard disk partitons using mkfs.vfat or mkfs.ext2, depending on the filesystem:

```
$mkfs.ext2 /dev/sda1
$mkfs.vfat /dev/sda1
```

Mount the file system as follows:

```
$mkdir /mnt/sda1
$mount -t ext2 /dev/sda1 /mnt/sda1
```

After mounting, file/directory, operations can be performed in `/mnt/sda1`.

Unmount the filesystem as follows:

```
$umount /mnt/sda1
```

Chapter 22
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.

22.1 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.

22.2 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 22-1. The `mxc_kpp_scan_matrix` function is called for every keypress and release interrupt.

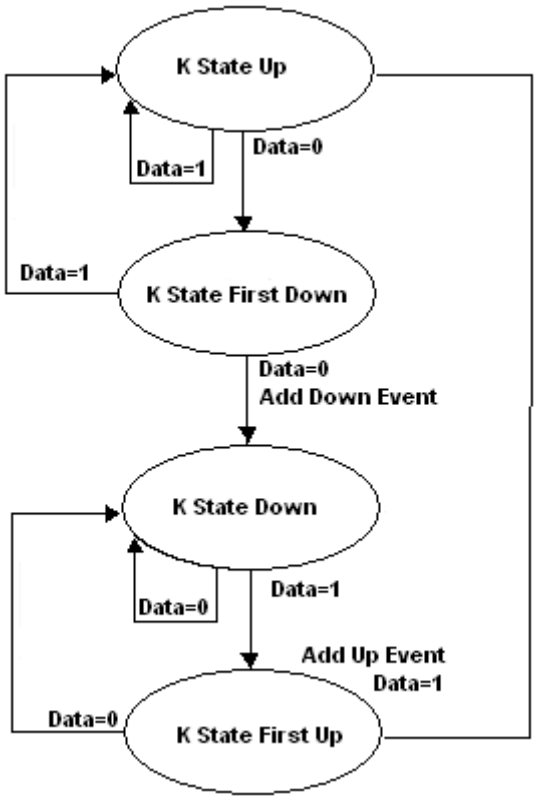


Figure 22-1. Keypad Driver State Machine

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;
```

22.3 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.

22.4 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

22.5 Implemented as a standard input device MX53 EVK Keypad

MX51 accessory card can be connected with MX53 EVK board to support keypad functions. However, the keypad codes can not be merged into current BSP because the keypad PINs conflict with CAN and audio features. Here one example is given to share how to support keypad in MX53:

- Add the platform data in to mx53_evk.c and register keypad device. Remove CAN device register.

```
--- a/arch/arm/mach-mx5/mx53_evk.c
+++ b/arch/arm/mach-mx5/mx53_evk.c
@@ -545,6 +545,20 @@ static struct mxc_mlb_platform_data mlb_data = {
        .mlb_clk = "mlb_clk",
 };

+static u16 keymapping[] = {
+       KEY_UP, KEY_DOWN, KEY_MENU, KEY_BACK,
+       KEY_RIGHT, KEY_LEFT, KEY_SELECT, KEY_ENTER,
+       KEY_F1,
+};
+
+static struct keypad_data keypad_plat_data = {
+       .rowmax = 3,
+       .colmax = 3,
```

```
+       .learning = 0,
+       .delay = 2,
+       .matrix = keymapping,
+};
/*!
  * Board specific fixup function. It is called by \b setup_arch() in
  * setup.c file very early on during kernel starts. It allows the user to
@@ -684,10 +698,12 @@ static void __init mxc_board_init(void)
                mxc_register_device(&mxc_pwm2_device, NULL);
               mxc_register_device(&mxc_pwm_backlight_device, &mxc_pwm_backlight_data);
        }
+#if 0
        mxc_register_device(&mxc_flexcan0_device, &flexcan0_data);
        mxc_register_device(&mxc_flexcan1_device, &flexcan1_data);
+#endif

-/*     mxc_register_device(&mxc_keypad_device, &keypad_plat_data); */
+       mxc_register_device(&mxc_keypad_device, &keypad_plat_data);
```

- Configure Keypad PINs in mx53_evk_gpio.c

```
--- a/arch/arm/mach-mx5/mx53_evk_gpio.c
+++ b/arch/arm/mach-mx5/mx53_evk_gpio.c
@@ -160,19 +160,19 @@ static struct mxc_iomux_pin_cfg __initdata mxc_iomux_pins[] = {
          PAD_CTL_DRV_HIGH | PAD_CTL_SRE_FAST),
         },
        {
-        MX53_PIN_KEY_COL0, IOMUX_CONFIG_ALT2,
+        MX53_PIN_KEY_COL0, IOMUX_CONFIG_ALT0,
         },
        {
-        MX53_PIN_KEY_ROW0, IOMUX_CONFIG_ALT2,
+        MX53_PIN_KEY_ROW0, IOMUX_CONFIG_ALT0,
         },
        {
-        MX53_PIN_KEY_COL1, IOMUX_CONFIG_ALT2,
+        MX53_PIN_KEY_COL1, IOMUX_CONFIG_ALT0,
         },
-        MX53_PIN_KEY_COL3, IOMUX_CONFIG_ALT4,
+        MX53_PIN_KEY_COL3, IOMUX_CONFIG_ALT0,
         },
        {
         MX53_PIN_CSI0_D7, IOMUX_CONFIG_ALT5,
@@ -225,6 +225,13 @@ static struct mxc_iomux_pin_cfg __initdata mxc_iomux_pins[] = {
        {
         MX53_PIN_GPIO_10, IOMUX_CONFIG_GPIO,
         },
+       { /* KEY COL2 */
+        MX53_PIN_KEY_COL2, IOMUX_CONFIG_ALT0,
+        },
+       { /* KEY_ROW2 */
+        MX53_PIN_KEY_ROW2, IOMUX_CONFIG_ALT0,
+        },
+#if 0
        { /* CAN1-TX */
         MX53_PIN_KEY_COL2, IOMUX_CONFIG_ALT2,
         (PAD_CTL_DRV_HIGH | PAD_CTL_HYS_ENABLE | PAD_CTL_PKE_ENABLE |
```

```
@@ -281,6 +288,7 @@ static struct mxc_iomux_pin_cfg __initdata mxc_iomux_pins[] = {
          (PAD_CTL_DRV_HIGH | PAD_CTL_HYS_ENABLE | PAD_CTL_PKE_ENABLE |
           PAD_CTL_PUE_PULL | PAD_CTL_100K_PU | PAD_CTL_ODE_OPENDRAIN_NONE),
          },
+#endif
         {
-          MX53_PIN_GPIO_11, IOMUX_CONFIG_GPIO,
```

22.6 Source Code Structure

Table 22-1 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 22-1. 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/mx53_evk.cmx50_rdp.c arch/arm/mach-mx5/device.c	Contains the platform-specific keymapping keycode array

22.7 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 22-1).
- Debounce delay—The user can configure the debounce delay by changing the variable `KScanRate` defined in `mxc_keyb.c`.

22.8 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/`.

22.9 Interrupt Requirements

Table 22-2 lists the keypad interrupt timer requirements.

Table 22-2. 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 23
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).

23.1 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 *i.MX53 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 23-1.

Table 23-1. 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 23-1. 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 *i.MX53MX50Multimedia 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.

23.2 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

23.3 Source Code Structure

Table 23-2 shows the source files available in the `<ltib_dir>/rpm/BUILD/linux/drivers/net` directory.

Table 23-2. 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.

23.4 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.

23.5 Programming Interface

Table 23-2 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.

23.5.1 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 */
```

23.5.2 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 24
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

24.1 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

24.2 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.

24.3 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

24.4 Software Operation

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

24.4.1 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.

24.4.2 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.

24.5 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.

24.6 Source Code Structure

Table 24-1 shows the I^2C bus driver source files available in the directory:

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

Table 24-1. I^2C Bus Driver Files

File	Description
i2c-imx.c	I^2C bus driver source file

24.7 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 > IMX I2C interface.

-

24.8 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`.

24.9 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 24-2.

Table 24-2. 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 25
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

25.1 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)

25.2 Software Operation

The following sections describe the CSPI software operation.

25.2.1 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 25-1 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.

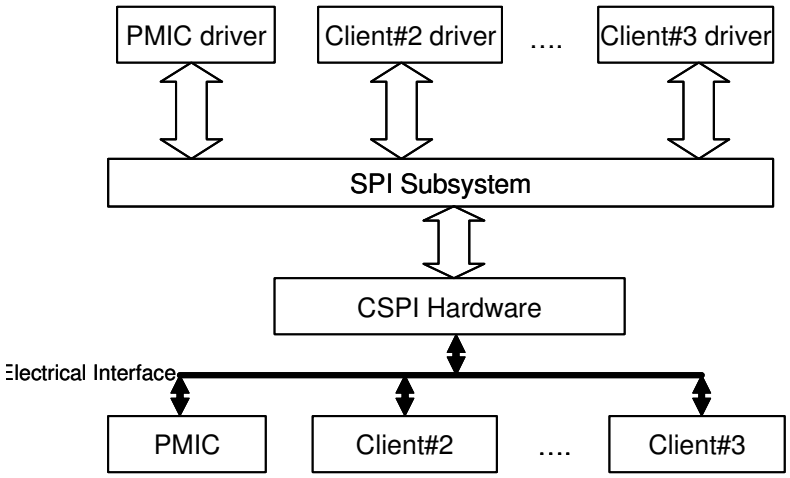


Figure 25-1. SPI Subsystem

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 25-2 shows how the different SPI drivers are layered in the SPI subsystem.

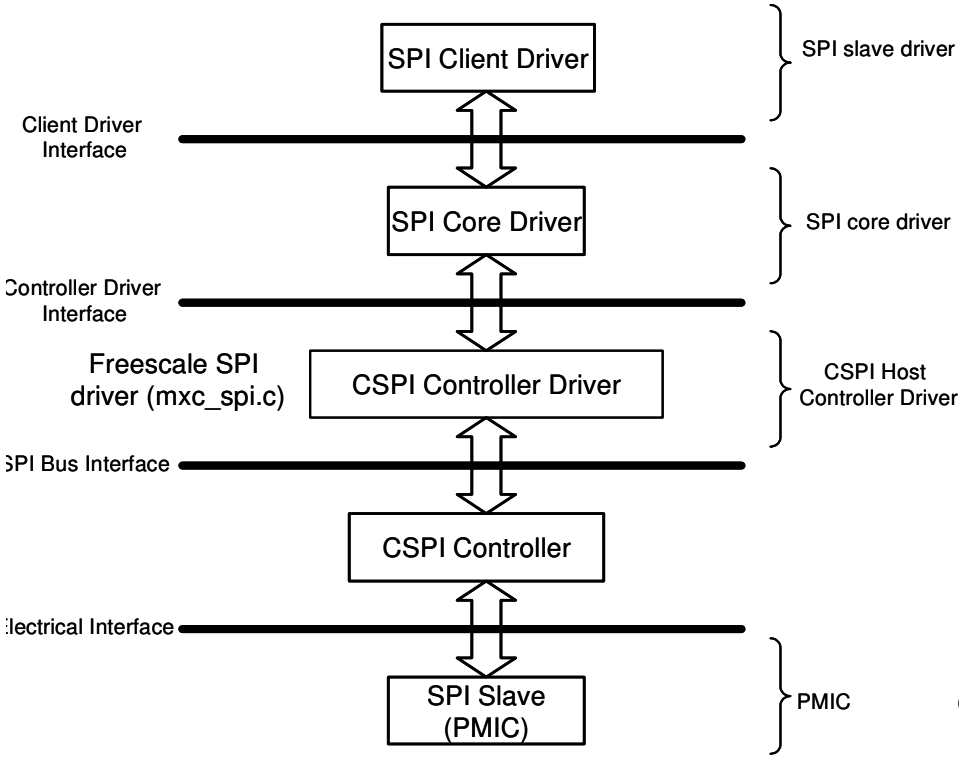


Figure 25-2. Layering of SPI Drivers in SPI Subsystem

25.2.2 Software Limitations

The 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

25.2.3 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.

25.2.4 CSPI Synchronous Operation

Figure 25-3 shows how the CSPI provides synchronous read/write operations.

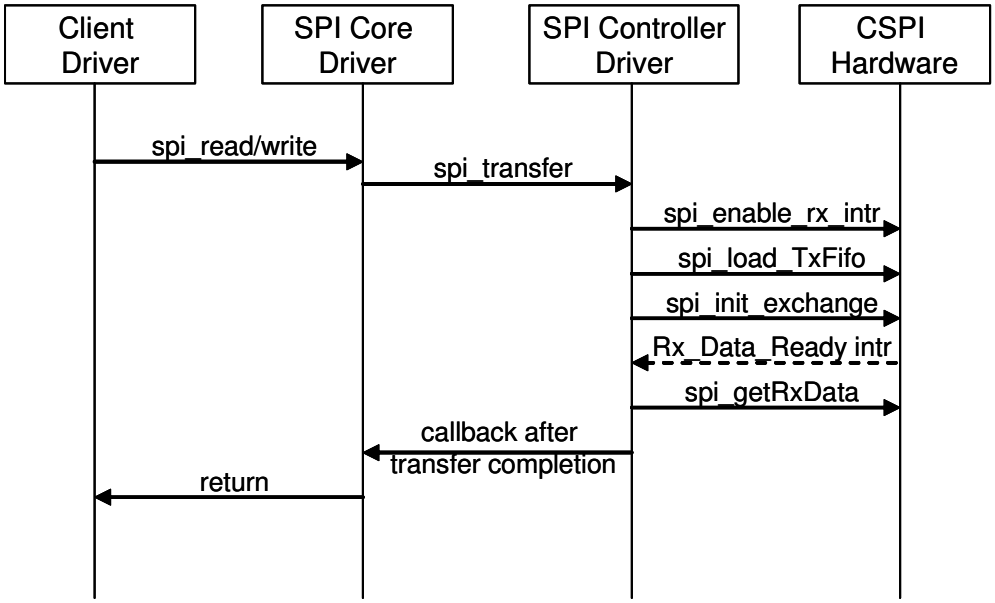


Figure 25-3. CSPI Synchronous Operation

25.3 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

25.4 Source Code Structure

Table 25-1 shows the source files available in the devices directory:

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

Table 25-1. CSPI Driver Files

File	Description
mxc_spi.c	SPI Master Controller driver

25.5 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.

25.6 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).

25.7 Interrupt Requirements

The SPI interface generates interrupts. CSPI interrupt requirements are listed in Table 25-2.

Table 25-2. 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 26
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

26.1 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.

26.2 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 26-1 shows how the MMC-related drivers are layered.

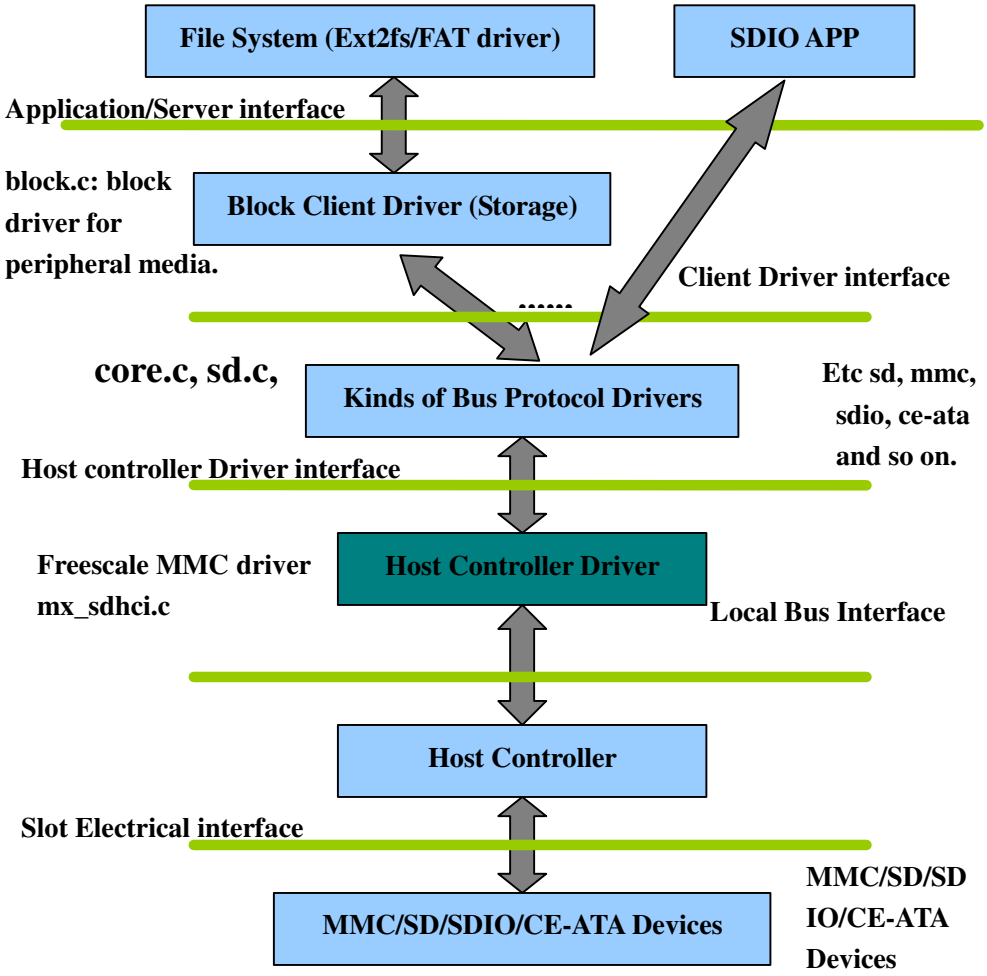


Figure 26-1. MMC Drivers Layering

26.3 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

26.4 Source Code Structure

Table 26-1 shows the eSDHC source files available in the source directory:
`<ltib_dir>/rpm/BUILD/linux/drivers/mmc/host/.`

Table 26-1. eSDHC Driver FilesMMC/SD Driver Files

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

26.5 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.

26.6 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 27
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.

27.1 Hardware Operation

Refer to the *i.MX53MX50 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.

27.2 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.

27.3 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

27.4 Source Code Structure

Table 27-1 shows the UART driver source files that are available in the directory:

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

Table 27-1. 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 27-2 shows the header files associated with the UART driver.

Table 27-2. 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-mx55`. The source file contains UART configuration data and calls to register the device with the platform bus.

27.5 Configuration

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

27.5.1 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.

27.5.2 Source Code Configuration Options

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

27.5.2.1 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 27-5.

27.5.2.2 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*i.MX53*, the board specific configuration options for the driver is set within

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

27.6 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.

27.7 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 27-3.

Table 27-3. 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.

27.8 Device Specific Information

27.8.1 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 27-4.

27.8.2 Board Setup Configuration

Table 27-4. UART General Configuration

Platform	Number of UARTs	Max Baudrate
i.MX508	3	4Mbps
i.MX53	5	4 Mbps

Table 27-5. UART Active/Inactive Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	1	1	1	0	0	0
i.MX53	1	1	1	1	1	—

Table 27-6. UART IRDA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	NO_IRDA	NO_IRDA	NO_IRDA			
i.MX53	NO_IRDA	NO_IRDA	NO_IRDA	NO_IRDA	NO_IRDA	—

Table 27-7. UART Mode Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	MODE_DCE	MODE_DCE	MODE_DCE			
i.MX53	MODE_DCE	MODE_DCE	MODE_DCE	MODE_DCE	MODE_DCE	—

Table 27-8. UART Shared Peripheral Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	-1	-1	-1			
i.MX53	-1	-1	SPBA_UART3	-1	-1	—

Table 27-9. UART Hardware Flow Control Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	0	0	0			
i.MX53	1	1	1	1	1	—

Table 27-10. UART DMA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	0	0	1			
i.MX53	0	1	0	0	0	—

Table 27-11. UART DMA RX Buffer Size Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	1024	512	1024			
i.MX53	1024	512	1024	512	512	—

Table 27-12. UART UCR4_CTSTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	16	-1	16			
i.MX53	16	-1	16	-1	-1	—

Table 27-13. UART UFCR_RXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	16	16	16			
i.MX53	16	16	16	16	16	—

Table 27-14. UART UFCR_TXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	16	16	16			
i.MX53	16	16	16	16	16	—

Table 27-15. UART Interrupt Mux Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	INTS_MUXED	INTS_MUXED	INTS_MUXED			
i.MX51	INTS_MUXED	INTS_MUXED	INTS_MUXED	INTS_MUXED	INTS_MUXED	—

Table 27-16. UART Interrupt 1 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	MXC_INT_UART1	MXC_INT_UART2	MXC_INT_UART3			
i.MX53	MXC_INT_UART1	MXC_INT_UART2	MXC_INT_UART3	MXC_INT_UART4	MXC_INT_UART5	—

Table 27-17. UART Interrupt 2 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	-1	-1	-1			
i.MX53	-1	-1	-1	-1	-1	—

Table 27-18. UART interrupt 3 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX508	-1	-1	-1			
i.MX53	-1	-1	-1	-1	-1	—

27.9 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 28
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

28.1 Architectural Overview

A USB host system is composed of a number of hardware and software layers. Figure 28-1 shows a conceptual block diagram of the building block layers in a host system that support USB 2.0.

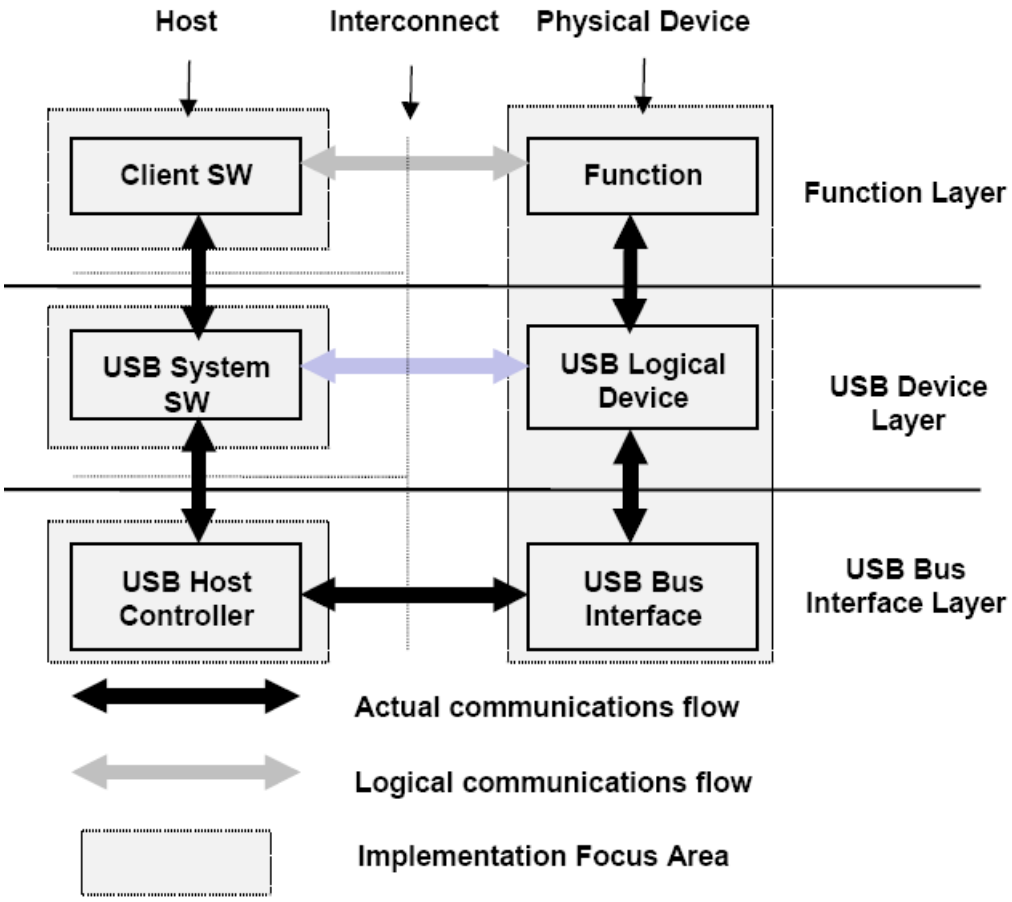


Figure 28-1. USB Block Diagram

28.2 Hardware Operation

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

28.3 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.

28.4 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

28.5 Source Code Structure

Table 28-1 shows the source files available in the source directory, `<ltib_dir>/rpm/BUILD/linux/drivers/usb`.

Table 28-1. 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 28-2 shows the platform related source files.

Table 28-2. USB Platform Source Files

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

Table 28-3 shows the platform-related source files in the directory:
`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx5/`

Table 28-3. USB Platform Header Files

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

Table 28-4 shows the common platform source files in the directory:

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

Table 28-4. USB Common Platform Files

File	Description
isp1504xc.c	ULPI PHY driver (USB3317 uses the same driver as ISP1504)
utmixc.c	Internal UTMI transceiver driver
usb_common.c	Common platform related part of USB driver

28.6 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.

28.7 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.

28.8 Default USB Settings

Table 28-5 shows the default USB settings.

Table 28-5. Default USB Settings

Platform	OTG HS	OTG FS	Host1	Host2(HS)	Host2(FS)
i.MX53 EVK	enable	N/A	enable (HS)	N/A	N/A
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 — usb ethernet gadget , build as M
  CONFIG_USB_FILE_STORAGE—usb mass storage gadget, build as M
  then , 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

28.9 Remote WakeUp

- OTG device do not support SET/CLEAR_FEATURE Remote-wakeup
- HOST support Remote-wakeup by usb device

28.10 System WakeUp

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

28.11 USB Wakeup usage

28.11.1 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
```

28.11.2 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)
```

28.11.3 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 29
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.

29.1 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.

29.2 Software Operation

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

29.2.1 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
```

29.2.2 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. 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.

29.3 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

29.4 Source Code Structure

The RTC module is implemented in the following directory:

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

Table 29-1 shows the RTC module files.

Table 29-1. RTC Driver Files

File	Description
rtc-mxc_v2.c	SRTC driver implementation file

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

29.5 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 30
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.

30.1 Hardware Operation

Once the WDOG timer is activated, it must be serviced by software on a periodic basis. If servicing does not take place in time, the WDOG times out. Upon a time-out, the WDOG either asserts the `wdog_b` signal or a `wdog_rst_b` system reset signal, depending on software configuration. The watchdog module cannot be deactivated once it is activated.

30.2 Software Operation

The Linux OS has a standard WDOG interface that allows support of a WDOG driver for a specific platform. WDOG can be suspended/resumed in STOP/DOZE and WAIT modes independently. Since some bits of the WGOD registers are only one-time programmable after booting, ensure these registers are written correctly.

30.3 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.

30.3.1 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

30.3.2 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.

30.3.3 Source Code Structure

Table 30-1 shows the source files for WDOG drivers that are in the following directory:

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

Table 30-1. WDOG Driver Files

File	Description
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`

30.3.4 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 31
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.

31.1 Hardware Operation

Figure 31-1 shows the PWM block diagram.

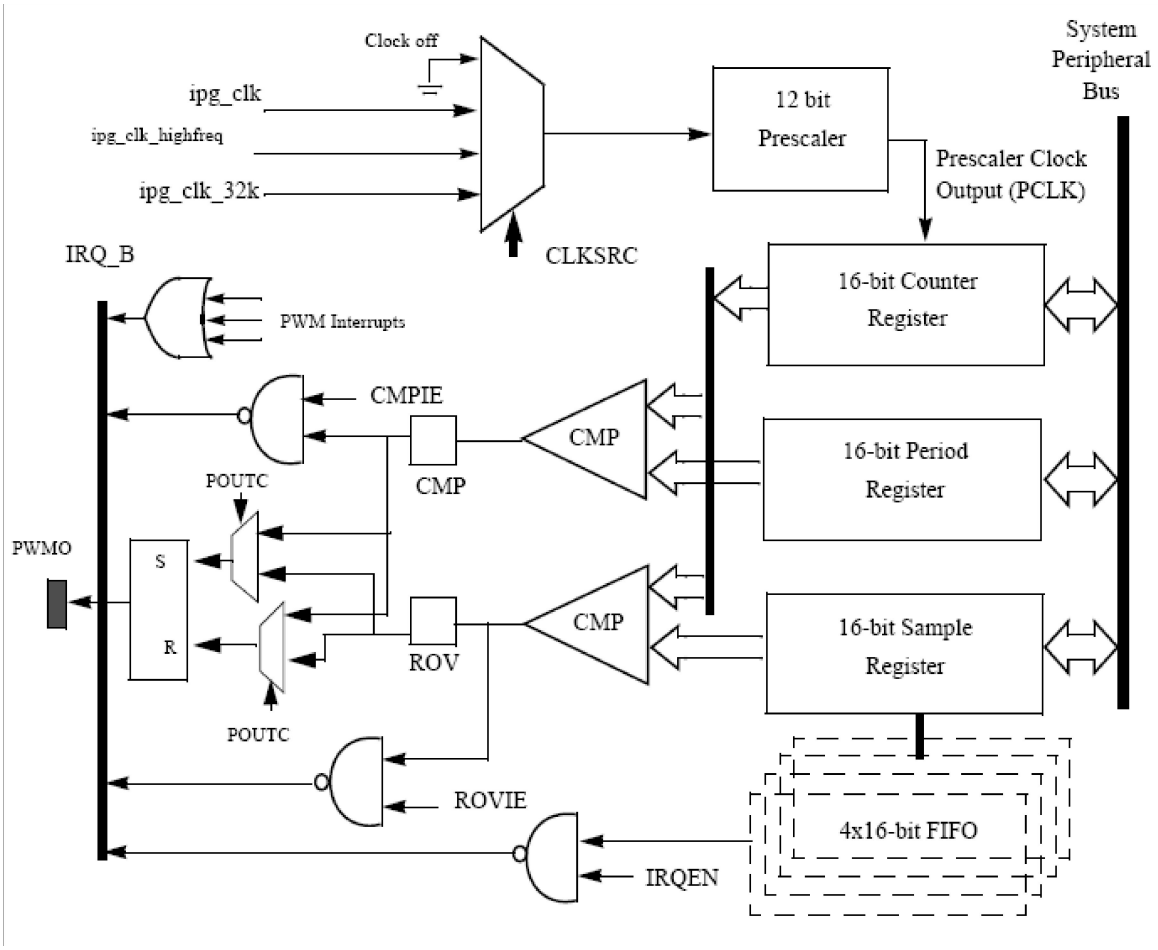


Figure 31-1. PWM Block Diagram

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

31.2 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.

31.3 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 31-1 provides a summary of the interface functions in source code.

Table 31-1. 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.

31.4 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

31.5 Source Code Structure

Table 31-2 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 31-2. PWM Driver Files

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

31.6 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 32
FlexCAN Driver

32.1 Driver Overview

FlexCAN is a communication controller implementing the CAN protocol according to the CAN 2.0B protocol specification. The CAN protocol was primarily designed to be used as a vehicle serial data bus, meeting the specific requirements of this field such as real-time processing, reliable operation in the EMI environment of a vehicle, cost-effectiveness and required bandwidth. The standard and extended message frames are supported. The maximum message buffer is 64. The driver is a network device driver of PF_CAN protocol family.

For the detailed information, see http://lwn.net/Articles/253425 or `Documentation/networking/can.txt` in Linux source directory.

32.2 Hardware Operation

For the information on hardware operations, see the *i.MX53 Multimedia Applications Processor Reference Manual*.

32.3 Software Operation

The CAN driver is a network device driver. For the common information on software operation, refer to the documents in the kernel source directory `Documentation/networking/can.txt`.

The driver includes parameters that need to be set by the user to use CAN such as the bitrate, clock source, and so on. Currently the driver only supports the configuration when the device is not activated. To configure the CAN parameters, enter directory `/sys/devices/platform/FlexCAn.x/` (`x` is the device number):

- `br_clksrc` configures the clock source
- `bitrate` configures the bitrate. Currenlty, this parameter only shows the bitrate that is supported. To ensure `bitrate` exactly, set the individual parameters:
  - `br_presdiv` configures prescale divider
  - `br_rjw` configures RJW
  - `br_propseg` configures the length of the propagation segment
  - `br_pseg1` configures the length of phase buffer segment 1
  - `br_pseg2` configures the length of phase buffer segment 2
- `abort` enables or disables abort feature
- `bcc` sets backwards compatibility with previous FlexCAN versions

- boff_rec configures support of recover from bus off state
- fifo enables or disables FIFO work mode
- listen enables or disables listen only mode
- local_priority enables or disables the local priority. In current version, this parameter is not used
- loopback sets hardware at loopback mode or not
- maxmb sets the maximum message buffers
- smp sets the sampling mode
- srx_dis disables or enables the self-reception
- state shows the device status
- ext_msg configures support for extended message
- std_msg configures support for standard message
- tsyn enables or disables timer synchronization feature
- wak_src sets wakeup source
- wakeup enables or disables self-wakeup
- xmit_maxmb sets the maximum message buffer for the transmission

There are two operations to activate or deactivate CAN interface. Using the CAN0 interfaces as an example:

- ifconfig can0 up
- ifconfig can0 down

32.4 Source Code Structure

Table 32-1 shows the driver source file available in the directory, <ltib_dir>/rpm/BUILD/linux/drivers/net/can/flexcan.

Table 32-1. FlexCAN Driver Files

File	Description
dev.c	Operation about parameters
drv.c	Network device driver
mbm.c	Management of message buffer
flexcan.h	Head file of FlexCAN

32.5 Linux 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_CAN – Build support for PF_CAN protocol family. In menuconfig, this option is available under

  Networking > CAN bus subsystem support.

- CONFIG_CAN_RAW – Build support for Raw CAN protocol. In `menuconfig`, this option is available under

  Networking > CAN bus subsystem support > Raw CAN Protocol (raw access with CAN-ID filtering).
- CONFIG_CAN_BCM – Build support for Broadcast Manager CAN protocol. In `menuconfig`, this option is available under

  Networking > CAN bus subsystem support > Broadcast Manager CAN Protocol (with content filtering).
- CONFIG_CAN_VCAN – Build support for Virtual Local CAN interface (also in Ethernet interface). In `menuconfig`, this option is available under

  Networking > CAN bus subsystem support > CAN Device Driver > Virtual Local CAN Interface (vcan).
- CONFIG_CAN_DEBUG_DEVICES – Build support to produce debug messages to the system log to the driver. In `menuconfig`, this option is available under

  Networking > CAN bus subsystem support > CAN Device Driver > CAN devices debugging messages.
- CONFIG_CAN_FLEXCAN – Build support for FlexCAN device driver. In `menuconfig`, this option is available under

  Networking > CAN bus subsystem support > CAN Device Driver > Freescale FlexCAN.

Chapter 33
Media Local Bus Driver

MediaLB is an on-PCB or inter-chip communication bus specifically designed to standardize a common hardware interface and software API library. This standardization allows an application or multiple applications to access the MOST Network data, or to communicate with other applications, with minimum effort. MediaLB supports all the MOST Network data transport methods: synchronous stream data, asynchronous packet data, and control message data. MediaLB also support an isochronous data transport method. For detailed information about the MediaLB, see the Media Local Bus Specification.

33.1 Overview

33.1.1 MLB Device Module

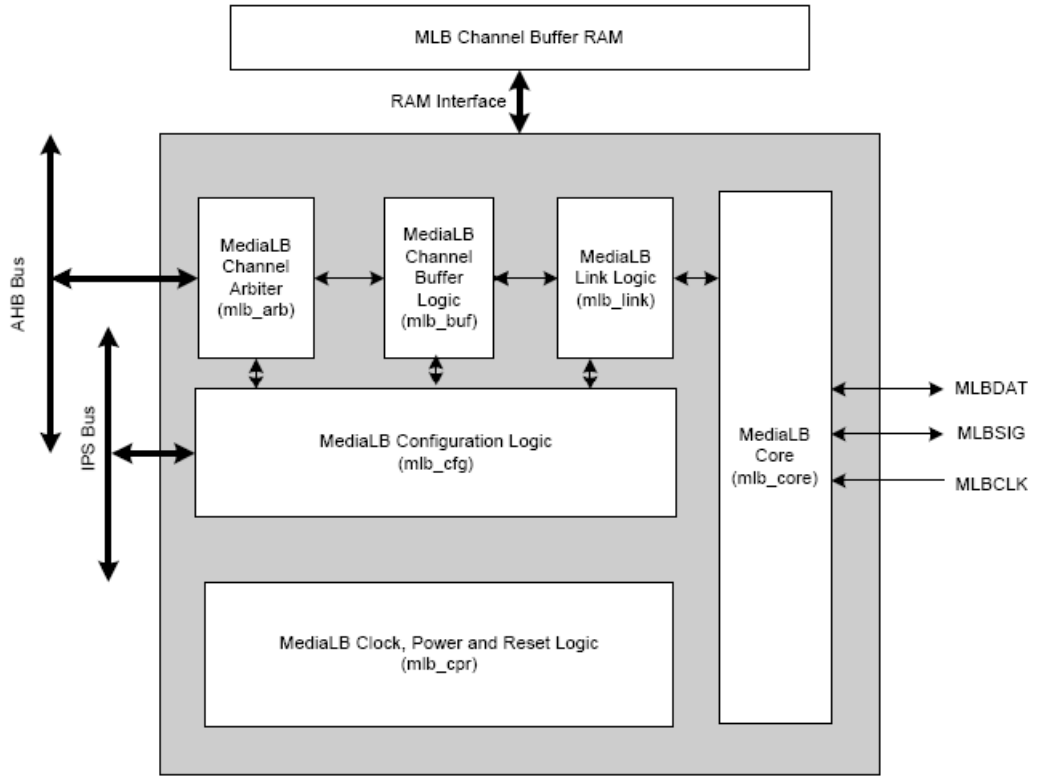


Figure 33-1. MLB Device Top-Level Block Diagram

The MediaLB module implements the Physical Layer and Link Layer of the MediaLB specification, interfacing the i.MX to the MediaLB controller. The MLB implements the 3-pin MediaLB mode and can run at speeds up to 1024Fs. It does not implement MediaLB controller functionality. All MediaLB devices

support a set of physical channels for sending data over the MediaLB. Each physical channel is 4 bytes in length (quadlet) and grouped into logical channels with one or more physical channels allocated to each logical channel. These logical channels can be any combination of channel type (synchronous, asynchronous, control, or isochronous) and direction (transmit or receive).

The MLB provides support for up to 16 logical channels and up to 31 physical channels with a maximum of 124 bytes of data per frame. Each logical channel is referenced using an unique channel address and represents a unidirectional data path between a MediaLB device transmitting the data and the MediaLB device(s) receiving the data.

33.1.1.1 Supported Feature

- Synchronous, asynchronous, control and isochronous channel.
- Up to 16 logical channels and 31 physical channels running at a maximum speed of 1024Fs
- Transmission of commands and data and reception of receive status when functioning as the transmitting device associated with a logical channel address
- Reception of commands and data and transmission as receive status responses when functioning as the receiving device associated with a logical channel address
- MediaLB lock detection
- System channel command handling

33.1.1.2 Modes of Operation

- Normal mode. The MediaLB Device dictates two particular methods:
  - — Ping-Pong Buffering mode
  - — Circular Buffering mode (only used on synchronous type transfer)
- Loop-Back test mode

33.1.2 MLB Driver Overview

The MLB driver is designed as a common linux character driver. It implements one asynchronous and one control channel device with Ping-Pong buffering operation mode. The supported frame rates are 256, 512, and 1024Fs. The MLB driver uses common read/write interfaces to receive/send packets and uses the `ioctl` interface to configure the MLB device module.

33.2 MLB Driver

33.2.1 Supported Features

- 256Fs, 512Fs and 1024Fs frame rates
- Asynchronous and control channel types
- The following configurations to MLB device module:
  - — Frame rate
  - — Device address

— Channel address

- MLB channel exception get interface. All the channel exceptions are sent and handled by the application.

33.2.2 MLB Driver Architecture

The MLB driver is a common linux character driver and the architecture is shown in Figure 33-2.

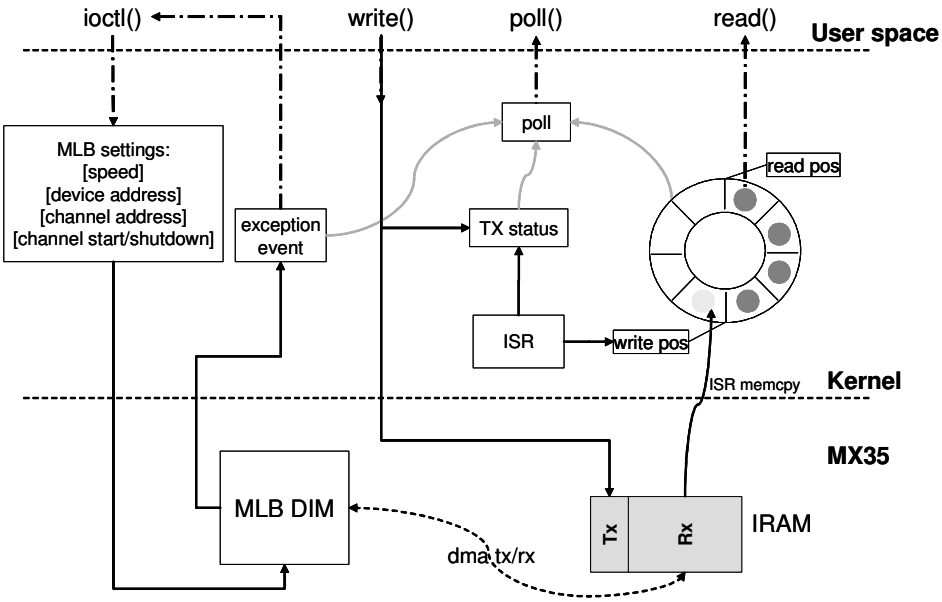


Figure 33-2. MLB Driver Architecture Diagram

The MLB driver creates two minor devices, one for control tx/rx channel and the other for asynchronous. Their device files are `/dev/ctrl` and `/dev/async`. Each minor device has the same interfaces, and handle both Tx and Rx operation. The following description is for both control and asynchronous device.

The driver uses IRAM as MLB device module Tx/Rx buffer. All the data transmission and reception between module and IRAM is handled by the MLB module DMA. The driver is responsible for configuring the buffer start and end pointer for the MLB module.

For reception, the driver uses a ring buffer to buffer the received packet for read. When a packet arrives, the MLB module puts the received packet into the IRAM Rx buffer, and notifies the driver by interrupt. The driver then copy the packet from the IRAM to one ring buffer node indicated by the write position, and updates the write position with the next empty node. Finally the packet reader application is notified, and it gets one packet from the node indicated by the read position of ring buffer. After the read completed, it updates the read position with the next available buffer node. There is no received packet in the ring buffer when the read and write position is the same.

For transmission, the driver writes the packet given by the writer application into the IRAM Tx buffer, updates the Tx status and sets MLB device module Tx buffer pointer to start transmission. After transmission completes, the driver is notified by interrupt and updates the Tx status to accept the next packet from the application.

The driver supports NON BLOCK I/O. User applications can poll to check if there are packets or exception events to read, and also they can check if a packet can be sent or not. If there are exception events, the application can call ioctl to get the event. The ioctl also provides the interface to configure the frame rate, device address and channel address.

33.2.3 Software Operation

The MLB driver provides a common interface to application.

- Packet read/write–BLOCK and NONBLOCK Packet I/O modes are supported. Only one packet can be read or written at once. The minimum read length must be greater or equal to the received packet length, meanwhile the write length must be shorter than 1024Bytes.
- Polling–The MLB driver provide polling interface which polls for three status, application can use select to get current I/O status:
  - Packet available for read (ready to read)
  - Driver is ready to send next packet (ready to write)
  - Exception event comes (ready to read)
- ioctl–MLB driver provides the following ioctl:
  - `MLB_SET_FPS`
    Argument type: unsigned int
    Set frame rate, the argument must be 256, 512 or 1024.
  - `MLB_GET_VER`
    Argument type: unsigned long
    Get MLB device module version, which is 0x02000202 by default on the i.MX35.
  - `MLB_SET_DEVADDR`
    Argument type: unsigned char
    Set MLB device address, which is used by the system channel MlbScan command.
  - `MLB_CHAN_SETADDR`
    Argument type: unsigned int
    Set the corresponding channel address [8:1] bits. This ioctl combines both tx and rx channel address, the argument format is: tx_ca[8:1] << 16 | rx_ca[8:1]
  - `MLB_CHAN_STARTUP`
    Startup the corresponding type of channel for transmit and reception.
  - `MLB_CHAN_SHUTDOWN`
    Shutdown the corresponding type of channel.
  - `MLB_CHAN_GETEVENT`
    Argument type: unsigned long
    Get exception event from MLB device module, the event is defined as a set of enumeration:
    `MLB_EVT_TX_PROTO_ERR_CUR`
    `MLB_EVT_TX_BRK_DETECT_CUR`
    `MLB_EVT_RX_PROTO_ERR_CUR`
    `MLB_EVT_RX_BRK_DETECT_CUR`

33.3 Driver Files

Table 33-1 lists the source file associated with the MLB driver that are found in the directory `<ltib_dir>/rpm/BUILD/linux/drivers/mxc/mlb/.`

Table 33-1. MLB Driver Source File List

File	Description
mxc_mlb.c	Source file for MLB driver
include/linux/mxc_mlb.h	Include file for MLB driver

33.4 Menu Configuration Options

To get to the MediaLB configuration, use the command `./ltib -c` when located in the `<ltib dir>`. In the screen, select **Configure Kernel**, exit, and a new screen appears. This option is available under:

- Device Drivers > MXC support drivers > MXC Media Local Bus Driver > MLB support.

Chapter 34
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.

34.1 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.

34.2 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 based platforms.

34.3 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.

34.4 Software Operation

34.4.1 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.

34.4.2 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.

34.4.3 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.

34.4.4 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).

34.4.5 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.

34.5 Requirements

The requirements of OProfile are as follows:

- Add Oprofile support with Cortex-A8 Event Monitor

34.6 Source Code Structure

Oprofile platform-specific source files are available in the directory:
`<ltib_dir>/rpm/BUILD/linux/arch/arm/oprofile/`

Table 34-1. 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
op_model_v7.c	Source file for ARM V7 (Cortex A8) Event Monitor Driver
op_model_v7.h	Header file for ARM V7 (Cortex A8) Event Monitor Driver

The generic kernel driver for Oprofile is located under `<ltib_dir>/rpm/BUILD/linux/drivers/oprofile/`

34.7 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)

34.8 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).

34.9 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.

34.10 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 34.7, “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.
```

Chapter 35
Frequently Asked Questions

35.1 Downloading a File

There are various ways to download files onto a Linux system. The following procedure gives instructions on how to do this through a serial download.

To download a file through the serial port using a Windows host system, follow these steps:

1. Make sure the Linux serial prompt goes to the Windows terminal. For more information about how to set this up, see the User Guide.
2. Make sure Linux boots to the serial prompt and log in using `root`
3. Type `rz` under the serial prompt at `/mnt/ramfs/root`
4. Under Hyper Terminal, click on Transfer > Send File > Browse... >, then go to the directory with the file to download.
5. Click on Open and then Send. The protocol should be `Zmodem with Crash Recovery`, which is the default.

This should start the downloading process. For the file transfer, the lrzsz package is required. Another way to transfer a file is to use FTP which makes the download much faster than through the serial port. To use FTP, the Ethernet interface has to be set up first.

35.2 Creating a JFFS2 Mount Point

To mount a pre-built JFFS2 file system onto the target, `mkfs.jffs2` can be used to generate the JFFS2 file system on the development system (the host) first and then mount it on the target. The following steps describe how to do this. If an empty JFFS2 file system is sufficient, then only step 2 is required.

1. Generate the JFFS2 file system under the host:

   Create a temporary directory on the host, for example `jffs2` under `/tmp` and then move all the files and directories to place inside the JFFS2 file system into the `jffs2` directory. Issue the following command from `/tmp`:

   ```
   mkfs.jffs2 -d jffs2 -o fs.jffs2 -e 0x20000 --pad=0x400000
   ```

   `jffs2` is the source directory. `-e`: erase block size. `--pad=0x400000` is to pad `0xff` up to 4 Mbytes. The output file is `fs.jffs2`.

   NOTE

   - Make sure the `fs.jffs2` file is within this size limit of 4 Mbyte.
   - Download the prebuilt version of the `mkfs.jffs2` from ftp://sources.redhat.com/pub/jffs2/mkfs.jffs2.

2. Mount the JFFS2 file system on the target system:

   The JFFS2 file system can be mounted on one of the MTD partitions. The partition table is set up in two ways: static and dynamic. If no RedBoot partition is created when Linux boots on the target, a static partition table is used from the MTD map driver source code (`mxc_nor.c` for example). Otherwise, the RedBoot partition is used instead of the static one.

   In most cases, it is more flexible to set up a partition in RedBoot for JFFS2 that can be used by Linux. To do this, use RedBoot to program (use `fis create`) the newly created JFFS2 image into the Flash on some unused space and then create a partition using `fis create`.

   The following example illustrates how to do this in more detail.

```
RedBoot> fis list
Name              FLASH addr  Mem addr    Length      Entry point
RedBoot           0xA0000000  0xA0000000  0x00040000  0x00000000
kernel            0xA0100000  0x00100000  0x00200000  0x00100000
root              0xA0300000  0x00300000  0x00D00000  0x00300000
jffs2             0xA1200000  0xA1200000  0x00200000  0xFFFFFFFF
FIS directory     0xA1FE0000  0xA1FE0000  0x0001F000  0x00000000
RedBoot config    0xA1FFF000  0xA1FFF000  0x00001000  0x00000000
```

   The above shows that a RedBoot partition called `jffs2` is created which contains the JFFS2 image inside the Flash. When booting Linux, the kernel is able to recognize the RedBoot partitions and create MTD partitions correspondingly when `CONFIG_MTD_REDBOOT_PARTS=y` is in the kernel configuration (it is the default configuration on all i.MX platforms). With the above example, the Linux kernel boot message shows:

```
Searching for RedBoot partition table in phys_mapped_flash at offset0x1fe0000
6 RedBoot partitions found on MTD device phys_mapped_flash
Creating 6 MTD partitions on "phys_mapped_flash":
0x00000000-0x00040000 : "RedBoot"
0x00100000-0x00300000 : "kernel"
0x00300000-0x01000000 : "root"
0x01200000-0x01400000 : "jffs2"
0x01fe0000-0x01fff000 : "FIS directory"
```

   The JFFS2 is the fourth MTD partition under Linux in this case. To mount this MTD partition after booting Linux, type:

```
cd /tmp
mkdir jffs2
mount –t jffs2 /dev/mtdblock/3 /tmp/jffs2
```

This mounts `/dev/mtdblock/3` to the `/tmp/jffs2` directory as the JFFS2 file system (directory name can be something other than `jffs2`). The static partition method uses the partition table defined in the NOR MTD map driver source code. The way to mount it is very similar to what is described above.

35.3 NFS Mounting Root File System

1. Assuming the root file system is under`/tmp/fs`, modify the `/etc/exports` file on the Linux host by adding the following line:

```
/tmp/fs *(rw,no_root_squash)
```

2. Make sure the NFS service is started on the Linux host machine. To start it on the host machine, issue:

```
service nfs start
```

   InstallNFS RPM if not already installed.

3. To boot with a NFS mounted file system under RedBoot, use the following command:

```
exec –b 0x100000 –l 0x200000 –c "noinitrd console=tty0 console=ttymxc1 root=/dev/nfs
        nfsroot=1.1.1.1:/tmp/fs rw init=/linuxrc ip=dhcp"
```

The above example assumes the Linux host IP address is `1.1.1.1`. This needs to be modified in the command line used.

NOTE

The `/etc/fstab` mounts several ramfs drives in places like `/root` and `/mnt` (see `/etc/fstab` for the complete list). This is desirable when the root file system is burned into Flash as it provides some read/write disk space. However, this causes problems when doing an NFS mount of the root file system because any files added or modified on these directories exists only in RAM, not on the NFS mount. In addition, these drives hide any contents of their respective directories on the host NFS mount. Not all directories of the root file system are affected by this, only the ones that fstab loads a ramfs on top of. This can be fixed by editing `/etc/fstab` and deleting or commenting out all lines that have the word “ramfs” in them.

35.4 Error: NAND MTD Driver Flash Erase Failure

The NAND MTD driver may report an error while erasing/writing the NAND Flash. One possible reason for this failure is the NAND Flash is write protected.

35.5 Error: NAND MTD Driver Attempt to Erase a Bad Block

This error indicates that a block marked as bad is attempting to be erased, which the MTD layer does not allow. Sometimes many or all the blocks of the NAND Flash are reported as bad. This could be because garbage was written to the block OOB area, possibly during testing of the board. To overcome this, the Flash must be erased at a low level, bypassing the MTD layer. For this, the NAND driver needs to be recompiled by enabling MXC_NAND_LOW_LEVEL_ERASE definition in the `mxc_nd.c` file. This produces an MXC NAND driver, which upon loading, erases the entire NAND Flash during initialization. Be careful when using this feature. Loading the NAND driver causes the entire NAND device to be erased at a low-level, without obeying the manufacturer-marked bad block information.

35.6 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.

35.7 How to Make Software Workable when JTAG is Attached

When the JTAG is attached, add option `jtag=on` in the command line when launching the kernel.