i.MX31 PDK 1.5 Linux

Reference Manual

Document Number: 926-77210
Rev. 1.5
02/2009

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

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

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

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

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

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

Information in this document is provided solely to enable system and software implementers to use Freescale Semiconductor products. There are no express or implied copyright licenses granted hereunder to design or fabricate any integrated circuits or integrated circuits based on the information in this document.

Freescale Semiconductor reserves the right to make changes without further notice to any products herein. Freescale Semiconductor makes no warranty, representation or guarantee regarding the suitability of its products for any particular purpose, nor does Freescale Semiconductor assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation consequential or incidental damages. "Typical" parameters that may be provided in Freescale Semiconductor data sheets and/or specifications can and do vary in different applications and actual performance may vary over time. All operating parameters, including "Typicals", must be validated for each customer application by customer's technical experts. Freescale Semiconductor does not convey any license under its patent rights nor the rights of others. Freescale Semiconductor products are not designed, intended, or authorized for use as components in systems intended for surgical implant into the body, or other applications intended to support or sustain life, or for any other application in which the failure of the Freescale Semiconductor product could create a situation where personal injury or death may occur. Should Buyer purchase or use Freescale Semiconductor products for any such unintended or unauthorized application, Buyer shall indemnify and hold Freescale Semiconductor and its officers, employees, subsidiaries, affiliates, and distributors harmless against all claims, costs, damages, and expenses, and reasonable attorney fees arising out of, directly or indirectly, any claim of personal injury or death associated with such unintended or unauthorized use, even if such claim alleges that Freescale Semiconductor was negligent regarding the design or manufacture of the part.

Freescale and the Freescale logo are trademarks or registered trademarks of Freescale Semiconductor, Inc. in the U.S. and other countries. All other product or service names are the property of their respective owners. ARM is the registered trademarks of ARM Limited. The ARM logo is a registered trademark of ARM Ltd.

© Freescale Semiconductor, Inc. 2008-2009. All rights reserved.

Contents

Contents

Chapter 4
Machine Specific Layer (MSL)

Contents

Contents

Chapter 7
PMIC Audio Driver

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

Contents

About This Book

The Linux board support package (BSP) represents a porting of the Linux operating system (OS) to the i.MX processors and to their associated reference boards. The BSP supports many of the hardware features on the platforms, as well as most of the Linux OS features 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 understanding of the Linux 2.6 kernel internals and driver models. An understanding of the i.MX processors is also required.

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.

Definitions and Acronyms (Continued)

Term	Definition
CPU	Central Processing Unit—generic term used to describe a processing core.
CRC	Cyclic Redundancy Check—Bit error protection method for data communication.
CSI	Camera Sensor Interface
DMA	Direct Memory Access—an independent block that can initiate memory-to-memory data transfers.
DRAM	Dynamic Random Access Memory
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 or solutions
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, but where erasing can only be done in blocks or the entire chip.
Flash path	Path within ROM bootstrap pointing to an executable Flash application.
Flush	A 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 will produce 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.

Definitions and Acronyms (Continued)

Term	Definition
ISR	Interrupt Service Routine.
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—a 16-bit peripheral that can be 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	There are 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 quality 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 in various ways to create other colors. The abbreviation RGB come from the three primary colors in additive light models.

Definitions and Acronyms (Continued)

Term	Definition
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 you place, the lighter the picture gets. PNG is the best known image format that uses the RGBA color space.
RNGA	Random Number Generator Accelerator—a security hardware module that produces 32-bit pseudo random numbers as part of the security module.
ROM	Read Only Memory
ROM bootstrap	Internal boot code encompassing the main boot flow as well as exception vectors.
RTIC	Real-time integrity checker—a security hardware module
SCC	SeCurity Controller—a security hardware module
SDMA	Smart Direct Memory Access
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System on a Chip
SPBA	Shared Peripheral Bus Arbiter—a three-to-one IP-Bus arbiter, with a resource-locking mechanism.
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—this module provides 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 12Mb/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.MX31 PDK Quick Start Guide for Linux*
- *BSP API Document (BSP Doxygen Code Documentation)*
- *i.MX31 PDK Linux User's Guide*
- *i.MX31 PDK Hardware User's Guide*
- *MCIMX31 Multimedia Applications Processors Reference Manual*, (MCIMX31RM)
- [KERN] *Linux kernel coding style* by Linus Torvalds. This is included in Linux distributions as the file Documentation/CodingStyle
- [WSAS] *WSAS Coding Conventions*, version 0.4
- [ASM] *WSAS Assembly Code Conventions*
- [DOXY] *WSAS Guidelines for Writing Doxygen Comments*

Chapter 1
Introduction

The i.MX family Linux board support package (BSP) supports the Linux operating system (OS) on the following processors:

- i.MX31 Applications Processor

NOTE

The family of all i.MX processors is known as the i.MX platforms. You will see this term used in sections that apply to any of these application processors.

As the name BSP implies, the purpose of this software package is to support Linux on the i.MX family of integrated circuits (ICs) and their associated platforms (3-Stack board). It provides the software necessary to interface the standard open-source Linux kernel to the i.MX hardware. The goal of this port 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, GUI components, 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 2.6.26 of the Linux kernel from the official Linux kernel web site (http://www.kernel.org). It is enhanced with features provided by Freescale.

1.2 Features

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

Table 1-1. Supported Features

Features	Description	Chapter Source	Applicable Platform
Machine Specific Layer			
MSL	MSL (Machine Specific Layer) 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 ARM11 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 4, “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 5, “Smart Direct Memory Access (SDMA) API”	i.MX31

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
Power Management IC (PMIC) Drivers			
PMIC Protocol	The PMIC protocol driver interfaces to the IC through the SPI driver. It manages the hardware interrupt from the IC and provides services for all IC client drivers. It exposes consistent APIs used by each IC client driver to access the IC component.	Chapter 6, “PMIC Protocol Driver”	i.MX31
PMIC Audio	The audio driver is a client of the IC protocol driver. It provides services for audio control of the IC and has the following features: • Supports configuration of the PMIC's Stereo DAC and Voice CODEC, including the 13-bit Voice CODEC and both narrow and wide band sampling and the 16-bit Stereo DAC with multiple sample rates. • Supports all input and output audio channels. • Provides a custom API to set volume, balance, mixer, and gain amplifiers. • Reports an event for microphone bias detected. • Provides a custom API for the configuration of the PMIC-side of the SSI audio bus interface for operating in network mode. • Used by the higher level Alsa Driver.	Chapter 7, “PMIC Audio Driver”	i.MX31
PMIC Digitizer	This driver is a client of the PMIC's protocol driver. It provides services for the digitizer controlled by the PMIC. It supports the following features: • Supports all types of digitizer input converters. • Starts the digitizer converter. • Reports an event when converted. • Supports the monitor function of the PMIC's digitizer. • Used by touch screen component of input sub-system and battery driver for charger/battery current/voltage measurement.	Chapter 8, “PMIC Digitizer Driver”	i.MX31
PMIC RTC	The PMIC RTC for Linux provides access to the PMIC’s RTC control circuits. It has the following features • Real-time clock control. • Alarm events.	Chapter 13, “PMIC Real Time Clock (RTC)”	i.MX31
PMIC Power Management	This driver is a client of the PMIC's protocol driver. It provides services for power management control through PMIC. It supports the following features: • Controls all ON/OFF switches. • Controls all voltage regulators.	Chapter 9, “PMIC Power Management Driver”	i.MX31

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
PMIC Connectivity	This driver is a client of the PMIC's protocol driver. It provides services for USB OTG and RS-232 connectivity controlled by PMIC. It supports the following features: • Supports an RS-232 transceiver in either DTE or DCE modes with full hardware flow control. • Supports a USB OTG transceiver with device insertion/removal detection capabilities and connection configuration using the Host Negotiation Protocol.	Chapter 10, "PMIC Connectivity Driver"	i.MX31
PMIC Battery	This component is a client of the PMIC's protocol driver. It provides services for battery control. It provides an API for battery control management.	Chapter 11, "PMIC Battery Driver"	i.MX31
PMIC Light	This driver is a client of PMIC's protocol driver. It supports the following features: • Supports all modes of LED control • Supports backlight control (when the back light is connected to the PMIC).	Chapter 12, "PMIC Light Driver"	i.MX31
Power Management Drivers			
Low-level PM drivers (including DVFS)	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. This driver implements DVFS or DFS techniques, depending on the platform, and low-power modes.	Chapter 14, "i.MX31 Low-level Power Management Driver"	i.MX31
DVFS	The Linux Dynamic Voltage Frequency Scaling (DVFS) device driver monitors the current operating point, using four reference circuits that test the IC processing under the current ambient temperature. The software module is comprised of a Linux driver that allows privileged users to control and monitor the DVFS operation. The DVFS Linux driver is designed as a character driver.	Chapter 15, "Dynamic Voltage Frequency Scaling (DVFS) Driver"	i.MX31
DPTC	The Dynamic Process Temperature Compensation (DPTC) Driver manages the DPTC power management technique. This technique reduces power consumption by adjusting the supply voltages according to the specific process case, chip fabrication, and ambient temperature.	Chapter 16, "Dynamic Process and Temperature Compensation (DPTC) Driver"	i.MX31
Multimedia Drivers			
TV-OUT	TV-OUT is a television encoder device that encodes video signals and generates synchronization signals for a given television standard.	Chapter 17, "CH7024 TV Encoder (TV-Out) Driver"	i.MX31

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
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 framebuffer driver, a V4L2 device driver, and low-level IPU drivers.	Chapter 18, “Image Processing Unit (IPU) Drivers”	i.MX31
GPU	The Graphics Processing Unit (GPU) Driver is a licensed core from Imagination Technologies. The GPU is a graphics accelerator. GPU is used synonymously with the names MBX-Lite and MBX.	Chapter 19, “MBX Driver”	i.MX31
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.	Section 18.2.3.3, “V4L2 Output Device”	All
V4L2 Capture	The Video for Linux 2 (V4L2) capture device includes two interfaces: the capture interface and the overlay interface. The capture interface uses IPU pre-processing ENC channels to record the YCrCb video stream. The overlay interface uses the IPU pre-processing VF channels to display the preview video to the SDC foreground panel without ARM processor interaction.	Section 18.2.3.1, “V4L2 Capture Device”	i.MX31
Hantro VGA Video Encoder	An integrated hardware VGA video encoder from Hantro. The encoder is operated through an application programming interface.	Chapter 20, “Hantro VGA Video Encoder Driver”	i.MX31
Camera (OV2640)	The OV2640 Camera driver is designed under Linux V4L2 architecture. It implements V4L2 capture interface.	Chapter 21, “OmniVision Camera Driver (OV2640)”	i.MX31
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 using the audio components provided by Freescale’s PMIC chips. 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, s/w mixing, snooping, and so on. The ASoc Sound driver supports stereo codec playback and capture through SSI.	Chapter 22, “Advanced Linux Sound Architecture (ALSA) Sound Driver with PMIC Hardware Support”	i.MX31

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
AudMux	The low level Digital Audio Multiplexer (AUDMUX) driver provides a custom, kernel-space API to the AUDMUX module. It supports all of the features of the hardware module. It provides runtime audio path configuration.	Chapter 23, “Digital Audio Multiplexer (AUDMUX) Driver”	i.MX31
SSI	The low level synchronous serial interface (SSI) driver provides a custom, kernel-space API to the SSI modules. It supports all of the features of the hardware modules including enabling/disabling of DMA request events.	Chapter 24, “Synchronous Serial Interface (SSI) Driver”	i.MX31
Memory Drivers			
NOR MTD	The NOR MTD driver is board-specific as it depends on the actual NOR Flash chip (Common Flash Interface or CFI-compliant) on the board and can have file systems, such as CRAMFS and JFFS2 on top of it. The driver implementation supports the lowest level operations on the Flash chip, such as read, write and erase. The NOR MTD supports XIP on Flash devices.	Chapter 41, “NOR Flash Memory Technology Device (MTD) Driver”	i.MX31
NAND MTD	The NAND MTD driver interfaces with the integrated NAND controller. It can support various file systems, such as CRAMFS and JFFS2. 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 25, “NAND Flash Memory Technology Device (MTD) Driver”	i.MX31
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. It supports up to an 8 x 8 external key pad matrix of single poll switches.	Chapter 26, “Low-Level Keypad Driver”	i.MX31
Networking Drivers			
LAN9217 Ethernet	The SMSC LAN9217 Ethernet driver interfaces SMSC LAN9217-specific functions with the standard Linux kernel network module.	Chapter 27, “SMSC LAN9217 Ethernet Driver”	i.MX31 3-Stack
WLAN	The WLAN driver is used to drive the APM6628 module to implement Wi-Fi functionality.	Chapter 28, “WLAN Driver”	i.MX31 3-Stack

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
Security Drivers			
SCC/SCC2	The Security Controller (SCC) is a part of the Freescale Platform Independent Security Architecture (PISA). This driver is comprised of two modules; the Secure RAM Module and the Secure Monitor Module.The Secure RAM module provides a secure way of storing sensitive data in on-chip and off-chip RAM memory. On-chip data can be cleared if necessary to prevent un-authorized access. Off-chip data is stored in encrypted form using an encryption key that is unique to each device and is accessible only through the Secure RAM module. The SCC module will only be accessible by ARM11.	Chapter 29, “Security Drivers”	i.MX31
RNGA/RNGC	The Random Number Generator Accelerator (RNGA) module is a digital integrated circuit capable of generating 32-bit random numbers. It is designed to comply with FIPS-140 standards for randomness and non-determinism. The oscillators with their unknown frequencies provide the required entropy needed to create random data. An Entropy register is provided which serves the purpose for seeding the Random number generator.	Section 29.1.13, “Random Number Generator Accelerator (RNGA)” and Section 29.2.2, “Random Number Generator Accelerator (RNGA)”	i.MX31
Bus Drivers			
I2C	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 400kbps • Standard I^2C master mode • Power management features by suspending and resuming I^2C.	Chapter 30, “Inter-IC (I2C) Driver”	i.MX31
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 32, “Configurable Serial Peripheral Interface (CSPI) Driver”	i.MX31
MMC/SD/SDIO - SDHC	The MMC/SD/SDIO Host driver implements a standard Linux driver interface to the MMC/Secure Digital Host Controller (SDHC). The MMC driver complies to the MMC specification version 4.1, SD version 1.10 and SDIO version 1.10.	Chapter 33, “MMC/SD/SDIO Host Driver”	i.MX31

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
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 34, "Universal Asynchronous Receiver/Transmitter (UART) Driver"	i.MX31
General Drivers			
USB	The USB driver implements a standard Linux driver interface to the ARC USB-HSOTG controller.	Chapter 35, "ARC USB driver"	i.MX31
Bluetooth	The Bluetooth driver provides synchronous and asynchronous wireless connection among multiple devices.	Chapter 36, "Bluetooth Driver"	i.MX31 3-Stack
ATA	The ATA module is an AT attachment host interface. Its main use is to interface with hard disk devices. The ATA driver is compliant with the ATA-6 standard, and supports the following protocols: • PIO mode 0, 1, 2, 3, and 4 • multiword DMA mode 0, 1, and 2 • Ultra DMA mode 0, 1, 2, 3, and 4and 3 with bus clocks of 50MHz or higher • Ultra DMA mode 5 with bus clock of 80MHz or higher. It supports the IDE and LibATA interfaces.	Chapter 37, "ATA Driver"	i.MX31
RTC	This is the integrated Real Time Clock (RTC) module. The RTC is used to keep the time and date while the system is turned off. Additionally, it provides the PIE (periodic interrupt at a specific frequency) and AIE (Wake up the system by providing an alarm) features.	Chapter 38, "Real Time Clock (RTC) Driver"	i.MX31
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. • The WDOG module generates a reset signal if it is enabled but not serviced within a predefined time-out value. • The WDOG module does not generate a reset signal if it is serviced within a predefined time-out value.	Chapter 39, "Watchdog (WDOG) Driver"	i.MX31
FM (Si4702)	The FM (Si4702) driver provides the interfaces to control Si4702 chips.	Chapter 40, "FM Driver"	i.MX31 3-Stack

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
MMA7450L Accelerometer	The MMA7450L is a feature-rich accelerometer device with a flexible programming interface exposed to the software.	Chapter 41, “MMA7450L Accelerometer Driver”	i.MX31 3-Stack
GPS	The communications driver provides a serial interface to the core driver and communicates with the external GPS chip set.	Chapter 42, “Global Positioning System (GPS) Driver”	i.MX31 3-Stack
Bootloaders			
RedBoot	RedBoot is an open source boot firmware based on the eCos Hardware Abstraction Layer. It was designed to be very portable, extensible, and configurable.		i.MX31
GUI			
Qt/E	Qt/E is a Graphical User Interface supported by the Linux BSP.		i.MX31
Tools			
OProfile	OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.	Chapter 43, “OProfile”	All

Chapter 2
Running Linux on the Hardware Boards

This chapter explains how to build and test this release of the i.MX Linux BSP on a hardware board, install and configure the development tools, and install and configure the components of the i.MX Linux BSP release.

Specifically, this chapter describes how to:

- Exercise the binary components of the i.MX Linux BSP release using a reference board
- Build the i.MX Linux BSP release from source files
- Debug the i.MX Linux BSP kernel and drivers

Exercising the binary release components covers the following:

— Using RedBoot to load a Linux image and a Linux file system onto a reference board

— Using RedBoot to boot the Linux OS on a reference board

Building the release from source files covers the following:

— Installing the source files for the Linux kernel and the i.MX Linux BSP onto a Linux host

— Installing the ARM cross-compiler tool chain onto a Linux host (Done automatically with Linux Target Image Builder)

— Building the i.MX Linux BSP image file for an ARM target

— Building the CRAMFS or JFFS2 root file system

Debugging the kernel and the drivers covers the following:

— Configuring the ARM RealView ICE interface unit

— Downloading RedBoot and Linux images using RealView ICE

Each section in this chapter includes a table describing the resources that must be obtained to install one of the development tools. For each resource, one of the table columns describes where the resource can be obtained.

2.1 Running the i.MX Linux BSP

This section provides the procedures for booting the Linux OS on a hardware board.

2.1.1 Preparation – Board Setup

Table 2-1 lists the hardware resources needed for each supported platform.

Table 2-1. Hardware Resources Needed

Resource	Description
i.MX31 PDK (3-stack)	i.MX31 PDK containing a board, serial cable, and other accessories

2.1.2 Terminal Console

HyperTerminal (on Microsoft Windows) or Minicom (on Linux) on your PC can be used to view console debug messages. Set the terminal to 115200 bps, 8 data bits, parity None, 1 stop bit, and no flow control.

The serial port is labeled as UART-DCE on the i.MX31 PDK board. Attach one end of the serial cable that comes with the development kit to the serial port on the board and the other end of the cable to a serial port of your PC.

2.1.3 Programming RedBoot into Flash

Running Linux requires a boot loader, a Linux kernel, and a root file system. The boot loader should be stored in Flash memory. The kernel should run directly from SDRAM memory. The file system can be stored in the Flash as MTD mounted root or can be stored remotely on a separate machine and be mounted through NFS.

RedBoot is used as the bootloader to load the Linux kernel. Make sure to install the version of RedBoot that is included in this Linux BSP release even if RedBoot was previously stored in the Flash memory of your board. Otherwise the Linux kernel may fail to boot.

Table 2-2. Resources Needed to Program RedBoot into Flash

Resource	Description	Source
`redboot_200904.zip`	The RedBoot release ZIP file that contains the RedBoot binaries for all the Freescale processors and release documentation (the PDF files for each platform). Refer to these documents for instructions on installing and using RedBoot.	Unpacked from tarball file

Unzip the RedBoot release ZIP file and find the PDF document that matches your platform inside the documentation folder. This document provides instructions on how to program RedBoot into Flash.

Also refer to this document for instructions on how to perform the following actions:

- Set up dip switches for different boot modes.
- Set up ARM RealView tools for the reference board including RealView ICE firmware upgrade.

After RedBoot is successfully programmed into Flash, change the settings for the dip switches to external boot mode to boot from Flash. Reset the board, and the RedBoot prompt should come up. Note that if the RedBoot prompt does not show up the first time after power on or pressing the `Reset` button, press "`Ctrl + c`" multiple times.

NOTES

See the *i.MX31 PDK Hardware User's Guide* for more information about the hardware setup and boot modes.

Ensure that `bootp` is enabled if you plan to obtain an IP address through DHCP.

Press the "Reset" button on the board if the RedBoot prompt comes up but no IP address was obtained through DHCP.

2.1.4 Running Linux

To run Linux, you need a Linux kernel and a root file system.

2.1.4.1 Downloading the Linux Kernel and File System to SDRAM

Table 2-3. Resources Needed to Download the Kernel and Root File System to SDRAM

Resource	Description	Source
`zImage` and `rootfs.ext2.gz`	Binary Linux Kernel Image and ext2 image of the root file system - QTEmbedded/Qtopia. Select the Kernel Image based on your platform.	Unpacked from the tarball.

The Linux kernel and Linux file system can be downloaded to SDRAM using either RedBoot or the RealView ICE unit. Refer to the PDF files in the RedBoot release ZIP file for instructions on how to set up RealView ICE.

The following steps explain how to download a Linux Kernel Image or the root filesystem using Ethernet or the serial port.

2.1.4.1.1 Downloading the Linux Kernel and File System with Ethernet Download from RedBoot

To download the Linux kernel and file system with Ethernet download from RedBoot:

1. Start the `tftp` server. Either copy the files to be downloaded to the directory pointed to by the server or modify the server settings to point to the directory where the files to be downloaded reside.

   Make sure `bootp` is enabled on the platform to obtain an IP address through DHCP or program a static IP address. Refer to the RedBoot documentation for instructions.

2. Use the RedBoot `fconfig` command to configure RedBoot with your `tftp` server IP address and reset the board for the changes to take effect

```
fconfig bootp_server_ip 10.81.68.96
```

Some of the very early boards may not have a valid MAC address in the EEPROM. To verify that, type "`setmac`" command under RedBoot. If the returned values are all `0xFF`s, then the MAC address needs to be re-programmed. Contact the board vendor to obtain valid addresses. To reconfigure the MAC address in the EEPROM, use the RedBoot "`setmac`" command. After that, reset the board.

3. Download the Linux kernel binary to SDRAM using the command

```
load -r -b 0x100000 <tftp folder>/zImage
```

4. Follow the instructions in Section 2.1.4.2, "Programming Linux Kernel and Filesystem to Flash Memory" to program the Linux kernel to Flash memory
5. Download the root file system to SDRAM using the command:

```
load -r -b 0x100000 rootfs.cramfs
```

NOTE

The `rootfs.cramfs` file is not shipped with the tar ball. You must create it prior to downloading to SDRAM. To do so, see Section 2.3.2.2, "Creating a CRAMFS Root Filesystem." Copy the CRAMFS file to TFTP directory and download it to SDRAM using the command mentioned above.

6. Follow the instructions in Section 2.1.4.2, "Programming Linux Kernel and Filesystem to Flash Memory" to program the file system to Flash memory.

2.1.4.1.2 Downloading the Linux Kernel and File System with Serial Download from RedBoot

To download the Linux kernel and file system with serial download from RedBoot:

1. Issue the following command under RedBoot prompt to download an image using a serial download:

```
load -r -b 0x100000 -m xmodem
```

2. RedBoot now is ready to receive data and prints out the character "`C`" continuously. To send a file using HyperTerminal, click **Transfer** > **Send File** > **Xmodem** (under **Protocol**) > **Browse**, choose the file to download and then click **Send**. Ymodem can also be chosen for download.
3. Follow the instructions in Section 2.1.4.2, "Programming Linux Kernel and Filesystem to Flash Memory" to program the downloaded image to Flash memory.

2.1.4.1.3 Downloading the Linux Kernel and File System with RealView ICE

To download the Linux kernel and file system with RealView ICE:

1. Stop the RedBoot execution from the RVD.
2. Type the following command into the RVD command window, to download the binary file into SDRAM with the RVD command:

```
readfile,raw,gui "<PATH TO THE KERNEL IMAGE>/zImage"=0x100000
```

The command downloads the Image file into the 0x100000 memory location. Note that this command must be modified with the proper path.

3. Resume RedBoot from RVD by typing the command "`go`" in the RVD command window.
4. Follow the instructions in Section 2.1.4.2, "Programming Linux Kernel and Filesystem to Flash Memory" to program the downloaded image to Flash memory.

2.1.4.2 Programming Linux Kernel and Filesystem to Flash Memory

To view the flashing procedures for the i.MX boards, see the Linux User's Guide for your board; for example, *i.MX 3-Stack Linux SDK User's Guide*.

2.1.5 Booting Linux

To boot Linux, issue the following RedBoot commands:

1. Run images from Downloaded NAND flash:

```
fis load kernel

exec -c "noinitrd console=ttymxc0,115200 root=/dev/mtdblock2 rw rootfstype=jffs2
ip=dhcp"
```

2. Run images from NFS

```
load -r -b 0x100000 zImage

exec –c "noinitrd console=ttymxc0 root=/dev/nfsroot rootfstype=nfsroot nfsroot=<the
IP address of the host machine>:/tools/rootfs rw ip=dhcp"
```

NOTES

For specific instructions on how to boot Linux on your board, refer to the Linux User's Guide for your platform. If the `fis load` command fails, try one of the following commands:

```
fis load -b 0x100000 kernel
```

OR

```
fis del kernel
```

Then re-program the kernel image using the instructions in the previous section.

The Linux prompt should appear on the terminal. The Linux logo should appear on the LCD screen. Be aware that during kernel testing, after downloading your kernel image to SDRAM, Flashing is not necessary. In this case, simply issue the `exec` command to launch the kernel.

RedBoot also allows passing command line options to the Linux kernel so that the default boot command line options built into the kernel can be overridden. The following is an example (note the new command line is in the quotes with the `-c` option):

```
exec -c "root=/dev/mtdblock2 rw"
```

A login prompt is displayed on the terminal console:

```
localhost login:
```

Use "root" as user name to log in (no password is needed). Once logged in, you can issue console commands to the sh shell.

If Qt/Embedded root file system is used, type the following command to start the Qtopia suite of applications:

```
# /etc/rc.d/init.d/qtopia
```

The i.MX LCD screen displays the *Welcome to Qtopia* screen with the instruction *Tap anywhere on the screen to continue*.

2.2 Exchanging Files with the i.MX Linux BSP

The files from your desktop PC can be sent to the i.MX Linux BSP running on the board using the terminal's Zmodem protocol. It is also possible to use Zmodem to send files in the opposite direction; that is, from the board to your desktop PC. If the Ethernet interface is enabled on the board, FTP can also be used for file transfers.

2.2.1 Sending Files to the i.MX Linux BSP

To send files to the BSP:

1. After the Linux BSP has booted up and you have logged in, type the following command in the Linux BSP console:
   ```
   # rz
   ```
   The Linux BSP is waiting to receive a file.
2. In HyperTerminal, select **Send File**... from the Transfer menu.
3. Choose **Zmodem** (with or without crash recovery).
4. Select the desktop file you want to transfer and click **Send**. A progress bar is displayed for the transfer.

Once the transfer is complete, the file is available in the current directory of the console session. If you use a read-only CRAMFS file system, you can only send files to /mnt/ramfs/root and its subdirectories.

2.2.2 Sending Files to the Desktop PC

To send files to the desktop PC:

1. In HyperTerminal, select **Receive File**... from the Transfer menu.
2. Select the desktop PC directory where you want the file to be placed.
3. Click **Receive** and then click **Cancel** and close the notification dialog.

   These steps set up the default receive directory. If these steps are skipped, then files are stored in you default directory which is most likely your home directory.
4. Type the following command in the Linux BSP console:
   ```
   # sz <filename>
   ```
   After the transfer is complete, the file is available in the destination directory on your desktop PC.

2.2.3 Exchanging Files with the Desktop PC using Ethernet

The boards have one Ethernet interface. To assign an IP address (for example:10.10.10.10) to this interface, issue the following command:

```
ifconfig eth0 10.10.10.10
```

This IP address must be a valid address on your network. After this, you should be able to ping other machines within the same subnet. To configure other parameters, such as the netmask and gateway, type "`ifconfig --help`" for the proper instructions. Once you have done this, you can use the FTP protocol tools such as ftpget, ftpput from your shell prompt of the board to send and receive files over the network from another `FTP` server.

NOTE

To use Ethernet under the Linux kernel, a valid Ethernet MAC address must be programmed into the EEPROM on the base board. To do this, use the `setmac` command under RedBoot.

2.3 Building the i.MX Linux BSP from Source

The i.MX Linux BSP is built on a Linux host computer using Linux Target Image Builder (LTIB), which is a tools framework used to manage, configure, extend, and build Linux software elements to easily build a Linux target image and a root filesystem.

Note that LTIB also runs on other Linux Distributions, such as Fedora Core or Suse distributions on an x86 PC running the Linux OS.

This section describes the build procedures for RedHat 9.0 Linux. This section provides instructions for a `bash` shell; certain shell commands may not work if you are using a different shell.

This BSP operates with LTIB running on a host development system with the following:

- Ethernet card
- Serial port
- 1 Gbyte of free disk space
- NFS Server
- TFTP Server
- rsync
- Perl

2.3.1 The GNU Tool Chain

The kernel for the i.MX Linux BSP is configured and built using cross development tools. The cross development tools run on the Linux distribution of your host computer, but build ARM Linux binaries and executables. These toolchains are installed during the LTIB install procedure described in the section below.

2.3.2 Installing the BSP

You should follow the steps below to install LTIB on host machine. Bypassing the install script leads to compile problems because LTIB will not be able to find the source packages it needs.

To install the BSP:

1. If the source package is `.iso`, copy the .iso file to your host machine, and as root, enter the following command:

   ```
   mount –o loop <target-bsp.iso> <mount point>
   ```

   OR

   If the source package is a `.tgz`, rather than an `.iso`, copy the `target tar.gz` file to your host machine and extract it:

   ```
   tar -zxf <target-bsp.tgz>
   ```

2. As a non-root user, install the LTIB:

   ```
   <unpacked/mounted target-bsp>/install
   ```

   The script needs you to accept the License Agreement and to have the correct permissions for the install path where the `ltib` directory will be located. The install script also copies source and patches for the kernel and the root filesystem from the `BSPs Common/pkgs` folder to an `/opt/freescale/pkgs` folder.

   There are no uninstall scripts. To uninstall LTIB, remove the `/opt/freescale/pkgs`, `/opt/freescale/ltib` and `<install_path>/ltib` directories manually.

   NOTE

   To rebuild Qtopia and tslib packages from source, several packages must be installed previously on your host. The exact package names may vary, depending on your Linux distro.

   - zlib:
     rpm-based distros: install zlib and zlib-devel
     debian-based distros: install zlib and zlib-dev
   - libuuid:
     rpm-based distros: install e2fsprogs and e2fsprogs-devel
     debian-based distros: install libuuid and uuid-dev
   - libjpeg:
     rpm-based distros: install libjpeg
     debian-based distros: install libjpeg and libjpeg-dev
   - libpng:
     rpm-based distros: install libpng and libpng-devel
     debian-based distros: libpng and libpng-dev

   For additional information, please visit the TrollTech website at:
   http://doc.trolltech.com/qtopia2.1/html/qtopia-dependencies.html

2.3.2.1 Running LTIB

LTIB requires that the environmental variable KBUILD_OUTPUT not be set. Also, note that if you run LTIB as root, it will cause compilation errors.

```
unset KBUILD_OUTPUT
```

To run LTIB, change to the directory into which you installed it and run `./ltib`.

```
cd <install_path>/ltib
./ltib
```

The first time LTIB runs on a machine, a number of host packages are built and installed that support LTIB. The toolchains also are installed. This may take a few minutes.

LTIB may provide messages about the settings in your host machines' `sudoers` file, giving specific directions on how to modify your `sudoers` file. This makes it possible to run `rpm` with root privileges, which LTIB needs to do.

LTIB needs `rpm-build` installed on the host machine.

Also, LTIB requires that the directory that it uses for its cache of package source and patches be on the same machine as the `ltib` directory, not mounted by `nfs`. This cache is called the local package pool (LPP) and is set in the `ltib/.ltibrc` file. The default is `/opt/freescale/pkgs`. If `/opt` is an `nfs` mount, edit the `.ltibrc` file and change the LPP path to something on the same machine as the `ltib` directory.

NOTE

If the kernel source package (linux-2.6.26.tar.bz2) is not included in the release package, you must provide this package for LTIB to build correctly. Go to www.kernel.org and look for the file `linux-2.6.26.tar.bz2`, choose one mirror and download the file. After it is downloaded, copy it to `/opt/freescale/pkgs/` in your host. You can create a checksum to validate that the package is the one required by LTIB, using the following command:

```
$md5sum linux-2.6.26.tar.bz2
```

You should see the following output:

```
3f23ad4b69d0a552042d
```

Once LTIB is past the installation phase, it pops up a configuration menu for selecting a platform. Select **Freescale iMX reference boards** as the platform choice. Exit after saving changes.

Another menu will pop up to select the board. Use the arrow keys to select <Platform type> and <Packages Profiles>. The default profile is a minimal rootfs. Save your changes and exit. Table 2-4 lists profile options and corresponding profile files.

Table 2-4. Profile Options and Corresponding Files

Profile Options	Corresponding Profile files
Use packages in preconfig (Min profile)	`None`
Minimum bootable root file system	`config/platform/imx/min.profile`

Table 2-4. Profile Options and Corresponding Files

Profile Options	Corresponding Profile files
Test and Development packages	config/platform/imx/dev.profile
Qtopia 2 Release packages	config/platform/imx/release.profile
Qtopia 4 Release package	config/platform/imx/release_qt4.profile
FSL GUI release package	config/platform/imx/release_fsl.profile
All supported packages	config/platform/imx/max.profile

The next configuration menu allows you to select the kernel source to use in building the kernel, either the patch that came from the release (default) or local kernel sources. If building with local kernel sources, `config` options are displayed in the menu to allow you to specify the absolute path to the kernel source tree on your host machine.

By default the kernel is built at `/rootfs/boot/zImage` relative to the LTIB installation directory. Another configuration option allows changing the kernel build path to be something other than that default. To run the kernel `menuconfig` before building the kernel, select the **Configure the kernel** option.

The “Package List “submenu allows selecting the packages that will be used in building the root filesystem. Busybox is in that list and “Configure busybox at build time” allows the user to do the Busybox menuconfig when LTIB gets to the point that it is going to build Busybox.

The “Target System Configuration Options” submenu allows various settings including the kernel’s default command line.

The “Target Image Generation Options” submenu allows selecting various root filesystem deployment options.

To modify the project configuration simply run:

```
./ltib –-configure
```

This prompts for the platform/board configuration. In the board configuration screens, change settings and select packages as appropriate. When you exit the configuration screen, your target image is adjusted accordingly and LTIB begins building the kernel, modules, and root filesystem.

Ltib has a 'profile' which specifies all the rest of the packages that are expected in default rootfs. To set it, run the command:

```
./ltib --profile config/platform/imx/release.profile --batch
```

To build using the default configuration:

```
./ltib --preconfig config/platform/imx/imx31_3stack.cf
```

To switch platforms:

```
./ltib --selectype
```

When LTIB builds, the results of building are packaged as `rpm` files. These `rpm` files are located at `ltib/rpm/RPMS/arm`. When doing rebuilds, LTIB re-uses the `rpm` files it has built unless they have been deleted or the ‘-f’ has been specified.

Once you build your project you will get the following directory/image files (Depending on the Target Image Generation Options selected in LTIB):

- `rootfs` – directory, the root file system that is to be deployed on your board, you can use this to boot from NFS..
- `rootfs.ext2.gz` – EXT2 filesystem - You can use this to create a CRAMFS filesystem that can be downloaded to SDRAM or flashed to your board. Note that this filesystem is more optimized in size than the `rootfs` directory as some unnecessary files have been deleted.
- `rootfs.cramfs` - LTIB-generated CRAMFS filesystem. You can download to SDRAM or flash this file to your board.
- `rootfs.jffs2` - LTIB-generated JFFS2 filesystem. You can flash this file to your board.
- Refer to RedBoot documentation and the i.MX31 PDK Linux User's Guide for detailed steps on flashing procedures.
- `rootfs/boot/zImage` – kernel image that can be loaded with RedBoot
- The kernel modules have been built and copied into the `rootfs`

If you want to fully re-configure and re-compile all the packages, you can do the following. (Note that this is generally not necessary.)

1. Clean up all the configure files and objects thoroughly:
   ```
   ./ltib –m distclean
   ```
2. You are prompted to confirm your choice. Type yes to perform a `distclean`.
3. Run `ltib`
   ```
   ./ltib
   ```

Make sure to set up the network parameters in LTIB if booting from NFS:

```
./ltib -c
```

Set the network parameters in the following path:

```
Target System Configuration
    Options--->
    Network setup
        IP address
        netmask
        broadcast address
        gateway address
        nameserver IP address
```

2.3.2.2 Creating a CRAMFS Root Filesystem

To create a CRAMFS root filesystem, use the `rootfs.ext2.gz` as it has been optimized in size compared to the `rootfs` or instruct LTIB to create this file with the option Target Image Generation Options -> Target Image -> cramfs:

```
mkdir temp
gunzip rootfs.ext2.gz
```

`su` to be root or do `sudo` for this:

```
mount -o loop -t ext2 rootfs.ext2 temp
```

Now `rootfs` is mounted on temp. You can point your nfs to the temp folder. To make a `cramfs`:

```
rm -rf temp/lost+found
mkcramfs temp rootfs.cramfs
```

Redboot exec command is as follows:

```
exec -c "noinitrd console=ttymxc0,115200 root=/dev/mtdblock2 rw rootfstype=cramfs
ip=dhcp"
```

NOTE

In some Linux distributions the mkcramfs tool is named mkfs.cramfs.

2.3.2.3 Various Useful LTIB Commands

Note that ltib will give a list of all its commands by invoking:

```
./ltib -h
```

2.3.2.3.1 Changing Platforms that LTIB is Building

The platform selection is saved in the `.config` file. The following brings up the LTIB platform selection menu.

```
rm -f .config
./ltib
```

2.3.2.3.2 Compiling the Kernel and Modules using Local Source

1. Do a `menuconfig` to select the platform and set the path of the local Linux source directory:
   ```
   ./ltib -m config
   ```
   A menu will come up.
2. Hit <enter> and select the platform using the up and down arrow keys.
3. Exit saving changes. Then another menu will appear.
4. Select to build the kernel from a local Linux source directory.
5. Change the path of where your kernel source directory is located to the correct absolute path like `/home/buffy/LINUX2.6/linux`. The `kbuild` output directory will default to `../kbuild/$platform` relative to your Linux folder. Or you can change it if you like.
6. To do the kernel `menuconfig`, select the “Configure the kernel” option. This option will get reset once the kernel successfully builds.
7. Exit, saving changes.

To do a clean kernel build, delete your `kbuild` directory.

LTIB’s -p option specifies to only build one package, so you can use it to select the local kernel build:

```
./ltib -p kernel -f
```

The build `zImage` ends up in `rootfs/boot`. The modules end up in `rootfs/lib/modules`.

2.3.2.3.3 Extracting the Kernel Source

The kernel source is released as a set of patches. To use `ltib` to patch them together:

```
./ltib -m config
```

A menu comes up. Press <**enter**> and select the platform using the up and down arrow keys. Exit saving changes.

Then another menu appears. Under “Choose your kernel” select “kernel (Linux 2.6.26-imx)”. Exit saving changes. Then type:

```
./ltib -p kernel -m prep
```

The kernel gets patched together and appears under `<ltib_dir>/rpm/BUILD/linux-2.6.26.`

2.3.2.3.4 Cleaning Up

To delete everything in the `rootfs` directory and clean up:

```
./ltib -m clean
```

To clean even more severely, delete the `rpm` directories that LTIB has built.

```
./ltib -m distclean
```

To clean up after a failed build:

```
rm -rf tmp rpm/BUILD/* rpm/SOURCES/*
```

2.3.3 Kernel Modules on the Target Platform

LTIB will build the kernel modules and install them in the root filesystem in the `/lib/modules/2.6.26-*` folder. During the LTIB build process the modules.dep file is created so that depmod does not need to be run on the target. One of the advantages of using LTIB to build the kernel is that it updates the modules in the root filesystem and the modules.dep automatically.

Chapter 3
Architecture

This chapter describes the overall architecture of the Linux port to the i.MX family of processors. The BSP supports all platforms in a single development environment, but not every driver is supported by all processors. Drivers common to all platforms are referred to as i.MX drivers and drivers unique to a specific platform are referred to by the platform name.

3.1 Linux BSP Block Diagram

Figure 3-1 shows the architecture of the BSP for the i.MX family of processors. It consists of user-space executables, standard kernel components that come from the Linux community, hardware-specific drivers, and functions provided by the Freescale for the i.MX family of processors.

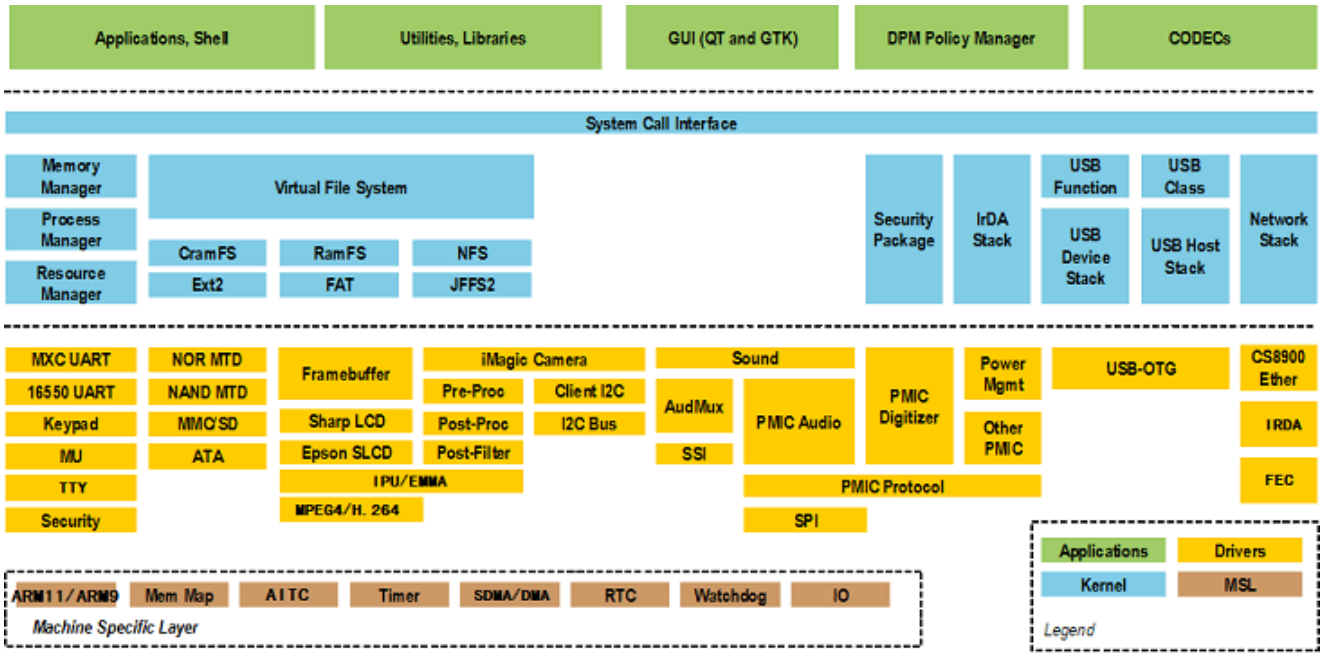


Figure 3-1. Linux BSP Block Diagram

3.2 Kernel

The i.MX Linux port is based on the standard Linux kernel. The kernel supports many of the features expected in most modern embedded OSes:

- Process and thread management

- Memory management (memory mapping, allocation/deallocation, MMU, and L1/L2 cache control)
- Resource management (interrupts)
- Power management
- File systems (VFS, cramfs, ext2, ramfs, NFS, devfs, jffs2, fat)
- Driver model
- Standardized APIs
- Networking stacks

ARM Linux Kernel customization to support each platform includes a custom kernel configuration and machine specific layer (MSL) implementation.

3.2.1 Configuration

For this BSP release, kernel configuration is done through LTIB. See the LTIB documentation for details. The following are some of the configuration settings apart from the standard features:

The serial port is labeled as UART-DCE on the i.MX31 PDK.

- Embedded mode
- Module loading/unloading
- ARM11
- File formats supported: ELF binaries, a.out and ECOFF
- Block devices: Loopback, Ramdisk
- i.MX Internal UART
- File systems: ext2, dev, proc, sysfs, cramfs, ramfs, jffs2, fat, pramfs
- Framebuffer
- Kernel debugging
- Automatic kernel module loading
- Power management
- Memory Technology Device (MTD) support

3.2.2 Machine Specific Layer (MSL)

The MSL provides a machine-dependent implementation as required by the Linux kernel, such as memory map, interrupt, and timer. Each ARM platform has its own MSL directory under the `arch/arm` directory as listed in Table 3-1.

Table 3-1. Machine Directories

Platform	Directory
i.MX31 3-Stack	<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3

For more information, see Chapter 4, “Machine Specific Layer (MSL).”

3.2.2.1 Memory Map

Before the kernel starts running in the virtual space, the physical-to-virtual address mapping for the IO peripherals needs to be provided for the MMU to do the translation for memory/register accesses. The mapping is done through a table structure in the MSL specific to a particular platform, with each entry specifying a peripheral's starting address of virtual addresses, starting address of physical addresses, and the size of the memory region and the type of the region.

3.2.2.2 Interrupts

The standard Linux kernel contains common ARM® code for handling interrupts. The MSL contains platform-specific implementations of functions for interfacing the Linux kernel to the ARM11™ vectored interrupt controller (AVIC).

Together, they support the following capabilities:

- AVIC initialization
- AITC initialization
- Interrupt enable/disable control
- ISR binding
- ISR dispatch
- Interrupt chaining
- Standard Linux API for accessing interrupt functions
- Static mapping of one interrupt source as FIQ

Vectored interrupts and fast interrupt are not supported.

3.2.2.3 General Purpose Timer (GPT)

The GPT is set up to generate an interrupt every 10 msec to provide OS ticks. This timer is also used by the kernel for additional timer events. Linux defines the MSL timer API required for the OS-tick timer and does not expose it beyond the kernel tick implementation. Linux facilitates timer use through various functions for timing delays, measurement, events, and alarms. The GPT is also used as the source to support the High Resolution Timer feature.

The timer tick interrupt is disabled when in low-power modes other than idle.

3.2.2.4 Smart Direct Memory Access (SDMA) API

The SDMA controller is responsible for transferring data between the MCU memory space, and peripherals. It is based on a microRISC engine that runs channel-specific scripts. The SDMA API allows other drivers to initialize the scripts, pass parameters, and control their execution. Complete support for SDMA is provided in three layers (see Figure 3-2). The first layer is the I.API, the second layer is the Linux DMA API, and the third layer is the TTY driver. The first two layers are part of the MSL and both are custom.

I.API is the lowest layer and it interfaces the Linux DMA API with the SDMA controller. The Linux DMA API interfaces other drivers (for example: MMC/SD or Sound) with the SDMA controller through the I.API. 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

Figure 3-2 is a block diagram of SDMA.

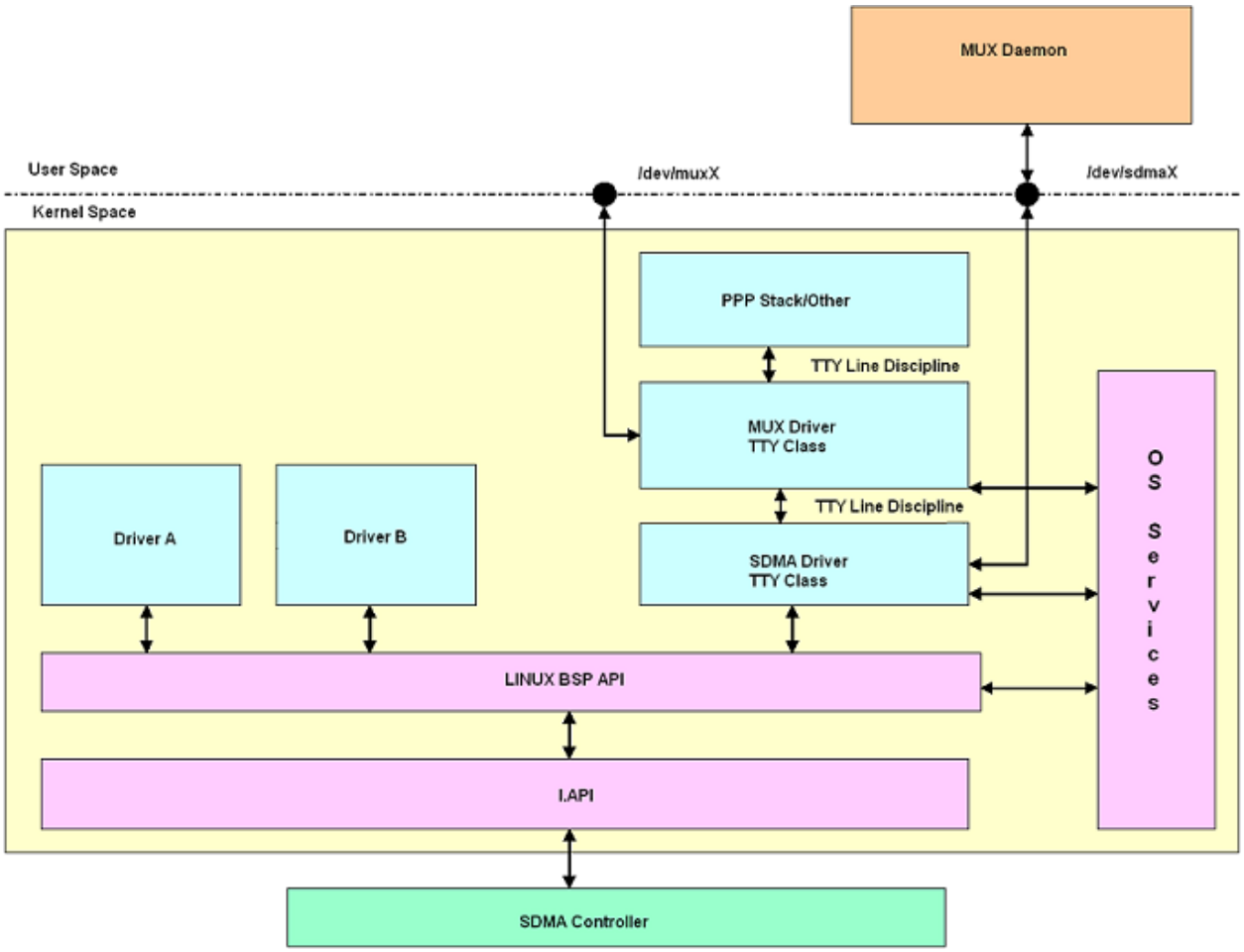


Figure 3-2. SDMA Block Diagram

The SDMA API is present on i.MX31.

3.2.2.5 Input/Output (I/O)

The Input/Output (IO) component in the MSL provides 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. It provides the following features as part of a custom kernel-space API:

- Initialization for the default I/O configuration after boot
- Functions for configuring the various I/O for active use
- Functions for configuring the various I/O for low power mode
- Functions for controlling and sampling GPIO and board I/O
- Functions for enabling, disabling, and binding callback functions to GPIO and EDIO interrupts
- Functions to support different priority levels during ISR registration for different modules; if more than one interrupt occurs at the same time, the higher priority ISR callback gets called first
- Atomic helper functions for GPIO, EDIO, and IOMUX configuration

These functions are organized by functional usage, and not by pin or port. This allows I/O configuration changes to be centralized in the GPIO module without requiring changes in the various drivers. These functions are used by other device drivers in the kernel space. User level programs do not have access to the functions in the GPIO module.

The exact API and implementations are different on each platform to account for the differences in hardware, drivers, and boards. This module is an evolving module. As more drivers are added, more functions are required from this module.

3.2.2.6 Shared Peripheral Bus Arbiter (SPBA)

The SPBA provides an arbitration mechanism among multiple masters to have 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. These functions are also exported so that they can be used by other loadable modules.

3.3 Drivers

There are many drivers provided by Freescale that are specific to the peripherals on the i.MX family of processors or to the development platforms. Many of these drivers are common across all of the platforms. Most can be compiled into the kernel or compiled as object modules which can be dynamically loaded from a file system through insmod or modprobe. Modules can be loaded automatically as required using the kernel auto-load feature. The BSP contains a modules.dep file and modprobe.conf file that contain the dependency information for the modules.

The i.MX multimedia applications processors have several classes of drivers, explained in the following sections.

3.3.1 Character Device Drivers

The i.MX family of processors support a Universal Asynchronous Receiver/Transmitter (UART) driver. The character device drivers, summarized in this section, are the UART Driver.

3.3.1.1 Universal Asynchronous Receiver/Transmitter (UART) Driver

UART driver interfaces the Linux serial driver API to all of the UART ports. It supports the following features:

- Interrupt-driven and SDMA-driven transmit/receive of characters
- Standard Linux baud rates up to 1.5Mbps
- Transmitting and receiving characters with 7-bit and 8-bit character lengths
- Transmitting 1 or 2 stop bits
- Odd and even parity
- XON/XOFF software flow control
- CTS/RTS hardware flow control (both interrupt-driven software controlled hardware flow control and hardware-driven hardware flow control)
- `TIOCMGET` ioctl to read the modem control lines. Supports the constants `TIOCM_CTS` and `TIOCM_CAR`, `TIOCM_RI` (only in `DTE` mode) only
- `TIOCMSET` ioctl to set the modem control lines. Supports the constants `TIOCM_RTS` and `TIOCM_DTR` only
- Send and receive of break characters through the standard Linux serial API
- Recognize 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 ioctls.
- Slow IrDA (IrDA at or below 115200 baud)
- Power management features - suspends and resumes the UART ports
- The standard TTY layer `ioctl` calls
- Includes console support that is needed to bring up the command prompt through one 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.

All the UART ports can be accessed through the device files `/dev/ttymxc0` through `/dev/ttymxcX` (where X is the maximum UART number supported by the IC). `/dev/ttymxc0` refers to `UART 1`. Autobaud detection is not supported.

3.3.1.2 Real-Time Clock (RTC)

The Real-Time Clock (RTC) is the clock that keeps the date and time while the system is running and even when the system is inactive. The RTC implementation supports `ioctl` calls to read time, set time, set up periodic interrupts, and set up alarms. Linux defines the RTC API.

3.3.1.3 Watchdog Timer (WDOG)

The Watchdog timer protects against system failures by providing a method of escaping from unexpected events or programming errors.

The WDOG software implementation provides routines to service the WDOG timer, so that the timeout does not occur. The WDOG is serviced (at the same time for the platforms with two WDOGs) if is already enabled before the Linux kernel boots (enabled by boot loader or ROM) with the service interval being configurable. In addition, compile-time options specify whether the Linux kernel should enable the watchdog, and if so, which parameters should be used. If the second WDOG presents (used to generate an interrupt after the timeout occurs), the highest interrupt priority (number 16) is assigned to the WDOG interrupt.

The Linux OS has a standard WDOG interface that allows a WDOG driver for a specific platform to be supported. This is supported under all i.MX platforms.

For the platforms that have two WDOG hardware modules, another implementation is done in the Machine-specific Layer as part of the `time.c` file per the requirement from the customers.

3.3.1.4 SDMA API

The SDMA controller is responsible for transferring data between the MCU memory space and the peripherals. It is based on a microRISC engine that runs channel specific scripts. The SDMA API allows other drivers to initialize the scripts, pass parameters, and control their execution. Complete support for SDMA is provided in three layers (see Figure 3-2). The first layer is the I.API, the second layer is the Linux DMA API, and the third layer is the TTY driver. The first two layers are part of the MSL and both are custom. I.API is the lowest layer and it is the interface between the Linux DMA API and the SDMA controller. The Linux DMA API interfaces with other drivers (for example: MMC/SD, Sound) with the SDMA controller through the I.API.

Functions of the SDMA API include:

- 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

3.3.2 Image Processing Unit (IPU) Architecture

The Image Processing Unit (IPU) is designed to support video and graphics processing functions in the i.MX architecture. It also interfaces with video/still image sensors and displays.

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 Display Controller (SDC)
- Asynchronous Display Controller (ADC)
- Display Interface (DI)
- Image DMA Controller (IDMAC)
- CMOS Sensor Interface (CSI)
- Image Converter (IC)
- Post-Filter (PF)

Figure 3-3 shows the IPU architecture.

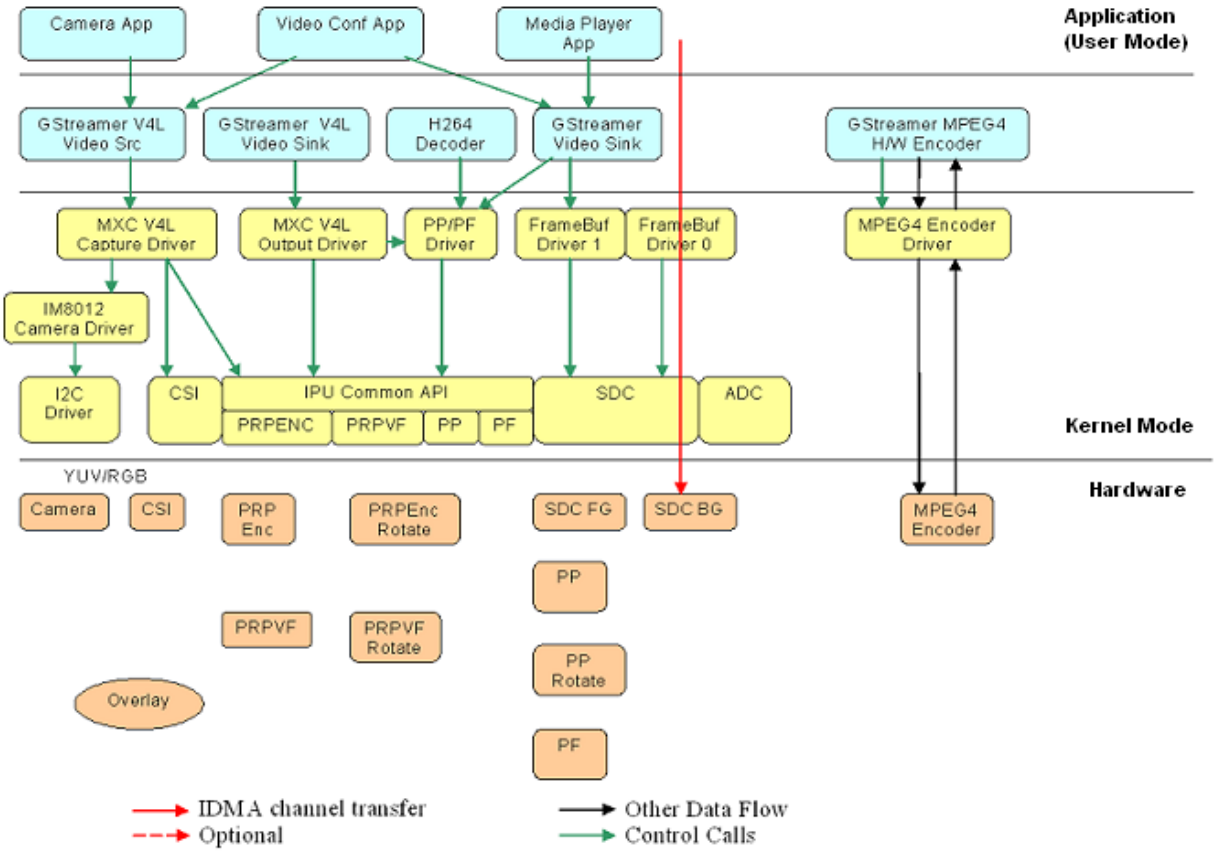


Figure 3-3. IPU Architecture

3.3.2.1 IPU Driver

The IPU driver abstracts the IPU hardware. 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 to between one and 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, to set up buffers, to enable and disable channels, to link channels for auto frame synchronization, and to set up interrupts.

Logical channels include:

- CSI Direct to Memory
- CSI to Viewfinder Pre-processing to Memory or ADC
- Memory to Viewfinder Pre-processing to Memory or ADC
- 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 or ADC
- Memory to Post-processing Rotation to Memory
- Memory to Post Filter (Y buffer) to Memory
- Memory to Post Filter (U buffer) to Memory
- Memory to Post Filter (V buffer) to Memory
- Memory to SDC Background
- Memory to SDC Foreground
- Memory to SDC Mask
- Memory to ADC System Channel 1
- Memory to ADC System Channel 2

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 listed below:

- SDC Functions
  - Panel interface initialization
  - Set foreground and background plane positions
  - Set global alpha and color key
  - Set backlight level
- CSI Functions
  - Sensor interface initialization
  - Set sensor clock
  - Set capture size
- ADC Functions
  - Panel interface initialization
  - Send commands to panel

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

3.3.2.2 Synchronous Display Controller (SDC) Framebuffer Driver

The Synchronous Display Controller (SDC) Framebuffer screen driver implements a Linux standard framebuffer driver API for synchronous or memory-less LCD panels. The SDC Framebuffer screen driver is the top level kernel video driver that interacts with kernel and user-level applications. The SDC Framebuffer driver is enabled by selecting the Framebuffer option under the graphics parameters in the kernel configuration. To supplement the framebuffer 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 framebuffer 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 framebuffer driver supports the Sharp QVGA and Epson VGA panels.

The SDC Framebuffer screen driver interacts with the IPU Driver using custom APIs:

- Initialization of panel interface settings
- Initialization of IPU channel settings for LCD refresh
- Control of IPU SDC PWM for backlight control
- Changing the frame buffer address for double buffering support

The following features are supported:

- Support for Sharp QVGA and EPSON VGA panels
- Configurable screen resolution
- Configurable RGB 16, 24 or 32 bits per pixel framebuffer
- Configurable panel interface signal timings and polarities
- Palette/color conversion management
- Power management
- LCD Power off/on
- Backlight control

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

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

A second framebuffer driver supports a second video/graphics plane

3.3.2.3 Asynchronous Display Controller (ADC) Framebuffer Driver

The Asynchronous Display Controller (ADC) Framebuffer screen driver implements a Linux standard framebuffer driver API for asynchronous or smart LCD panels. The ADC Framebuffer screen driver is the top level kernel video driver that interacts with the kernel and user level applications. The ADC Framebuffer driver is enabled by selecting the Framebuffer option under the graphics parameters in the kernel configuration. To supplement the framebuffer 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.

The framebuffer interacts with the IPU Driver using custom APIs:

- Initialization of panel interface settings for serial or parallel mode
- Initialization of IPU channel settings for ADC commands and data
- Control of IPU bus snooping for automatic update of panel memory

The following features are supported:

- Support for Epson L2F50032T00 dual mode 176x220 panel
- Configurable RGB 16, 24 or 32 bits per pixel framebuffer
- Configurable panel interface signal timings and polarities
- Palette/color conversion management
- Power management
- LCD Power off/on
- Backlight control

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

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

3.3.2.4 V4L2 Camera Driver

The V4L2 camera driver is a plug-in to the V4L2 framework that enables support for the IPU camera and preprocessing functions. The V4L2 camera driver implements support for all camera related functions. The V4L2 camera driver uses the iMagic or OmniVision sensor driver (or other sensor) and the IPU driver.

The features supported by the V4L2 driver are:

- Direct preview to SDC foreground overlay plane (no ARM® processor intervention and synchronized to LCD refresh)
- Direct preview to graphics frame buffer (no ARM processor intervention, but NOT synced to LCD refresh)
- Support for color keying 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
- Support for RGB 16-bit, 24-bit, and 32-bit preview formats
- Support for 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 (TBD) formats for capture
- Control of sensor properties including exposure, white-balance, brightness, contrast, and so on
- Support for plug-in of different sensor drivers

An example of the data flow for camera with capture to MPEG4 is shown in Figure 3-4. The example includes rotation of viewfinder and encode channels using automatic IPU frame synchronization.

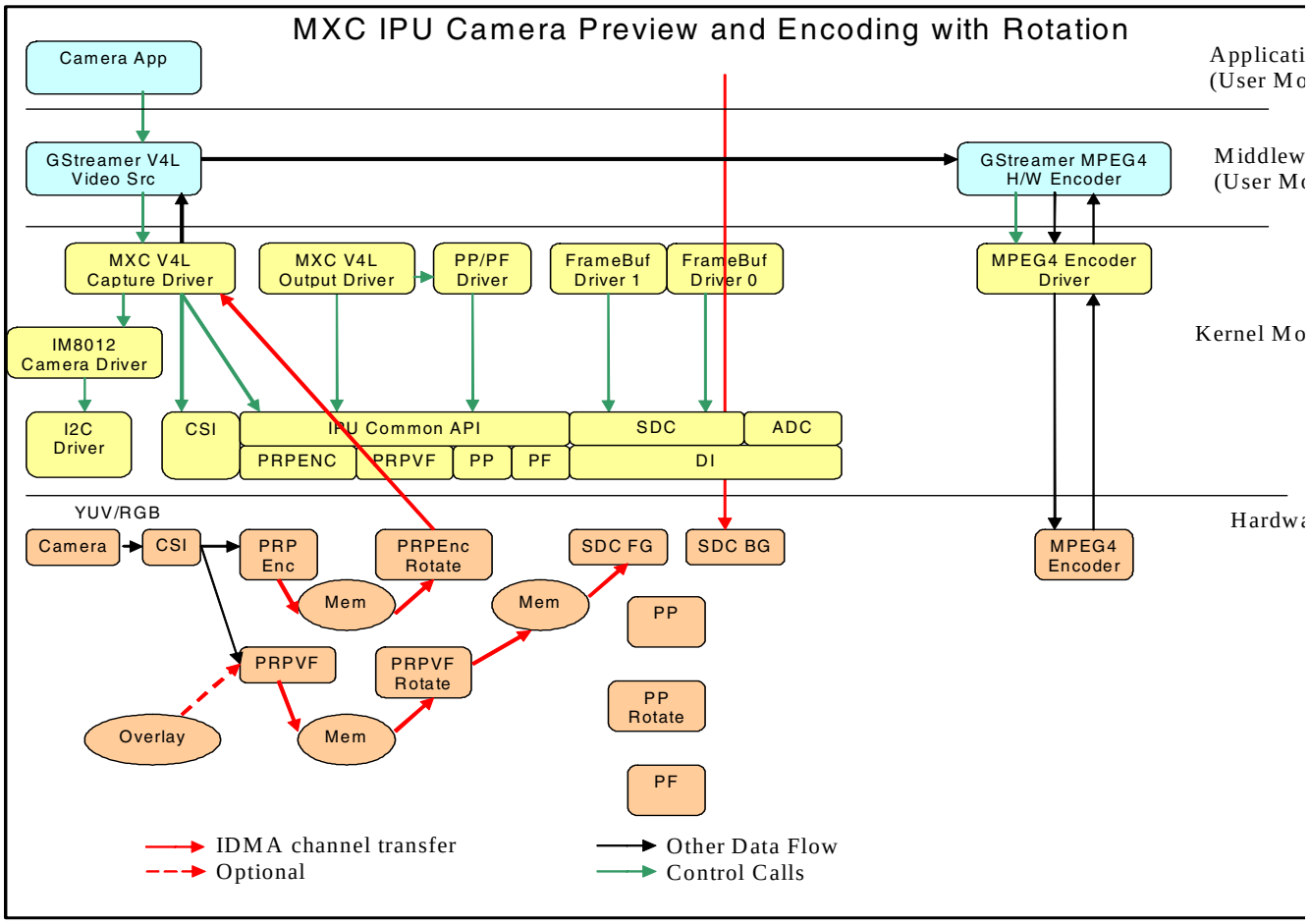


Figure 3-4. Camera Preview and Encoding

3.3.2.5 V4L2 Output Driver

The V4L2 Output driver uses the IPU post-processing functions for video output. The driver implements the standard V4L2 API for output devices. The V4L2 features supported by the driver are:

- Direct output to SDC foreground overlay plane (no ARM processor intervention and synchronized to LCD refresh)

- Support for color keying alpha blending of frame buffer and overlay planes
- Support for linking post-processing resize and CSC, rotation, and display IPU channels for 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
- Support for RGB 16, 24, and 32 bit, YUV 4:2:0 and 4:2:2 planar, and YUV 4:2:2 interleaved input formats
- Support for TV output

The features are supported using custom APIs:

- Output to user buffer instead of overlay display
- Programmable rotation

3.3.2.6 MPEG4/H.264 Post-Filtering Driver

The Post-filtering driver provides a custom user API for IPU post-filtering functions. The features supported by the driver are:

- Support for MPEG4 ordering and/or deblock
- Support for H264 deblock
- Support for intra-frame pause and resume (H.264 only)
- Synchronous and asynchronous operation (async depends on time required to process a frame)
- Support for driver allocated or user allocated buffers

3.3.3 Graphics Processing Unit (GPU) Driver

The GPU on i.MX31 is a licensed core from Imagination Technologies. In the i.MX31 documentation, the name GPU is used synonymously with the names MBX-Lite and MBX. The GPU is an accelerator for Graphics.

The OpenGL ES Driver Development Kit (DDK) has been ported for the i.MX31 architecture from the base drop provided by Imagination Technologies. The driver provides the following features:

- Utilize MBX-Lite 3D capabilities for hardware acceleration of graphics.
- Expose OpenGL-ES API to applications.
- Use PVR2D a small loadable module for 2D on MBX-Lite. This module works with Power VR Services which provide the correct interaction with the OpenGL ES driver.

3.3.4 Sound Driver

The components of the audio subsystem are applications, the Advanced Linux Sound Architecture (ALSA), the audio driver, and the hardware. Applications interface with ALSA, and ALSA interfaces with the audio driver, which in turn controls the hardware of the audio subsystem. For more information about ALSA, see www.alsa-project.org.

The sound driver runs on the ARM11 processor. Digital audio data is carried over the digital audio link interface to the codec hardware. This is managed by the audio driver. There may be one or more audio streams, depending on the codec, such as voice or stereo DAC. The audio driver also configures sample rates, audio MUXing, formats, and audio clocks. The audio driver also manages the setup and control of the CODEC, DMA, and audio accessories, such as headphones and microphone detection. Stream mixing may also be supported, depending on the codec.

3.3.5 Input Device Drivers – Keypad Driver

The keypad driver interfaces Linux to the keypad controller (KPP) in the i.MX architecture. The software operation of the keypad driver follows the Linux keyboard architecture.

It supports the following features:

- Supports up to 8 x 8 external key pad matrix of single poll switches. The keypad matrix can be configured
- Any pins not used for keypad are available as general purpose I/O through `ioctl` call
- The keypad driver supports two modes of interface to the upper layer. The modes are raw mode and map mode (xlate mode)
- The keypad driver is implemented as a single driver.
- When configured in raw mode the scancode of keys pressed and released are sent to the upper layer
- When configured in map mode the mapcode (key mapping) of keys pressed and released are sent to the upper layer. Scancodes are converted into mapcodes from the keymap lookup table
- Dynamic configuration of keymap translation table (static default) through `ioctl` call
- The keypad mode can be set using `KDSKBMODE` `ioctl` call
- Supports multiple key presses (with required keypad design)
- A long key press can be configured to generate multiple key press events
- Supports key press detection in standby mode
- The keypad driver follows the standard Linux Keyboard API
- Key chording handled by users of this driver (GUI)

3.3.6 Memory Technology Device (MTD) Drivers

Memory Technology Devices (MTDs) in Linux cover all memory devices, such as RAM, ROM, and different kinds of Flashes. As each memory device has its own idiosyncrasies in terms of read and write, the MTD subsystem provides a unified and uniform access to the various memory devices.

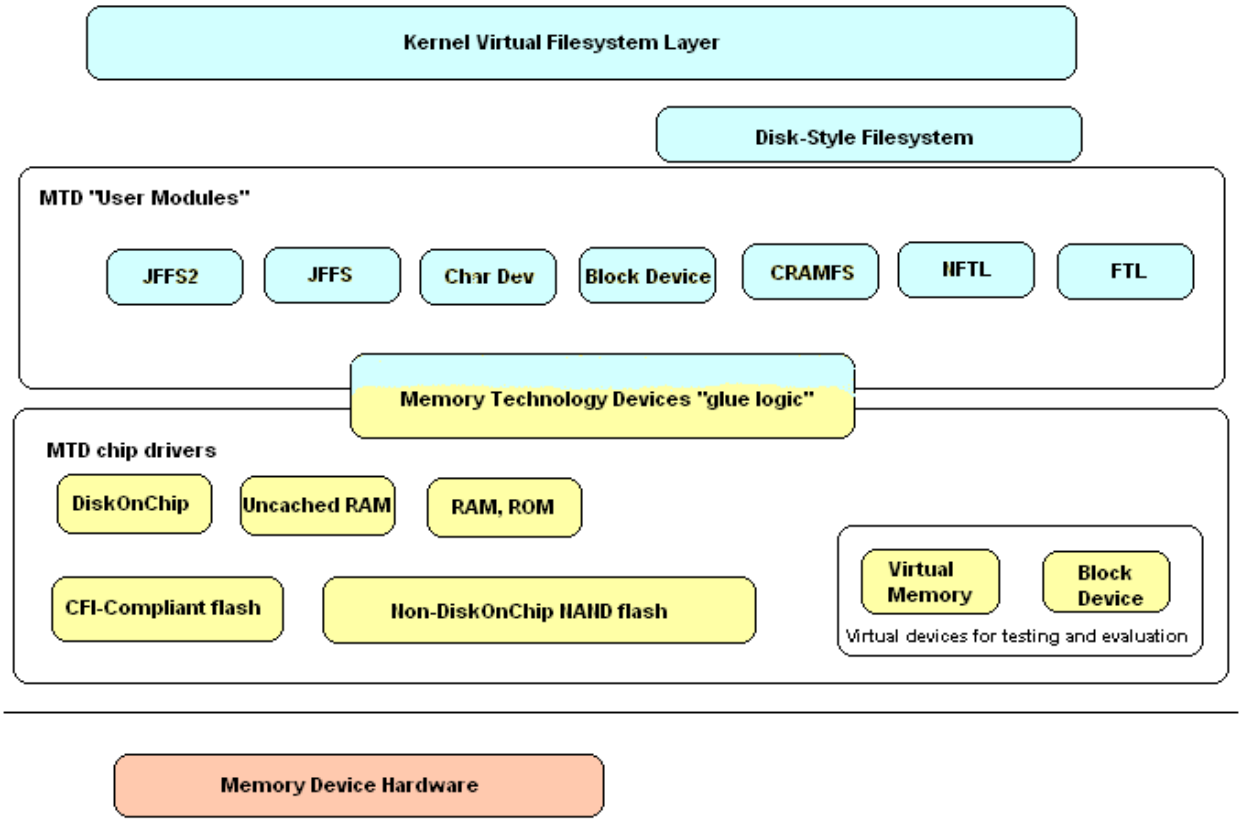


Figure 3-5. MTD Architecture

Figure 3-5 is excerpted from the *Building Embedded Linux Systems* book, which describes the MTD subsystem. The “user modules” should not be confused with kernel modules or any sort of user-land software abstraction. The term “MTD user module” refers to software modules within the kernel that enable access to the low-level MTD chip drivers by providing recognizable interfaces and abstractions to the higher levels of the kernel or, in some cases, to user space.

MTD chip drivers register with the MTD subsystem by providing a set of predefined callbacks and properties in the `mtd_info` argument to the `add_mtd_device()` function. The callbacks an MTD driver has to provide are called by the MTD subsystem to carry out operations, such as erase, read, write, and sync.

3.3.6.1 NAND MTD Driver

The NAND MTD driver interfaces with the integrated NAND controller on the i.MX processors. It can support various file systems, such as CRAMFS and JFFS2. 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. This driver is part of the kernel image.

3.3.7 Networking Drivers

The networking drivers are described in the next sections.

3.3.7.1 SMSC LAN9217 Ethernet Driver

The SMSC LAN9217 Ethernet Driver interfaces SMSC LAN9217-specific functions with the standard Linux kernel network module. The LAN9217 is a full-featured, single-chip 10/100 Ethernet controller designed for embedded applications where performance, flexibility, ease of integration, and system cost control are required. The LAN9217 has been specifically designed to provide the highest performance possible for any 16-bit application. The LAN9217 is fully IEEE 802.3 10BASE-T and 802.3u 100BASE-TX compliant, and supports HP Auto-MDIX.

The SMSC LAN9217 Ethernet Driver has the following features:

- The efficient PacketPage Architecture can operate in I/O and memory space, and as a DMA slave
- Supports full duplex operation
- Supports on-chip RAM buffers for transmission and reception of frames
- Supports programmable transmit features like automatic retransmission on collision and automatic CRC generation
- EEPROM support for configuration
- Supports MAC address setting
- Supports obtaining statistics from the device, such as transmit collisions

This network adapter can be accessed through the `ifconfig` command with interface name (`eth0`). The probe function of this driver is declared in `drivers/net/Space.c` to probe for the device and to initialize it during boot.

3.3.8 Disk Drivers

The disk drivers include the ATA driver.

3.3.8.1 ATA Disk Driver

The ATA module is an AT attachment host interface. Its main use is to interface with hard disk devices.

The ATA driver is compliant to the ATA-6 standard, and supports the following protocols:

- PIO mode 0, 1, 2, 3, and 4
- multiword DMA mode 0, 1, and 2
- Ultra DMA mode 0, 1, 2, 3, and 4 with bus clocks of 50MHz or higher
- Ultra DMA mode 5 with bus clock of 80MHz or higher

3.3.9 USB Drivers

It relies on the existing USB host stack in Linux and is combined with the Belcarra device stack with enhancements to support OTG. The solution is comprised of several drivers as shown in Figure 3-6.

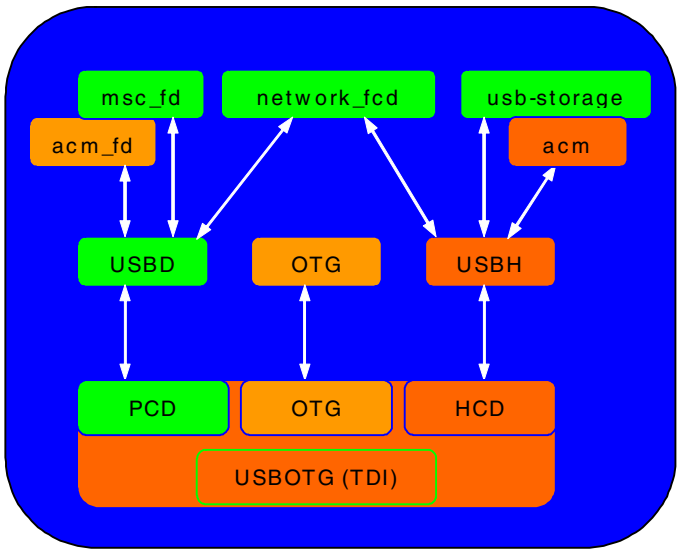


Figure 3-6. USB-OTG SW Block Diagram

The solution uses the following USB host modules from the standard Linux kernel:

- USB Host Stack (usbh)
- USB Mass Storage Class Driver (usb-storage)
- USB ACM Class Driver (acm)

The USB host stack is modified by Belcarra to support USB-OTG.

The following drivers are provided from Belcarra's standard product:

- USB Device Stack (usbd)
- Mass Storage Function Driver (msc_fd)
- ACM Function Driver (acm_fd)
- Network Function Driver combined with Network Class Driver (network_fcd)

With USB-OTG, the Network Function and Network Class driver have been combined into a single driver in order to provide seamless service during a role reversal.

To support USB-OTG, Belcarra has defined a new driver known as the OTG Manager. The OTG Manager implements the administrative control required by user space.

The portions of the USB stack specific to the TDI OTG module are integrated into a single driver. This driver encompasses the functionality typically available in the PCD and HCD drivers as well as OTG coordination and transceiver control. This driver is implemented in a layered approach in order to allow a significant portion of the driver to be portable to other devices.

3.3.10 Security Drivers

The i.MX processors support many hardware and software security modules, discussed in the following sections.

3.3.10.1 Security Controller Module (SCC) Driver

The security layer is comprised of two modules, the Secure RAM Module and the Secure Monitor Module. The Secure RAM module provides a secure way of storing sensitive data in on-chip and off-chip RAM memory. On-chip data can be cleared if necessary to prevent un-authorized access. Off-chip data is stored in encrypted form using an encryption key that is unique to each device and is accessible only through Secure RAM module. The Security Controller (SCC) is a part of the Freescale platform independent security architecture (PISA). The SCC module will only be accessible by the ARM11. It supports the following features:

- An autonomous hardware security state controller with debug inputs that are tied to all platform test access detection signals to trigger a security shutdown
- Controls to ensure supervisory mode only configuration access
- Controls to ensure that high assurance internal boot is the only mechanism to reach the Secure state after Reset
- An autonomous hardware security state controller with “debug” inputs that are tied to all platform test access detection signals to trigger shutdown
- A self-clearing (zeroizing) 2KB RAM block, which clears itself upon command and can therefore be used to store security sensitive Red data (that is, security sensitive plain text), such as cryptographic keys
- A Security Timer which is an independent security watchdog timer whose time-out triggers a security violation
- An Algorithm Sequence Checker (ASC) which can be used by software to force software synchronization to the ASC's internal linear feedback shift register (LFSR) as a software assurance check
- A “Bit Bank” counter that can be used with the ASC to ensure that a scrambler function uses the same number of algorithm bits as traffic bits to ensure that no traffic data is “accidentally” left in the clear
- A Plaintext/Ciphertext comparator that may be used to ensure that a cryptographic algorithm scrambler has not been replaced with a simple pattern EXOR function
- Some portion of the SCC is used during initial boot-up from the iROM
- Some portion is used as a security measure during runtime, for example, tampering of the hardware. This is used to clear the secure data either in the internal RAM or externally encrypted data RAM.
- Power management

3.3.10.2 Random Number Generator Accelerator (RNGA) Driver

The Random Number Generator Accelerator (RNGA) module is a digital integrated circuit capable of generating 32-bit random numbers. It is designed to comply with FIPS-140 standards for randomness and non-determinism. The oscillators with their unknown frequencies provide the required entropy needed to create random data. An Entropy register is provided which serves the purpose for seeding the Random number generator.

The RNGA includes the following features:

- Generates Random numbers
- Interface for user to enter initial seed value for generating random numbers
- 16 x 32 FIFO
- Secure mode
- Power saving mode

3.3.11 General Drivers

General drivers discussed in the following sections, include the following:

- Multimedia Card (MMC)/SD driver
- PCMCIA driver
- I^2C Client and Bus drivers
- Digital Audio Multiplexer (AUDMUX) driver
- Synchronous Serial Interface (SSI) driver
- Graphics Processing Unit (GPU) driver
- Dynamic Power Management (DPM) driver

3.3.11.1 Multimedia Card (MMC)/SD Driver

The Multimedia Card (MMC)/SD driver implements a standard Linux slot driver as well as a block driver interface to the MMC/SDHC controller. The interface to the upper layer follows the standard Linux driver API.

- SDHC module supports MMC and SD cards
- MMC version 3.0 spec is supported. SD Memory Card spec 1.0 and SD I/O card spec 1.0 are supported.
- Hardware contains 32x16 bit data buffer built in
- Plug and play support
- 100 Mbps Maximum hardware data rate in 4-bit mode
- 1/4 bit operation
- For SD card access, only SD bus mode is supported. SPI mode is not supported.
- Supports card insertion and removal events
- Supports the standard MMC/SD/SDIO commands

- Supports Power management
- Supports set/reset of password or card lock/unlock commands
- Power management

3.3.11.2 Inter-IC (I^2C) Bus Driver

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. 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:

- Compatibility with the I^2C bus standard
- Supports bit rates up to 400kbps
- Start and stop signal generation/detection
- Acknowledge bit generation/detection
- Interrupt-driven, byte-by-byte data transfer
- Supports standard I^2C master mode
- Supports power management features by suspending and resuming I^2C

The I^2C slave mode is not supported by this driver.

3.3.11.3 Digital Audio Multiplexer (AUDMUX) Driver

The low level Digital Audio Multiplexer (AUDMUX) driver provides a custom, kernel-space API to the AUDMUX module. It supports all of the features of the hardware module.

3.3.11.4 Synchronous Serial Interface (SSI) Driver

The low-level synchronous serial interface (SSI) driver provides a custom, kernel-space API to the SSI modules. It supports all of the features of the hardware modules including enabling/disabling of DMA request events. Drivers configure DMA channels through the SDMA API.

3.3.11.5 Configurable Serial Peripheral Interface (CSPI) Driver

The low-level Configurable Serial Peripheral Interface (CSPI) driver interfaces a custom, kernel-space API to the 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

DMA is not supported.

3.3.11.6 Dynamic Power Management (DPM) Driver

Dynamic Power Management (DPM) refers to power management schemes implemented while programs are still running. DPM focuses on system wide energy consumption while it is running. In any CPU-intensive application, lowering bus frequencies from their maximum performance points can result in system wide energy savings.

DPM implementation includes the following data structures:

- Operating Points
- Operating States
- Policies
- Policy manager

3.3.11.7 Policy Architecture

A DPM policy is a named data structure installed in the DPM implementation within the operating system, and managed by the policy manager, which may be outside of the operating system. Once a DPM system is initialized and activated, the system is always executing a particular DPM policy.

3.3.11.8 Operating Points

At any given point in time, a system is said to be executing at a particular *operating point.* The operating point is described using hardware parameters, such as core voltage, CPU and bus frequencies, and the states of peripheral devices. A DPM system could properly be defined as the set of rules and procedures that move the system from one operating point to another as events occur.

3.3.11.9 Operating States

As already mentioned, the system supports multiple operating points. Some rules and mechanisms are required to move the system from one operating point to another. Each operating state is associated with an operating point. The system at a particular operating point is said to be in an operating state.

3.3.11.10 Policy Managers

A *policy* maps each operating state to a congruence class of operating points. The system supports multiple operating states and hence multiple operating points. At any point in time, the system operates using a single policy. For example, a power management strategy contains at least one policy, and may specify as many different policies as necessary for different situations. If multiple policies are needed, then a *policy manager* must exist in the system to coordinate the activation of different policies.

Figure 3-7 shows the high level design for DPM.

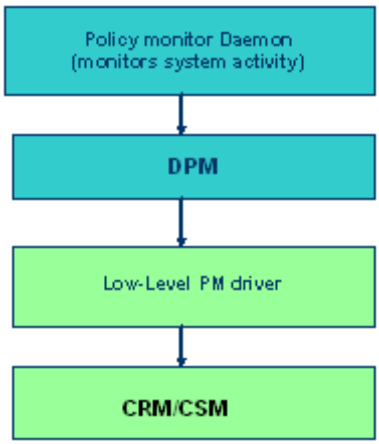


Figure 3-7. DPM High Level Design

Figure 3-8 specifies the DPM architecture block diagram.

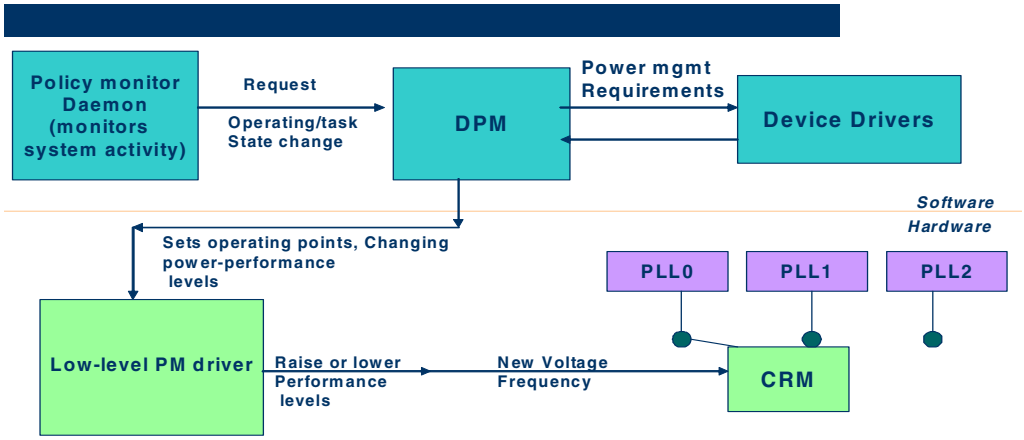


Figure 3-8. DPM Architecture Block Diagram

3.3.11.11 Low-Level Power Management Driver

The low-level power management driver is responsible for implementing hardware-specific operations to meet power requirements, and also to conserve power. Driver implementation may be different for different platforms. It is used by the DPM (Dynamic Power Management) layer. This driver implements dynamic voltage and frequency scaling (DVFS) or dynamic frequency scaling (DFS) techniques, depending on the platform, and low-power modes. The DVFS or DFS driver is used to change the frequency/voltage or frequency only when the DPM layer decides to change the operating point to meet the power requirements. This is done when the system is in RUN mode which helps in conserving power while the system is running. Low-power modes, such as WAIT and STOP are also implemented to save power. In all these cases, power consumption is achieved by reducing the voltage/frequency and the severity of clock gating.

3.3.11.12 Dynamic Voltage and Frequency Scaling (DVFS) Driver

The DVFS driver is responsible for varying the frequency and voltage of the ARM core. Other software modules interface to it through a custom, kernel-space API. The mode can be controlled manually through the API and automatically on those processors with the required monitor hardware.

3.3.11.13 Dynamic Process and Temperature Compensation (DPTC) Driver

The dynamic process and temperature compensation (DPTC) driver is responsible for varying the voltage of the system based on the speed of the actual silicon, which varies depending on temperature and where the specific IC device falls within the allowable process variation. It requires no API.

3.4 Boot Loaders

A boot loader is a small program that runs first after a CPU powers up. A boot loader is required to boot an ARM Linux system. The boot loader for ARM Linux serves two purposes:

- Sets up the system, such as the AHB Lite IP Interface (AIPS) and the Multi Layer Cross Bar Switch (MAX), memory, and different clocks
- Obtains proper information for the Linux kernel before jumping to it.

NOTE

Not all boot loaders are supported on all boards.

3.4.1 Functions of Boot Loaders

A boot loader provides the functions outlined in the following steps:

1. Set up AIPS and MAX
2. Set up Phase-Locked Loop (PLLs) for various system clocks
3. Set up and initialize the RAM
4. Initialize one serial port (optional)
5. Detect the machine type
6. Set up the kernel tagged list
7. Jump to the kernel image (either the `Image` file or the `zImage` file for compressed kernel)

The first step, setting up AIPS and MAX, is a required step for a boot loader to get access to proper peripherals, such as Timer and UART. The MAX should also be set up properly for different bus master priorities.

The second step, setting up the PLLs, is necessary because default PLL settings may not be optimal. The boot loader should tune the settings before trying to execute the image to set up the desired clocks.

For more information about steps three to seven, see the following directory:

```
<ltib_dir>/rpm/BUILD/linux-2.6.26/Documentation/arm/Booting
```

Note that in the last, jump to the kernel image, the boot loader calls the kernel image directly regardless of whether the kernel is compressed. For a compressed kernel (`zImage`), the expansion is done by the code surrounding kernel image during the kernel build.

The following boot loaders are provided in the BSP:

- RedBoot

RedBoot is the boot loader with the most features. RedBoot downloads images using either serial or Ethernet connections, handles image decompression, scripting and stores the image into Flash. RedBoot is mainly used for software development.

NOR Flash is controlled by the EIM module, while the NAND Flash is controlled by the integrated NAND Flash controller. NAND Flash is a sequential access device appropriate for mass storage of code and applications, while NOR Flash is a random access device appropriate for storage as well as execution of code and applications. Code stored on NAND Flash must be loaded into RAM for execution. For more information about these two Flash technologies, see http://www.linux-mtd.infradead.org/.

3.4.2 RedBoot

RedBoot is an open source boot firmware based on the eCos Hardware Abstraction Layer. It was designed to be very portable, extensible, and configurable. Some of the features are:

- Host connectivity through RS-232 or Ethernet
- Command line interface through RS-232 or Telnet
- Image downloads through HTTP, TFTP, X-Modem, or Y-Modem
- Support for compressed images (download and Flash load)
- Flash Image System for managing multiple Flash images
- Flash stored configuration
- Boot time script execution
- GDB (for debugging)
- BOOTP (for network booting)
- Watchdog servicing

RedBoot supports a wide variety of architectures and is very well documented. It is generally used for software development. For more information on RedBoot, see http://sources.redhat.com/redboot/.

LCD display is not supported.

3.5 Graphical User Interface

The GUI resides in the user-level application space and interacts with the Video drivers transparently. However, there are certain parts of the GUI that need to be ported, such as the touch screen driver and keypad driver.

3.5.1 Qt/Embedded

The following are the components of Qt/Embedded as a windowing manager:

- Qt/Embedded v2.3.10
- Qtopia v4.3

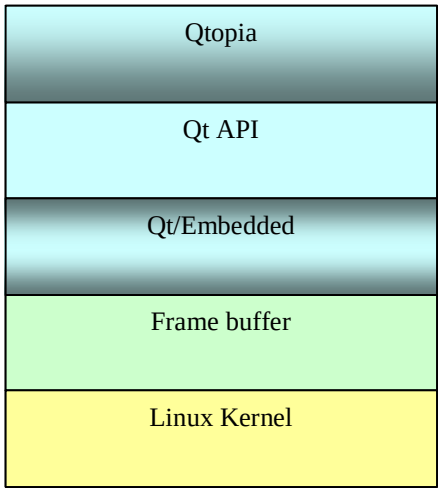


Figure 3-9. Qt/Embedded

3.6 Tools

GCC ARM cross-compiler tool chains are used for compiling the kernel, associated drivers, libraries, and applications. The pre-built tool chains are available in this release as described in Chapter 2, "Running Linux on the Hardware Boards" and in the `readme.html`. The tool chain runs on the Linux Host PC.

ARM ADS is used for kernel level debugging and GDB for application level debugging.

For more information, see Chapter 2, "Running Linux on the Hardware Boards" for more information.

3.7 Root File System

The Root file system is built as a `cramfs` or `JFFS2` image. `cramfs` is a Linux filesystem designed to be simple, small, and to compress things well. It is used on a number of embedded systems and small devices. To know more on deploying `cramfs`, see Section 2.1.5, "Booting Linux."

RAM is mounted as a `ramfs`. This is used for `/tmp`, `/var` and `/home`. There is also support for `ramdisk` and `jffs2` file systems. In a future release, the `/var` region may be mounted as `jffs2` so as to provide persistent storage for user data files.

The release also contains a `tar` file containing the root file system binaries.

3.7.1 Utilities

The cramfs is a read-only file system, so unlike other file systems it must be created along with its contents. The mkcramfs utility is used to construct a cramfs image which later can be written to Flash/ROM and mounted:

```
mkcramfs dir img.cramfs
```

where dir is the name of a directory containing the files and subdirectories to be added to the cramfs image, and img.cramfs is the name of the file to store the cramfs image.

mkcramfs runs on Linux 2.6. It is available on a standard Linux distribution. In some distributions it is named mkfs.cramfs.

3.7.2 Contents

Pre-built binaries for the standard applications and libraries available in the file system provided by LTIB. These include:

- base-files: This contains the basic root file system and configuration files
- busybox: Core of the system
- libc6: The C libraries from the latest stable arm-linux GNU tools release
- modutils: Linux kernel module support
- procps: /proc support utilities
- tinylogin

The binaries for the GUI are built from source. Qt/Embedded sources and can be obtained from http://www.trolltech.com.

3.8 Source of Linux BSP Components

Figure 3-10 shows the source of the code for each of the Linux BSP components.

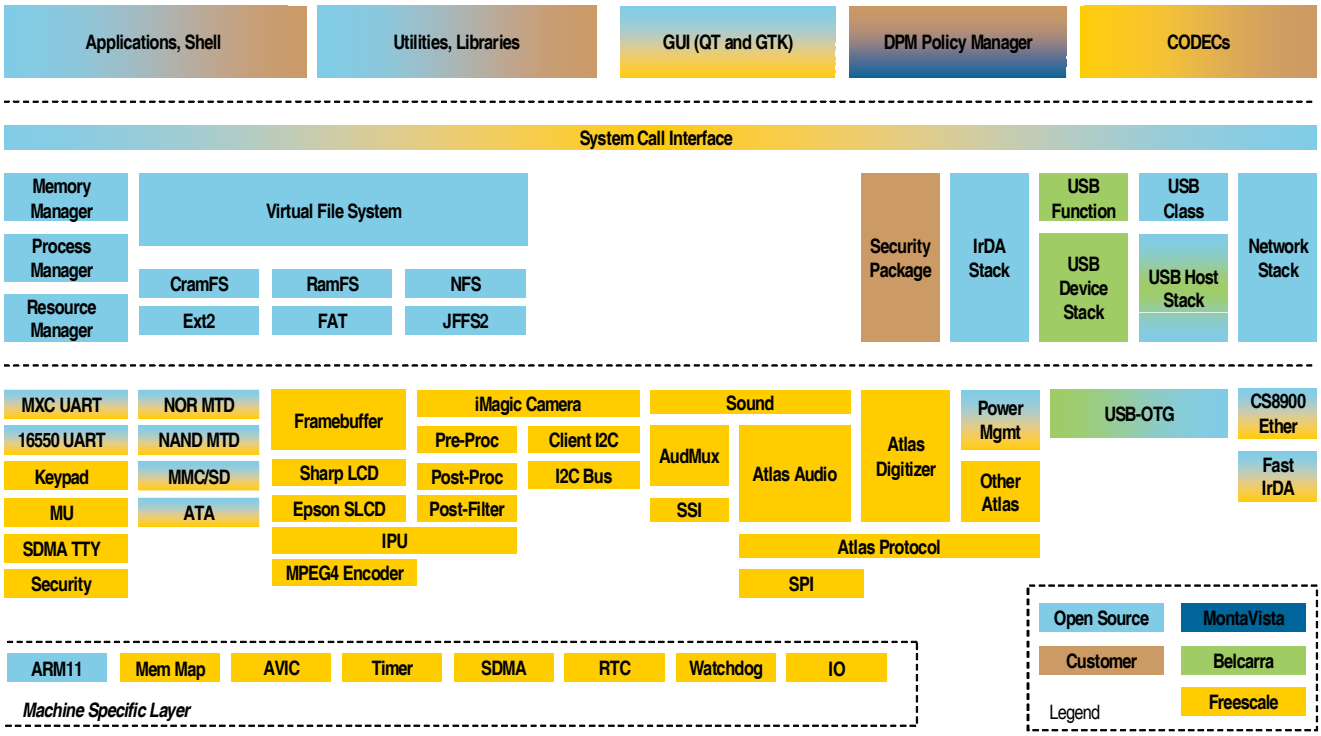


Figure 3-10. Linux BSP Source Diagram

3.9 Linux BSP APIs

Figure 3-11 shows a high level view of the Linux BSP components.

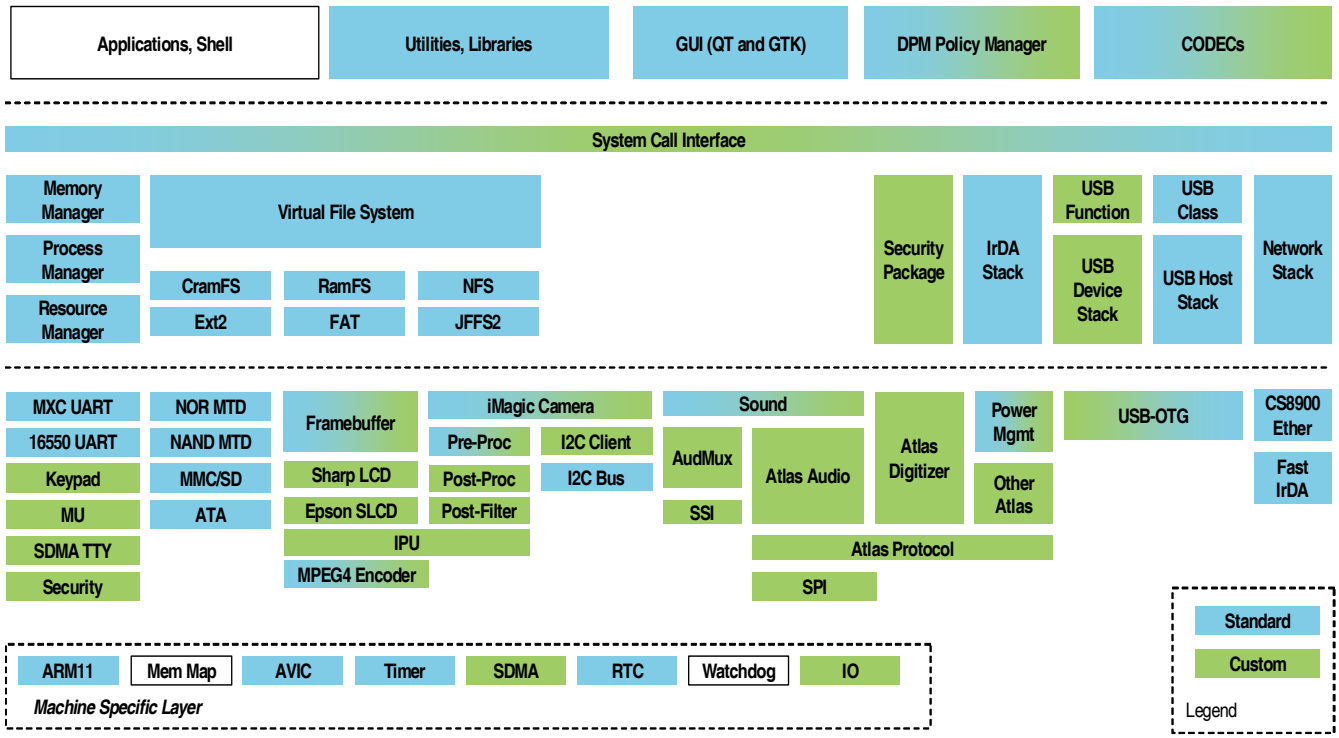


Figure 3-11. Linux BSP API Diagram

Table 3-2 lists the types of APIs that are exported by each of the Linux BSP components.

Table 3-2. List of Linux BSP Component APIs

SW Component	Kernel API	User API	Comment
MSL			
Interrupts	X		Linux Std
Timer	X		Linux Std
SDMA(DMA) API	X		FSL Custom
IOMUX	X		FSL Custom
GPIO	X		FSL Custom
SPBA	X		FSL Custom
Character Device Drivers			
16552 UART	X	X	Linux Std
MXC UART	X	X	Linux Std
Watchdog	X		Linux Std
RTC	X		Linux Std

Table 3-2. List of Linux BSP Component APIs (Continued)

SW Component	Kernel API	User API	Comment
Graphics Drivers			
Framebuffer		X	Linux Std with FSL Extensions
Sharp LCD	X		FSL Custom
Epson SLCD	X		FSL Custom
IPU	X		FSL Custom
VPU		X	FSL Custom
Multimedia			
iMagic Camera		X	V4L2 with FSL Extensions
Video Post-Processing			
Video Pre-Processing			
Video Post-Filtering			
MPEG4 VGA Encoder			
MPEG4/H.264 D1 CODEC		X	FSL Custom
MC13783			
MC13783 Protocol	X		FSL Custom
MC13783 Audio	X		FSL Custom
MC13783 Digitizer	X		FSL Custom
MC13783 RTC	X		FSL Custom
MC13783 Power Management	X		FSL Custom
MC13783 Connectivity	X		FSL Custom
MC13783 Battery	X		FSL Custom
MC13783 Light	X		FSL Custom
Sound Drivers			
Sound		X	ALSA with FSL Extensions
Input Device Drivers			
Keypad		X	FSL Custom
MTD Drivers			
NOR MTD		X	Linux Std
NAND MTD		X	Linux Std
Networking Drivers			
CS8900A Ethernet		X	Linux Std
Fast IrDA			Linux Std

Table 3-2. List of Linux BSP Component APIs (Continued)

SW Component	Kernel API	User API	Comment
Disk Drivers			
ATA			
USB Drivers			
USB Host Stack	X		Linux Std
USB Device Stack	X		Belcarra
USB Class Drivers		X	Linux Std
USB Function Drivers		X	Belcarra
USB-OTG (TDI)	X		Linux Std/Belcarra
USB-OTG (ARC)	X		Linux Std/Belcarra
Security Drivers			
SCC	X		FSL Custom
RNGA	X		FSL Custom
RTIC	X		FSL Custom
General Drivers			
MMC/SD		X	Linux Std
PCMCIA		X	Linux Std
Memory Stick			TBD
I2C Bus	X		Linux Std
I2C Client	X		FSL Custom
SDMA TTY		X	FSL Custom
AUDMUX	X		FSL Custom
SSI	X		FSL Custom
SPI	X		FSL Custom
Power Management			
Power Management		X	DPM with FSL Extensions
DVFS	X		FSL Custom
DPTC			No API
GUI			
Qt/E		X	Qt/E
GTK		X	GTK

Chapter 4
Machine Specific Layer (MSL)

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

- Interrupts including GPIO and EDIO (only on certain platforms)
- Timer
- Memory map
- General purpose input/output (GPIO) including IOMUX on certain platforms
- Shared peripheral bus arbiter (SPBA)
- Smart direct memory access (SDMA)

These modules are normally available in the following directory:

```
<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/plat-mxc
```

The header files are implemented under the following directory:

```
<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc
```

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 layer modules. First, the common modules, such as Interrupts and Timer are discussed. Next, the board-specific modules, such as Memory Map and general purpose input/output (GPIO) (including IOMUX on some platforms) are detailed. Because of the complexity of the SDMA module, its design is explained in a separate chapter.

Each of the following sections contains an overview of the hardware operation. For more information, see the corresponding IC Specification document.

4.1 Interrupts

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

4.1.1 Interrupt Hardware Operation

The Interrupt Controller controls and prioritizes a maximum of 64 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's 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.

4.1.2 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`). 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-2.6.26/Documentation/arm/Interrupts
```

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

4.1.3 Interrupt Requirements

The interrupt implementation meets the following requirements:

- The interrupt module implements the Interrupt Controller interrupt disable and enable functions.
- The interrupt module implements all the functions required by the Linux interrupt architecture as defined in the standard ARM interrupt source code (mainly the `<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/kernel/irq.c` file).

4.1.4 Interrupt Source Code Structure

The interrupt module is implemented in the following file:

```
<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/plat-mxc/irq.c
```

There are also two header files:

```
<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/hardware.h
<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/irqs.h
```

Table 4-1 lists the source files for interrupt.

Table 4-1. Interrupt Files List

File	Description
`hardware.h`	register descriptions
`irqs.h`	declarations for number of interrupts supported
`irq.c`	actual interrupt functions

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

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

4.2 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). Once 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 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 milliseconds) and is used by the Linux kernel.

4.2.1 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.

4.2.2 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 4.2, “Timer.” Another function provides the time elapsed as the last timer interrupt.

4.2.3 Timer Requirements

The timer implementation meets the following requirements:

- The timer module implements all the functions required by Linux to provide the system timer and dynamic timers.
- The timer is set up to generate an interrupt every 10 ms.

4.2.4 Timer Source Code Structure

The timer module is implemented in `arch/arm/plat-mxc/time.c` file. The source file for the timer is `time.c` and it describes timer function implementation.

4.2.5 Timer Programming Interface

All the timer functions required for the Linux port are implemented in the `time.c` file.

4.3 Memory Map

As the Linux kernel is running under the virtual address space with the Memory Management Unit (MMU) on, a predefined virtual to physical memory map table is required for the device drivers to access to the device registers.

4.3.1 Memory Map Hardware Operation

The MMU (Memory Management Unit) 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.

4.3.2 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-2.6.26/arch/arm/mach-mx3/mm.c`file.

4.3.3 Memory Map Requirements

The Memory Map implementation should meet the requirement where the Memory Map module creates the physical to virtual memory map for all the I/O modules.

4.3.4 Memory Map Source Code Structure

The Memory Map module implementation is in `mm.c` under the platform-specific MSL directory. The `<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/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. Table 4-2 lists the source file for the Memory Map.

Table 4-2. Memory Map File List

File	Description
mx31.h	Header files for the IO module physical addresses.
mm.c	Memory map definition file

4.3.5 Memory Map Programming Interface

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

4.4 IOMUX

The limited number of pins of highly integrated processors can have multiple purposes. The IOMUX module controls a pin's 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 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's operation is controlled by a specific hardware module. A GPIO pin, is controlled by the user through software with further configuration through the GPIO module. For example, the TXD1 pin might have the following functions:

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

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose and cannot be changed by software. Otherwise, the IOMUX module needs to be configured to serve a particular purpose that is dictated by the system (board) design. If the pin is connected to an external UART transceiver and therefore to be used as the UART data transmit signal, it should be configured as the primary function. If the pin is connected to an external Ethernet controller for interrupting the ARM core, 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 a pin's usage according to the system design.

4.4.1 IOMUX Hardware Operation

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

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 a pin's input path. This register is only required when multiple pads drive the same internal port.

SW_PAD_CTL -- Used to control a pad's: slew rate, driver strength, pull-up/down resistance, etc.

4.4.2 IOMUX Software Operation

The iomux software implementation provides an API to setup a pin’s functionality and pad features.

4.4.3 IOMUX Requirements

The iomux implementation should meet the requirements where the iomux module implements all the functions to configure the pins that are supported by the hardware.

4.4.4 IOMUX Source Code Structure

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

```
<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx*/
```

(where "*" stands for a specific CPU).

Table 4-3 lists the source files for the iomux.

Table 4-3. IOMUX File List

File	Description
`iomux.c`	iomux function implementation
`mx*_pins.h`	pin definitions in the iomux_pins enum

4.4.5 IOMUX Programming Interface

All the iomux functions required for the Linux port are implemented in the `iomux.c` file.

4.4.6 IOMUX Control through the GPIO Module

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

For a multi-purpose pin, the GPIO controller provides the multiplexing control so that each pin may be configured either as a functional pin (which can be subdivided into either major function or one alternate function) whose operation is controlled by a specific hardware module, or it can be configured as a GPIO pin, in which case, the pin is controlled by the user through software with further configuration through

the GPIO module. In addition, there are some special configurations for a GPIO pin (which can be subdivided into not only output based A_IN, B_IN, C_IN or DATA register, but input based A_OUT or B_OUT).

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose which can not be changed by software; otherwise, the GPIO module needs to be configured properly to serve a particular purpose that is dictated with the system (board) design: if this pin is connected to an external UART transceiver, it should be configured as the primary function; if this pin is connected to an external Ethernet controller for interrupting the core, 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 a pin for that usage according to the system design.

4.4.6.1 GPIO Hardware Operation

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

4.4.6.1.1 Muxing Control

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

4.4.6.1.2 PULLUP control

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

4.4.6.2 GPIO Software Operation

The GPIO software implementation provides an API to setup a pin’s functionality and pad features.

4.4.6.3 GPIO Requirements

The GPIO implementation should meet the requirement where the GPIO module implements all the functions to configure the pins that are supported by the hardware.

4.4.6.4 GPIO Source Code Structure

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

```
<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx*/
```

Table 4-4 lists the source files for the IOMUX.

Table 4-4. IOMUX File List

File	Description
mx3_3stack_gpio.c	iomux function implementation
mx*_pins.h	pin name definitions

4.4.6.5 GPIO Programming Interface

All the GPIO muxing functions required for the Linux port are implemented in the gpio_mux.c file.

4.5 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, a pin's state (high or low) can be controlled by writing to an internal register; when configured as an input, a pin's input state can be read from an internal register.

4.5.1 GPIO Software Operation

The general purpose input/output (GPIO) module provides an API to configure the i.MX processor's 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 will not be initialized by a dedicated hardware module. Setting the pad's pull-up, pull-down, slew rate, etc. with the pad control function may be required as well.

4.5.1.1 API for GPIO

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

The following sub-sections shows examples of using this API.

4.5.1.2 IOMUX/GPIO API Usage—Example 1

To configure MX31_PIN_GPIO1_3 for an MC13783 interrupt: gpio_mc13783_active()from <ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3/mx31ads_gpio.c configures the pin as GPIO, sets its direction to be input and enables the interrupt at the rising edge of the signal.

```
/*!
 * This function configures the IOMux block for MC13783 standard operations.
 *
 */
void gpio_mc13783_active(void)
{
    mxc_request_iomux(MX31_PIN_GPIO1_3, OUTPUTCONFIG_GPIO,
                                                    INPUTCONFIG_GPIO);
    mxc_set_gpio_direction(MX31_PIN_GPIO1_3, 1);
    mxc_set_gpio_edge_ctrl(MX31_PIN_GPIO1_3, GPIO_INT_RISE_EDGE);
}
```

To register an interrupt service routine, it could call:

```
    ...
    ret = request_irq(PMIC_INT_LINE, mc13783_irq_handler, 0, 0, 0);
```

and PMIC_INT_LINE would be defined as:

```
#define PMIC_INT_LINE          IOMUX_TO_IRQ(MX31_PIN_GPIO1_3)
```

In this example, the same IOMUX pin name is used for both IOMUX calls and the GPIO calls, and the standard request_irq() function for registering an interrupt handler is called. The interrupt number when calling this function has to be converted with the IOMUX_TO_IRQ macro as shown above. Ideally, if this pin is not used when the driver is unloaded, mxc_free_iomux() should be called to free the pin, otherwise a warning will be displayed when another driver tries to make use of this pin when it calls mxc_request_iomux().

4.5.1.3 IOMUX/GPIO API Usage—Example 2

To configure the RXD1 pin for the functional mode used by the UART driver, gpio_uart_active() calls:

```
...
mxc_request_iomux(MX31_PIN_RXD1, OUTPUTCONFIG_FUNC, INPUTCONFIG_FUNC);
```

In the gpio_uart_inactive() function (which will be called when the driver is shutdown) these two calls are needed to put this pin in the GPIO input mode and then release the ownership of this pin:

```
...
mxc_request_gpio(MX31_PIN_RXD1);
mxc_free_iomux(MX31_PIN_RXD1, OUTPUTCONFIG_GPIO,
```

4.5.2 GPIO Requirements

This GPIO implementation should meet the following requirements:

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

4.5.3 GPIO Source Code Structure

All of the GPIO module source code is in the MSL layer, in the following file:

`<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/plat-mxc/gpio.c`

Includes are available in the following files:

`<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/gpio.h`
`<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx*/mx*_pins.h` (where * stands for a specific CPU)

Table 4-5. GPIO File List

File	Description
`mx*_pins.h`	GPIO private header file
`gpio.h`	GPIO public header file
`gpio.c`	Function implementation

4.5.4 GPIO Programming Interface

For more information, see the API documents for the programming Interface.

4.6 EDIO

Not all platforms have the EDIO hardware module. This section applies only to those that do.

The EDIO module provides external interrupt capability to the processors.

4.6.1 EDIO Hardware Operation

The interrupt (EDIO) module recognizes the external asynchronous signal as an interrupt source. When it matches the selected criteria, low level or edge (rising, falling or both edges), it asserts an interrupt request to the processor's interrupt controller. This module can handle eight such interrupts simultaneously with selectable configurations for each incoming signal reaching EDIO.

4.6.2 EDIO Software Operation

The EDIO interrupt has been integrated into the generic platform level interrupt implementation as in `irq.c` in the `<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/plat-mxc` directory. For drivers that need to set up the interrupt attributes, such as interrupt edges or levels, the `set_irq_type()` can be called. The interrupt clearing that is needed for the EDIO interrupts is hidden from the driver.

4.6.3 EDIO Requirements

This EDIO implementation should meet the following requirement where the EDIO module provides a method to set the EDIO interrupt attributes provided by the hardware.

4.6.4 EDIO Source Code Structure

All of the EDIO module source code is in the `irq.c` is under the `<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/plat-mxc/` directory and the `hardware.h` is under the `<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/` directory.

Table 4-6. EDIO File List

File	Description
`platform.h`	EDIO interrupt defines
`irq.c`	Common functions for various boards

4.6.5 EDIO Programming Interface

For more information, see the API documents for the programming Interface.

4.7 SPBA Bus Arbiter

Note that not all platforms have the SPBA hardware module. Therefore, this section may only apply to the platforms with SPBA module in them.

The SPBA bus arbiter provides arbitration mechanism among multiple masters to have access to the shared peripherals.

4.7.1 SPBA Hardware Operation

The SPBA is a three-to-one IP Sky-Blue line interface (IP-Bus) arbiter, with a resource locking mechanism. The masters can access up-to thirty-one shared peripherals through the SPBA. It has the following features:

- Multi-master bus arbiter
- 32-bit data access
- Supports up to 31 shared peripherals, each consuming 16 KB of address space
- Can be considered as the 32nd peripheral, used for resource ownership and access control mechanism to the 31 peripherals
- Provides 31 sets of Out of Band Steering Control signals to the off-module steering logic
- Operating frequency up to 67 MHz,
- Clocks: ipg_clk, ipg_clk_s (mcu clock domain).

4.7.2 SPBA Software Operation

Functions are provided to allow different masters to take/release ownership of a shared peripheral. These functions are also exported to be used by other loadable modules.

4.7.3 SPBA Requirements

This SPBA implementation should meet the following requirements where:

- The SPBA module provides an API to allow different masters to take/release ownership of a shared peripheral.
- The SPBA module conforms to the Linux coding standard as documented in the *Coding Conventions* chapter.

4.7.4 SPBA Source Code Structure

All of the SPBA module source code is in the MSL layer.

The following files are available within the directories indicated:

```
<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/plat-mxc/spba.c
<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/spba.h
```

Table 4-7. SPBA File List

File	Description
`spba.h`	SPBA public header file
`spba.c`	Common SPBA functions

4.7.5 SPBA Programming Interface

For more information, see the API documents for the programming interface.

Chapter 5
Smart Direct Memory Access (SDMA) API

5.1 Overview

SDMA API driver controls the SDMA hardware. It provides an API to other drivers for transferring data between MCU memory space, DSP memory space and 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

5.1.1 Hardware Operation

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

- 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 2-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.

5.1.2 Software Operation

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

Complete support for SDMA is provided in three layers (see Figure 3-2). The first layer is the I.API, the second layer is the Linux DMA API and the third layer is the TTY driver or DMA-capable drivers, such as ATA, SSI and the UART driver. 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 5-1 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 5-1. SDMA Channel Usage

Driver Name	No. of SDMA Channels	SDMA Channel Used
SDMA TTY	8	Static Channel allocation -- uses SDMA channels 1, 2, 3, 4, 5, 6, 7, 8
Sound	2 per device	Dynamic channel allocation
UART	2 per device	Dynamic channel allocation
MMC	1 per device	Dynamic channel allocation
Fast IR (FIRI)	2 per device	Dynamic channel allocation
DVFS	1	Dynamic channel allocation

5.2 Source Code Structure

The source file, `sdma.h` (header file for SDMA API) is available in the directory `/<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc`.

Table 5-2 lists the source files available in the directory, `/<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/plat-mxc/sdma`.

Table 5-2. SDMA API Source Files

File	Description
`sdma.c`	SDMA API functions
`sdma_malloc.c`	SDMA functions to get DMA'able memory
`iapi/`	iAPI source files

Table 5-3 lists the header files available in the directory, `/<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3/.`

Table 5-3. SDMA Script Files

File	Description
`sdma_script_code.h`	SDMA RAM scripts for SDMA ROM Pass 1
`sdma_script_code_pass2.h`	SDMA RAM scripts for SDMA ROM Pass 2

5.3 Configuration

5.3.1 Linux Menu Configuration Options

In order to get to the SDMA configuration use the command `./ltib -c` when located in the <`ltib dir`>. In the screen select `configure kernel`, exit and a new screen will appear.

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`. By default, this option is Y for all architectures.

5.4 Programming Interface

The module implements custom API and partially standard DMA API. Custom API is needed for supporting non-standard DMA features like 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 methods implemented in the driver.

5.5 Example Usage

Refer to one of the drivers from Table 5-1 that uses the SDMA API driver for a usage example.

Chapter 6
PMIC Protocol Driver

This chapter describes the Power Management Integrated Circuit (PMIC) protocol device driver for Linux that provides the low-level read/write access to the PMIC's hardware control registers.

One key objective of the PMIC protocol driver and the other PMIC-related drivers is to provide a complete API interface to all supported PMIC chips, despite differences in hardware design and implementation. This is necessary to minimize the effort to design, implement, test, and support PMIC device drivers.

With a single API interface, a single application can be reused without any changes across all supported PMIC chips. Such an application, however, must either restrict itself to a core set of features supported by all PMIC chips, or detect at runtime which PMIC chip is installed before performing any PMIC-specific operations.

This chapter describes the requirements, design, implementation, and client API that is provided for accessing PMIC hardware. Additional information about the PMIC device driver APIs, especially programming-related details, can also be located in the Doxygen-generated HTML documentation that is provided with the Linux BSP distribution. As shown in Figure 6-1, the PMIC protocol driver handles all low-level communications between many other Linux device drivers and the PMIC hardware. The PMIC protocol driver uses one of the available SPI buses to communicate with the PMIC chip.

NOTE

The PMIC protocol driver is intended only for use with the MCU core and the Linux OS. An equivalent PMIC driver for the DSP core within a dual core platform is beyond the scope of this document.

6.1 Key PMIC Features and Capabilities

The PMIC protocol typically provides hardware to support the following functions for Freescale's i.MX-based platforms:

- Audio playback and recording
- Power supply control, battery charging, and power management support
- Analog-to-digital conversion (including touchpanel support)
- External RS-232 and USB OTG connectivity
- LED and LCD backlight control
- Real-time clock (RTC) support
- Event notification through the use of hardware interrupts

These functions are all selected and configured through the PMIC control registers, which are accessible through two separate SPI interfaces. The Primary SPI interface initially has full read/write access to the

PMIC control registers, while the Secondary SPI interface initially has only read access but can be granted selective read/write access. When used with dual core platforms, the PMIC can be controlled by both the MCU and the DSP through their respective SPI interfaces. Depending upon the actual system requirements, either the MCU or the DSP can be designated as the Primary Processor and connected to the PMIC through the Primary SPI bus. The other processor would then be designated as the Secondary Processor and be connected to the PMIC through the Secondary SPI bus. For single core platforms, only the Primary SPI interface to the PMIC is typically used.

Figure 6-1 shows the main functional blocks provided by the PMIC.

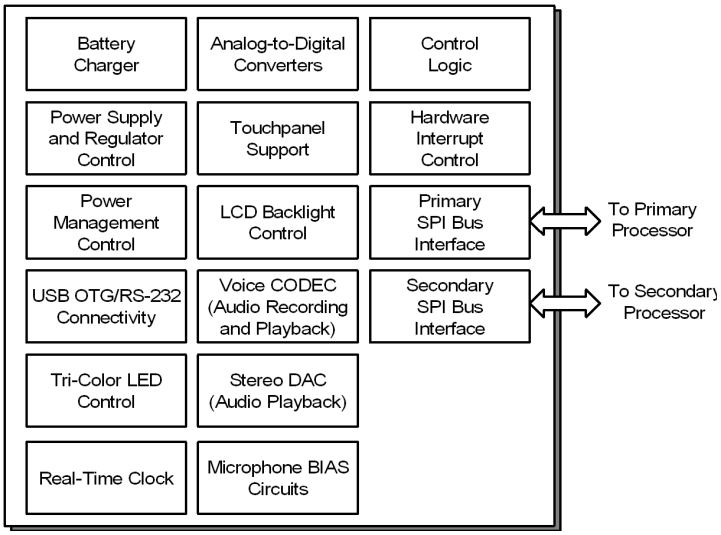


Figure 6-1. PMIC Block Diagram

Note that not all of the functions can be used at the same time because of hardware constraints. For example, some of the I/O pins are shared between the USB OTG and RS-232 transceivers. Therefore, USB OTG and RS-232 connectivity cannot be used at the same time, although it is certainly possible to switch between the two modes. For the sake of simplicity, only the SPI bus interfaces are shown in Figure 6-2 and all of the other PMIC data buses and external I/O connections have been omitted.

Table 6-1 provides a brief description of the PMIC functional blocks for which Linux device drivers have already been implemented. Additional information about the device drivers for each of these PMIC functional blocks can be located in this reference manual.

Table 6-1. Summary of all Available PMIC Client Device Drivers

PMIC Device Driver	Functions
Power Management Driver	• Battery charger interface for wall charging and USB charging. • Regulators with internal and external pass devices. • Power up and power down control.
Analog-to-Digital Conversion (ADC) Driver	• 10-bit ADC for battery monitoring and other readout functions. • Touch screen interface.

Table 6-1. Summary of all Available PMIC Client Device Drivers

PMIC Device Driver	Functions
Audio Driver	• Audio input amplifier selection and gain control. • Microphone bias circuit control. • Audio output amplifier selection and gain control. • Audio output hardware mixing and mono adder control. • 13-bit Voice CODEC supporting playback and recording at either 8 kHz or 16 kHz sampling rates. • 13-bit Stereo DAC supporting playback at multiple sample rates.
RTC Driver	• Real-time clock support (MC13783 PMIC only).
Backlight and LED Driver	• Manages the LCD backlight level and each of the Red, Green, and Blue LEDs.
Connectivity Driver	• USB OTG and RS-232 transceiver control. • USB OTG device insert/removal detection and notification. • USB OTG connection negotiation and voltage level control.
Battery Driver	• Configures the battery control/monitoring interface.

6.1.1 PMIC Register Access and Arbitration

The main purpose of the PMIC protocol driver is to provide the necessary read/write access to the PMIC control registers using the SPI bus interfaces to support all of the higher-level PMIC client drivers that are shown in Figure 6-1 and are briefly described in Table 6-1. There are two possible techniques for accessing the PMIC control registers: exclusive sharing and logic sharing. For each PMIC control register, choose either of the following techniques:

- **Exclusive Sharing**—One processor has exclusive control. The processor connected to the primary SPI bus interface determines which processor has control by setting the appropriate arbitration control bits. Only the designated processor can modify the register. By default, only the primary SPI has read/write access to the PMIC control registers, while the secondary SPI has only read access. However, some of the PMIC control registers and settings cannot be accessed at all from the secondary SPI, regardless of the arbitration bit settings. See the appropriate PMIC detailed technical specifications (DTS) document for complete information about primary versus secondary SPI bus access to the control registers.
- **Logic Sharing**—Control is determined by analyzing logical expressions. Values of both the primary and secondary control register settings are logically ANDed or ORed to create the final resource control value. Logic Sharing of a resource, through either a single bit or a multi-bit vector, can also be selected by setting the appropriate arbitration bit values through the primary SPI interface.

Immediately following a Power up or Reset event, the processor that is connected to the Primary SPI interface can modify the PMIC control registers to configure the desired access mode for control registers. The specific registers that need to be updated and the appropriate arbitration bit values can be located in the Detailed Technical Specifications document for the PMIC.

PMIC register access and arbitration settings are not issues on platforms where Linux is running on the primary processor. However, where Linux is running on the secondary processor (on a dual-core platform),

additional steps must be taken to provide the required level of access to the PMIC registers from the secondary SPI interface. The following options may be used to resolve this issue:

- Modify the platform or the PMIC hardware to swap the primary and secondary processor connections.
- Implement additional software on the primary processor (which is not running Linux in this case) to grant the secondary processor the required access rights to the PMIC control registers.

The second option is typically the preferred solution. However, in situations where additional software development on the primary processor (usually DSP core, but note that the i.MX31 does not use an extra DSP core) is not practical in the short-term, then a possible interim solution is to modify the PMIC hardware so that both the primary and secondary SPI interfaces are connected to a secondary processor running Linux (for example, by connecting both CSPI1 and CSPI2 from the ARM core to the PMIC). A single function can be implemented that will be called during the Linux boot process and use the primary SPI interface to reconfigure the PMIC arbitration bits as required. This allows the rest of the Linux system to operate properly using only the secondary SPI interface, after the boot process has been completed.

Note that this is strictly an interim solution for getting Linux to run properly on the secondary processor with full PMIC functionality. This hardware change to the PMIC SPI interfaces completely disconnects the DSP core from the PMIC and, therefore, cannot be used as a true solution to the arbitration problem. Ultimately, implementing the appropriate software on the primary processor to reconfigure the PMIC arbitration settings is the only appropriate solution when Linux is running on the secondary processor.

6.1.2 Interrupt Notification

Events are reported to either the primary or secondary processor through the use of a PMIC-generated hardware interrupt. A single interrupt signal can indicate one or more events. The PMIC protocol driver first receives the interrupt signal and then checks the PMIC's interrupt status register to determine exactly which events are being signaled. Finally any client-registered callback functions are called to complete the handling of the event. If no callback functions are currently registered, then the event is ignored.

Table 6-2 lists all events that the MC13783 PMIC protocol driver supports.

Table 6-2. MC13783 PMIC Hardware Interrupt Events

Event	Description
ADC has finished requested conversions	ON1B event
Touchscreen wake up	ON2B event
ADC reading above high limit	ON3B event
ADC reading below low limit	System reset
Charger attach	SW1A low setting stabilized
Charger over voltage detection	SW1A high setting stabilized
Charger path reverse current	SW1B low setting stabilized
Charger path short circuit	SW1B high setting stabilized

Table 6-2. MC13783 PMIC Hardware Interrupt Events (Continued)

Event	Description
BP regulator in regulation	SW2A low setting stabilized
Dual path selection	SW2A high setting stabilized
End of trickle charge	SW2B low setting stabilized
End of life / low battery detect	SW2B high setting stabilized
USB 4V detect	Thermal warning
USB 2V detect	Power cut event
USB 1V detect	Warm start event
Microphone bias 2 detect	Memory hold event
Headset attach	Clock source change
Stereo headset detect	Semaphore cleared
Thermal shutdown Asp	ICTEST state
Short circuit on Ahs outputs	CHRGMOD state
1 Hz time tick	USBMOD state
Time of day alarm	BOOT state
Wake up event	SW1A and SW1B joined

6.2 Driver Requirements

The PMIC protocol driver module (also called the “core” driver in the Linux source tree) is responsible for providing two types of services for all of the PMIC client driver components:

- Control Services
- Event Notification Services

The PMIC protocol driver may be built as a Linux loadable kernel module and manually loaded following system boot. However, the protocol driver is typically configured to be built into the Linux kernel image itself, because the PMIC card is not intended to be dynamically added or removed once the system has been powered on. Also, some of the Linux power management functions require that the PMIC protocol driver be properly loaded and fully operational.

6.2.1 Control Services

The key control services provided by the protocol device driver are:

- The ability to configure the SPI bus driver to communicate with the PMIC.
- The ability to read the current value of any PMIC hardware control register, by initiating the appropriate SPI bus transaction.
- The ability to write new values to any PMIC hardware control register, by initiating the appropriate SPI bus transaction.

- As the SPI bus transactions are asynchronous in nature, the PMIC protocol driver must not disable interrupts or be operating in an atomic context when making calls to the SPI driver.

Note that both the read and write capabilities may be affected by the Primary and Secondary SPI bus arbitration settings.

6.2.2 Event Notification Services

The PMIC protocol device driver must support the following event notification services:

1. Register a default interrupt handler to handle all PMIC-related hardware interrupt events.
2. Allow other PMIC client drivers to subscribe and unsubscribe to one or more PMIC events and to specify an appropriate “callback” function.
3. Call all previously registered callback functions when the corresponding PMIC event has been received.
4. The ability to properly set the PMIC interrupt event mask register to selectively control which hardware events are enabled or disabled.
5. The ability to query the PMIC interrupt status register to determine which events are being signaled by the current hardware interrupt.

6.2.3 Miscellaneous Requirements

In addition to the specific services-related requirements given above, the PMIC protocol driver must also satisfy the following additional requirements:

- Be able to properly reconfigure the PMIC arbitration settings (if required) to support the functionality that is expected by the rest of the Linux system.
- Conform to the Linux coding standards.

6.3 Driver Software Operation

The PMIC protocol driver controls the PMIC by reading and writing the PMIC hardware control registers. Both read and write access to the PMIC hardware control registers is done through the SPI driver. The PMIC protocol driver requires the SPI driver to perform all of the following functions:

- Create the proper data packets for transmission on the SPI bus. This includes putting the proper destination address for accessing a specific PMIC control register.
- Send the data packet and verify its transmission status.
- Receive and decode any data packets that were sent by the PMIC.
- Return any data received from the PMIC hardware back to the PMIC protocol driver.

Figure 6-2 shows the relationship between the PMIC protocol driver and all of the other related device drivers in the system as well as the interaction between them.

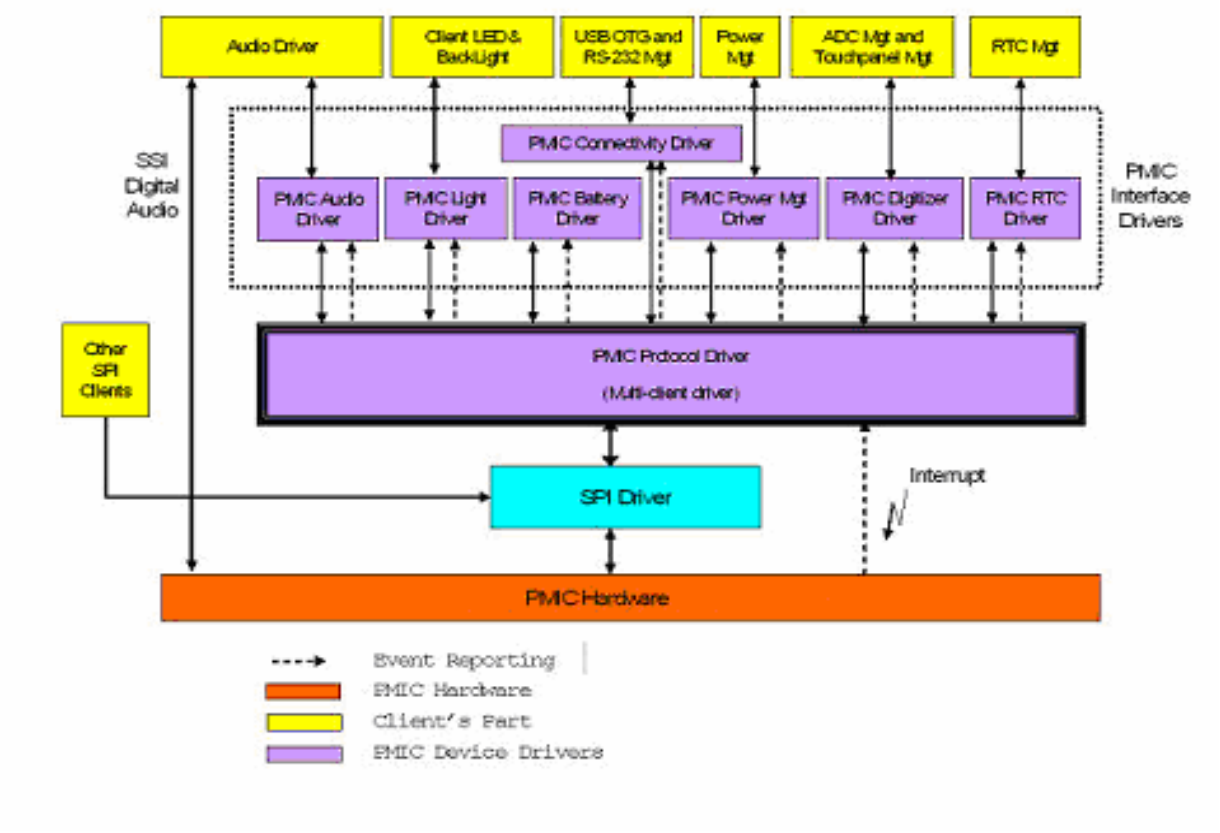


Figure 6-2. PMIC Device Driver

The hardware interrupt signal that can be generated by the PMIC is first received and handled by the PMIC protocol driver. The PMIC protocol driver determines events that are being signaled by the PMIC by examining the PMIC's interrupt status register. Finally, all PMIC interface drivers that have previously registered for the currently active events are signaled through their respective callback functions.

6.4 Driver Architecture

Figure 6-3 shows the overall architecture and external interfaces for the PMIC protocol driver.

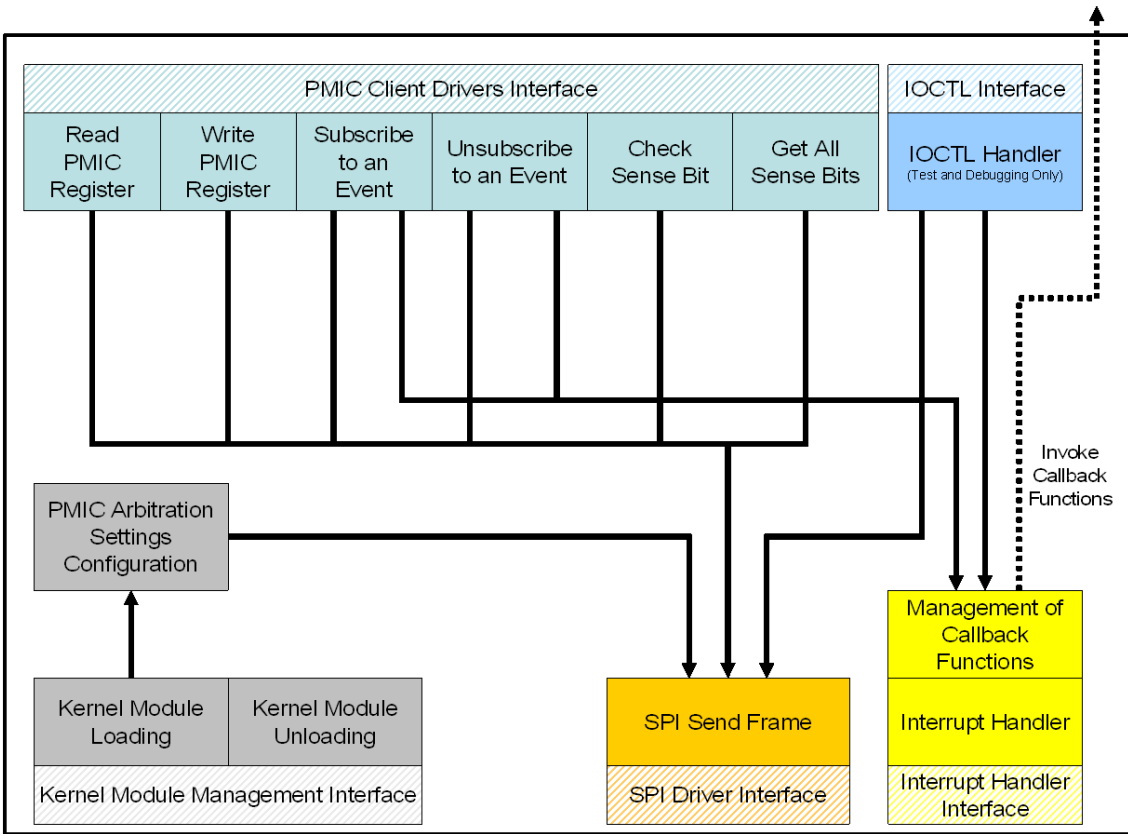


Figure 6-3. PMIC Protocol Driver Architecture and Interfaces

The key components are as follows:

- Read/Write interfaces for the PMIC control registers, subscribing/unsubscribing to PMIC events, and checking on one or more of the PMIC sense bits.
- PMIC device interface supporting the Linux IOCTL interface, for use with `/dev/pmic` device.
- Interface for sending and receiving SPI data packets to and from the PMIC.
- API for hardware interrupt notifications and which will then invoke the appropriate event callback functions.
- API supporting the Linux kernel module, loading/unloading operations and device driver initialization requirements.
- Internal function, called only during device driver initialization, that reconfigures the arbitration bits on the PMIC, if necessary, to support proper operation from the secondary processor.

Each of these main device driver components will be described in greater detail, including any implementation-specific issues, in the following section.

6.5 Driver Implementation Details

This section describes implementation-specific details associated with the PMIC protocol driver. The device driver source files should also be consulted to fully understand the implementation of the PMIC protocol driver. The CSPI driver documentation and sources should also be consulted if required.

6.5.1 Driver Initialization

The PMIC protocol driver performs the following operations when it is first loaded/initialized:

- Create either a `/dev/pmic` character device entry, depending on which version of the driver is actually being loaded, and register the new device with the kernel.
- Perform any required PMIC arbitration fixes (see Section 6.1.1, “PMIC Register Access and Arbitration”)
- Initialize the PMIC registers to a known state (optionally done here; or can be done by the individual PMIC client drivers on a component-by-component basis).
- Initialize all driver-specific global variables.
- Enable the PMIC hardware interrupt line and bind it to the top half interrupt handler (see Section 6.5.4.1, “Top Half Interrupt Handler”).

6.5.2 Driver Unloading

The following operations are performed when unloading/deinitializing the PMIC protocol driver:

- Remove the `/dev/pmic` device entry and tell the kernel to deregister this device.
- Disable the PMIC hardware interrupt line to prevent any further interrupts from occurring.

6.5.3 Event Notification List

The PMIC protocol driver uses a static array of `list_head` to manage the event notification list. The subscript of the array corresponds to a specific event ID, and each array element is actually the head of a linked list. Each element of the linked list contains all the information needed to invoke a callback function. Initially the array of `list_head` is initialized to indicate that all of the linked lists are currently empty and that no callback functions are currently registered.

Whenever an event callback function is to be registered, a new linked list element consisting of a structure with the following fields is allocated:

- A pointer to a callback function that takes a single (void *) argument and which does not return anything.
- A (void *) field that holds that argument that is to be used when invoking the callback function.
- A pointer to the next callback data structure for the same event.

This structure element is then initialized with the proper values and added to the appropriate linked list in the array of event notification lists.

When a PMIC-generated hardware interrupt arrives, the interrupt handler starts by examining the PMIC’s interrupt status register to determine the currently active events. The corresponding elements in the array

of event notification lists are then examined to see if any callback functions have been registered and, if so, they are all invoked in the same order that they were registered.

De-registering a callback function simply involves removing the callback data structure by adjusting the linked list pointers and then deallocating the memory for the callback data structure.

The only important thing to keep in mind with the handling of the event notification list is that it must always be kept in a consistent state and that any possible race conditions must be prevented. This basically means that all of the following scenarios must be properly handled:

- Registration and deregistration of callback functions must always be performed in a critical section, so that the array of pointers and the associated linked lists are always kept in a consistent state. This also avoids any possible memory leaks during allocation and deallocation of the memory required for the linked list elements. As part of the callback registration and deregistration process, calls to the SPI driver must be made to update the PMIC's interrupt mask register. When calls are made to the SPI driver, interrupts must be enabled, which eliminates atomic contexts. Therefore, the critical section must be implemented using only a mutex and not a spinlock.
- Callback function registration, deregistration, and the interrupt handler must all use a critical section when accessing the array of pointers and the linked list of callback data structures. As the interrupt handler is involved here, spinlocks must be used to implement the critical section. Fortunately, the interrupt handler itself does not require making any calls to the SPI driver, so running in an atomic context does not cause any problems.

These two requirements specify that a mutex must be used to guard against race conditions between callback registration and deregistration operations. Furthermore, within the mutex critical section, a spinlock must be used to guard against race conditions when the contents of anything in the event notification list (either the array of pointers or the associated linked lists) are used or modified. However, the spinlock can be released as soon as modifying the event notification list is no longer required, and the mutex can be used to perform any operation that is not directly associated with or impacted by the interrupt handler.

6.5.4 Interrupt Handler

The PMIC interrupt handler is divided into two parts. The "top half" is called directly by the Linux kernel when the hardware interrupt is first raised, and all interrupts are disabled while the "top half" interrupt handler is executing. The "top half" acknowledges and handles the interrupt.

However, if handling the interrupt also requires significant processing or other, possibly time-consuming operations, then all such operations should be deferred to a separate "lower half" interrupt handler that can be executed at a lower priority and with hardware interrupts re-enabled.

6.5.4.1 Top Half Interrupt Handler

The top half interrupt handler in the PMIC protocol driver performs the following operations:

1. Acknowledges and clear the hardware interrupt condition.
2. Schedules a work queue task to complete the handling of the interrupt event.

Note that PMIC-related interrupts typically do not have any hard real-time requirements. Therefore, it is perfectly acceptable to defer much of the interrupt handling to a separate work queue task.

6.5.4.2 The Lower Half Interrupt Handler

The lower half interrupt handler for the PMIC protocol driver is implemented as a work queue. Scheduling is done by the top half whenever a hardware interrupt is received. The lower half handler does the work that is required to handle the PMIC interrupt. The steps are as follows:

1. Read the current value of the PMIC's interrupt status register to determine the list of currently active events.
2. Clear the PMIC interrupts that will be handled by the PMIC device drivers. Note that if no callback functions have been registered yet for an event, the PMIC protocol driver will just silently ignore the event.
3. Invoke any callback functions that have been registered for the currently active events.

As already noted in the previous section, the interrupt handler must use a spinlock to implement a critical section around any code that accesses the event notification list. This is needed to ensure that the event notification list remains in a consistent state while the interrupt handler is running.

6.5.5 Event Handlers

Event handlers are callback functions that a device driver may use to be notified by the PMIC interrupt handler that a particular event has just occurred. The device driver that is registering a callback function may also specify a single (void *) argument that will be returned later when the callback is invoked. This argument can be used to identify a specific instance of the callback function or be used to access any context-specific data. No return value is expected from the event handler.

6.5.6 Register Access

The PMIC protocol driver exports APIs that allow other device drivers to read and write to PMIC control registers. The PMIC control registers are accessed using one of the two available SPI interfaces. Either the Primary or Secondary SPI interface is used, depending upon the specific design for the hardware connections between the platform and the PMIC. As previously described in Section 6.1.1, "PMIC Register Access and Arbitration," there are significant operational differences between register access by the primary and secondary SPI bus interfaces. However, the PMIC protocol driver is implemented in such a way that all these differences are taken care within the PMIC protocol driver. Externally, the PMIC protocol driver simply provides APIs to read and write to the PMIC control registers.

A separate IOCTL-based interface using the `/dev/pmic` device to read and write to the PMIC control registers has also been implemented as a separate test module. However, this interface is intended only for debugging and testing, and is not intended for general use.

6.6 Driver Source Code Structure

The source files for the PMIC protocol driver are available in the drivers directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/pmic/core`.

Table 6-3 provides a brief description of each of the device driver source files.

Table 6-3. PMIC Protocol Driver Sources File List

File	Description
pmic_core_spi.c	Main function of the module, register access function
pmic_event.c	Event notification function.
pmic_external.c	This files contains client API implementation, define SPI interface.
pmic-dev.c	This provides /dev interface to the user-space programs.
pmic.h	Declaration of all the functions whose implementation differs from PMIC chip to PMIC chip.
mc13783.c	This file contains PMIC specific code (implementation of functions in pmic.h)

Note that in addition to the driver-specific source files, there also exists a `Kconfig` file that is used to define the device driver's build configuration (see Section 6.7, "Driver Configuration") and a `Makefile` that is used during the Linux kernel image build process.

6.7 Driver Configuration

The PMIC protocol driver is configured using the same mechanisms that are provided to configure the Linux kernel image. That is, a `Kconfig` file within the source files directory is used to select whether or not the device driver is to be included in the kernel build process and whether it is to be built as a loadable kernel module or not. Any of the standard kernel configuration tools, such as `menuconfig`, can be used to select and configure the PMIC protocol driver.

The following Linux kernel configuration options are provided for the PMIC protocol driver. In order to enter the configuration screens, use the following command. You should be located in the ltib directory.

`/<ltib dir>/.ltib -c`:

- Device Drivers -> MXC Support Drivers -> MXC PMIC Support -> MXC PMIC device Interface - Choose this to provide /dev interface to PMIC. This makes it possible to have user-space programs use or control PMIC and for notification of PMIC events to user space.
- Device Drivers -> MXC Support Drivers -> MXC PMIC Support -> MC13783 Client Drivers - Used by all MC13783 clients - Used to enable the PMIC client drivers. Some of the MC13783 client drivers that can be selected are:
  - MC13783 ADC support
  - MC13783 Audio support
  - MC13783 Real Time Clock (RTC) support
  - MC13783 Light and Backlight support
  - MC13783 Battery API support
  - MC13783 Connectivity API support
  - MC13783 Power API support

Chapter 7
PMIC Audio Driver

This chapter describes the PMIC audio device driver for Linux. The PMIC audio driver provides low-level control of the PMIC audio playback and recording devices.

The PMIC audio device driver uses the PMIC protocol driver (Chapter 6, “PMIC Protocol Driver”) to control the audio playback and recording components of the PMIC.

7.1 PMIC Audio Driver Features

Figure 7-1 shows the key audio-related components that are provided by the MC13783 Power and Audio Management IC.

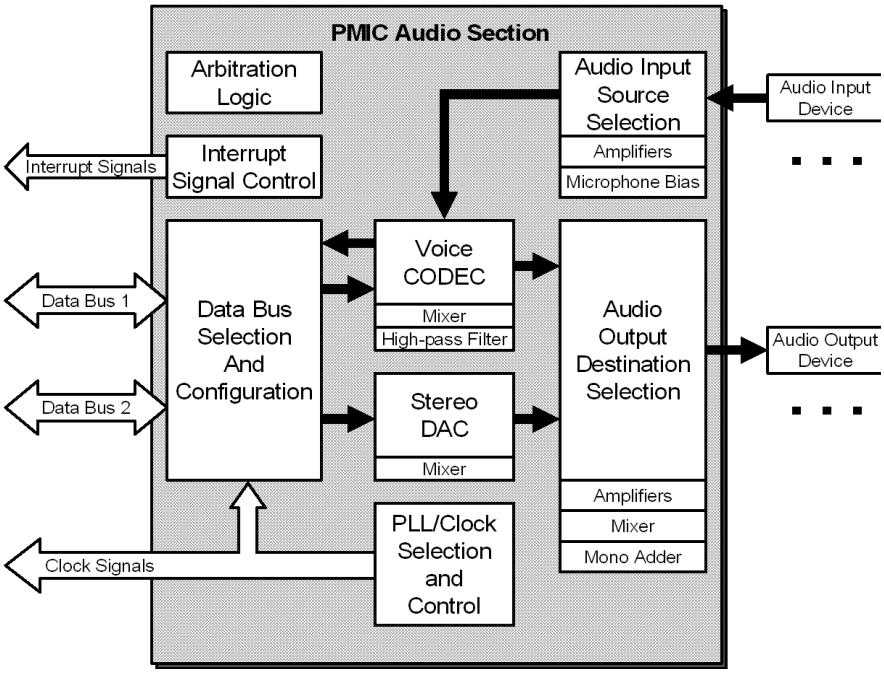


Figure 7-1. PMIC Audio Hardware Components

Even though each specific power management IC may have some unique capabilities and features, they all share the following common components and general capabilities:

- Stereo DAC—Provides both left and right channel audio output with sampling rates from 8 kHz to 96 kHz (in the case of MC13783). The stereo DAC also has an optional internal mixer that can be used to mix together two separate input stereo audio streams to produce a single stereo output stream.

- Voice or telephone codec—Provides both mono playback and recording capabilities with either an 8 kHz or 16 kHz sampling rate. The voice codec on certain power management ICs may also support stereo recording but this feature is platform-specific. The voice codec also includes an optional high-pass filter that can be used to filter the input or output audio streams.
- Data Bus Selection and Configuration section—Controls the connections between the audio data buses and the voice codec and stereo DAC components. The data buses can be configured to operate either in a standard MSB-aligned mode, network mode, or I^2C mode. Each data bus can be routed to either the voice codec or the stereo DAC and both data buses can be active simultaneously along with the voice codec and stereo DAC.
- Audio output control section—Determines which devices or audio output connectors will be used as well as the gain settings on the various output amplifiers. The output control section may also include an additional mixer for mixing together the outputs from the voice codec and the stereo DAC and a mono adder for converting the stereo DAC output to a single mono channel. However, whether the availablity of these components and their exact capabilities are specific to each power management IC.
- Audio input control section—Used to select an appropriate audio recording signal source as well to configure the various input amplifiers and microphone bias circuits. The output of this section is fed directly to the voice codec which then performs the analog-to-digital conversion at either an 8 kHz or 16 kHz sampling rate.
- PLL or clock control section—Internally generates the clock signals to drive the data buses and thereby act as a bus master. Alternatively, the internal clock generator can be disabled and the power management IC operated as a slave device using an external clock source. It is recommended that the power management IC always be configured as the bus master to ensure that the correct clock frequencies needed to support the various audio playback and recording sampling rates are generated. This avoids having to rely on external clock sources that may have to be shared with other system devices and which may not be operating at exactly the correct frequency thereby possibly causing distortion in both audio playback and recording.
- Interrupt signal control section—Determines which hardware interrupts are enabled. The exact number and type of interrupts that may be generated is specific to each power management IC but they may include events, such as the insertion of a microphone or headset.
- Arbitration control block—Determines the level of access to the audio-related hardware registers that is provided to both the primary and secondary processors. Various combinations of read-write or read-only access can be configured as required. However, the audio API does not include any access to the arbitration control block because this component is expected to be properly configured during device power-up and there is currently no operating scenario which would require reconfiguring the arbitration settings while the device is running.

As shown in Figure 7-1, the audio components of the power management IC connect with the processor core and other peripheral devices through the data buses, interrupt signals, and clock signals. However, Figure 7-1 does not show the SPI bus interfaces that are used to access the hardware control registers (including the audio-related registers) on the power management IC. The SPI bus interface and associated APIs are described in more detail in Chapter 6, “PMIC Protocol Driver.”

Finally, the external audio devices, such as headsets, loudspeakers, and microphones are connected to the voice codec or stereo DAC through the appropriate audio jacks and plugs. The exact type and placement

of these jacks and plugs is implementation and device design-dependent. Therefore, while the current implementation does allow access to all of the available audio input and output ports provided by the power management IC, the actual device schematics or design documentation must be used to determine which ports are available and what type of connector is being used. The result of trying to use a port or external audio device which is not available or disconnected is undefined.

The power management ICs all share a similar set of components and features in terms of audio recording and playback functions. However, each power management IC also has its own unique set of features and capabilities beyond what has been described thus far. The documentation for the specific IC should be consulted to fully understand all of the audio-related components, features, and functions provided by a specific power management IC.

The audio API includes the ability to make use of both common and device-specific audio features as required. For example, it is possible to perform mono audio recording using the voice codec on the MC13783 power management IC. However, the API also provides stereo recording through the MC13783 voice codec, as that is supported through the MC13783 power management IC.

7.2 Driver Requirements

The PMIC audio driver provides full access to all of the features that are supported by the PMIC hardware. The API must be identical for all ICs. Attempting to use a feature or select a configuration option that is not supported by the PMIC that is being used returns PMIC_NOT_SUPPORTED. Successful operations always return PMIC_SUCCESS, while any supported operations that failed due to an error condition return PMIC_ERROR.

7.2.1 Audio Device Handle Management

The PMIC audio device driver must provide an API to support the following operations:

- Obtain a device handle for accessing the stereo DAC, voice codec, or external stereo input.
- Release a previously acquired device handle.

Higher-level device drivers that wish to access the PMIC audio components must first request and receive a valid device handle. This ensures that there will never be a conflict over access to and control of a particular audio component. Separate device handles have been defined for the stereo DAC, the voice codec, and the external stereo input.

7.2.2 Digital Audio Bus Selection and Configuration

After successfully acquiring the appropriate device handle, another set of APIs must be provided to allow for the selection and configuration of the digital audio bus:

- Select the digital audio data bus to be used.
- Configure the operating mode, timeslot selection, and timing signal parameters for the selected digital audio data bus.

Note that both the voice codec and the stereo DAC can be connected to either of the two available digital audio buses but only to one bus at a time. Furthermore, a single digital audio bus cannot be simultaneously connected to both the voice codec and the stereo DAC.

The digital audio bus must be able to operate in either master or slave modes at all of the permissible sampling rates.

7.2.3 Stereo DAC and Voice Codec Control and Configuration

An API interface must be provided for directly controlling the voice codec and the stereo DAC audio components. The required functionality includes the following:

- Enable/disable the audio device
- Enable/disable the available hardware mixing devices
- Perform a digital filter reset

7.2.4 Audio Input Section Control and Configuration

An API must be provided to configure the PMIC's audio input section to support using the voice codec to record an audio stream. The required functionality includes the following:

- Select the desired audio input source and recording mode (stereo or mono)
- Enable/disable the audio input source
- Select the desired input amplifier gain level
- Enable/disable the appropriate microphone bias circuit

7.2.5 Audio Output Section Control and Configuration

An API must be provided to configure the PMIC's audio output section to support playback using either the voice codec or the stereo DAC. The required functionality includes the following:

- Select the desired audio source for output (for example, voice codec, stereo DAC, or external stereo input)
- Select the desired output amplifier gain and balance levels
- Enable/disable the available hardware mixing devices
- Enable/disable the phantom ground circuit

7.2.6 Resetting the PMIC Audio Components

An API must be provided to allow partial or complete resetting of the PMIC audio components. This will provide a means to ensure that audio components are in a consistent state and to recover from any errors that might occur. The required functionality includes the following:

- Reset only the voice codec or stereo DAC settings to their respective power-on settings
- Reset all PMIC audio-related settings in all registers to their respective power-on settings

7.2.7 Audio-Related Interrupts and Event Notification

An API must be provided to allow other device drivers (for example, the OSS sound driver) to register for and to receive notification of audio-related events. The PMIC audio driver will first receive and handle all audio-related interrupts as required but it must also allow higher-level drivers and applications access to the event notification and any associated data so that they too can respond as required. The required functionality includes the following:

- Register an event callback function
- Deregister an event callback function
- Enable/disable headset detection
- Enable/disable microphone bias detection notification (MC13783 PMIC only)

7.2.8 Additional Audio-related Configuration Options

An API must be provided to support some additional audio driver-related functions that do not necessarily fit into any of the functional categories that have already been given. The required functionality includes the following:

- Ability to query for which PMIC chip and driver is currently being used
- Ability to control the power consumption of the audio components by completely or selectively powering up and powering down specific audio circuits and devices
- Enable/disable the anti-pop circuitry
- Provide a fully decoded PMIC audio control register dump (for debugging/testing purposes only) driver

7.3 Software Operation

The PMIC audio driver makes calls to the PMIC protocol driver to reconfigure the PMIC's control registers to the desired setting. All higher-level audio configuration and operation requests are converted to the appropriate PMIC control register settings and then the PMIC's hardware state is updated through the SPI bus interface.

7.4 Driver Architecture

Figure 7-2 shows the basic architecture of the PMIC audio driver and the interfaces to the higher-level Linux OSS sound driver as well as the underlying PMIC hardware.

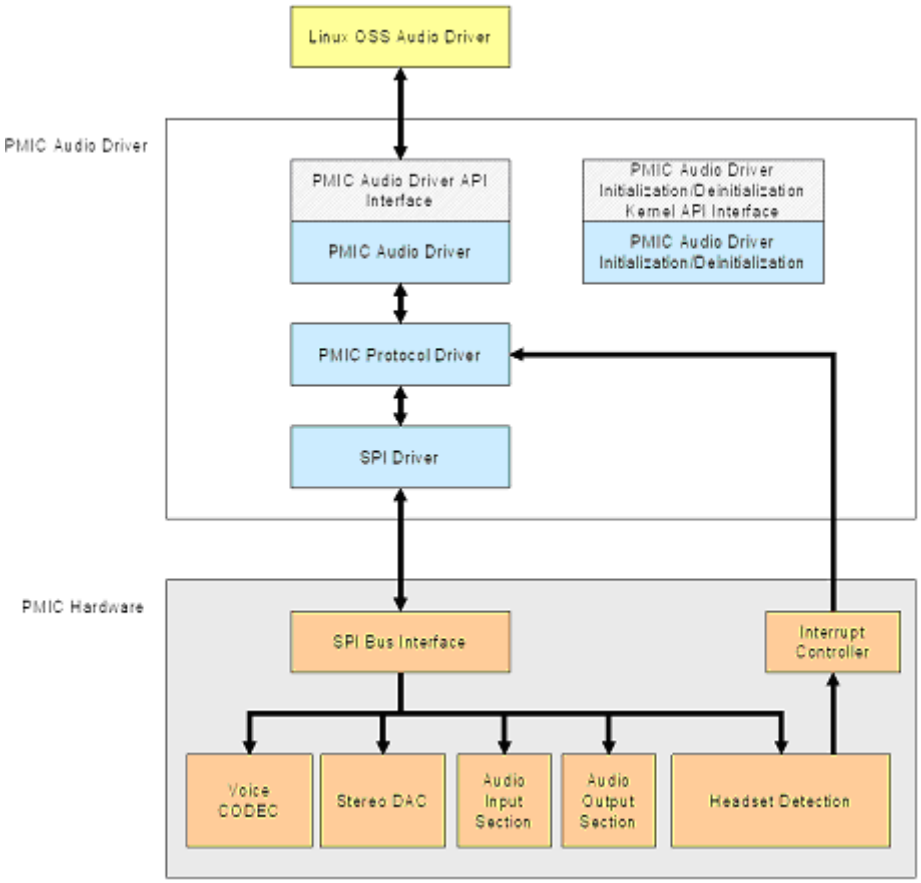


Figure 7-2. PMIC Audio Driver Architecture

7.5 Driver Implementation Details

A structure defines the fields within each of the PMIC's audio-related control registers. Each element of the structure defines the size and offset of the register field. This enables the use of simple macros to access each register field.

Note that the PMIC's hardware registers are not exported outside the device driver. There is no need to provide external low-level access to the PMIC's registers. This also helps to ensure the maintenance of complete control over the PMIC hardware state.

Another structure keeps track of the current PMIC hardware state. This data structure always mirrors exactly how the hardware has been configured, and avoids the possibility of conflicting or invalid configurations. It is also possible to easily return the current state of the PMIC audio hardware without using extra SPI bus transactions to directly query the hardware.

7.5.1 Driver Initialization

Nothing special needs to be done during the initialization phase for this device driver. The higher-level OSS sound driver makes calls to this driver only through the exported API and no other access method needs to be supported.

7.5.2 Driver Deinitialization

When deinitializing this driver, make sure that any still-opened device handles are properly closed and that the PMIC hardware is restored to the default power on state. This will help to ensure that the PMIC is never left in an inconsistent state and that it will never signal an interrupt event when there is nothing registered to properly handle it.

7.6 Driver Source Code Structure

Table 7-1 lists the MC13783-specific source files that are available in the device driver directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/pmic/mc13783/`.

Table 7-1. MC13783 Audio Driver Source Files

File	Description
`pmic_audio.c`	Implementation of the MC13783 audio PMIC client driver.
`pmic_audio.h`	Header file for the MC13783 audio client driver.

The header file for PMIC audio drivers is `<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/pmic_audio.h`.

7.7 Driver Configuration

This module can be selected using the ltib menu options.

To get to the PMIC ADC 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 PMIC ADC driver.

- Device Drivers > MXC Support Drivers > MXC PMIC Support > MC13783 Audio support
  This is the configuration option to choose the MC13783-specific audio driver.

Chapter 8
PMIC Digitizer Driver

This chapter describes the PMIC digitizer driver for Linux that provides low-level access to the PMIC's analog-to-digital converters (ADC).

The PMIC digitizer driver controls the analog-to-digital converter (ADC) components of the PMIC. This capability includes taking measurements of the X-Y coordinates and contact pressure from an attached touchpanel. This device driver uses the PMIC protocol driver (see Chapter 6, "PMIC Protocol Driver") to access the PMIC hardware control registers that are associated with the ADC.

8.1 PMIC Digitizer Driver Features and Capabilities

The PMIC digitizer driver is used to provide access to and control 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 touchpanel 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.

Note that 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.

8.2 Driver Requirements

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

The following are the requirements for supporting a touchpanel device:

- Must be able to select either a single ADC input channel or an entire group of input channels to be converted.
- Must be able to specify high and low level thresholds for each ADC conversion.
- Must be able to start an ADC conversion by issuing the appropriate start conversion command.
- Must be able to start an ADC conversion immediately following the rising edge of the ADTRIG input line or after a predefined delay following the rising edge.
- Must be able to enable/disable hardware interrupts for all ADC-related event notifications.
- Provide 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 must be able to register/deregister additional callback functions to provide custom handling of all ADC-related event notifications.
- Provide a read-only device interface for passing touchpanel (X,Y) coordinates and pressure measurements to applications.
- Provide the ability to read out one or more ADC conversion results.
- Implement the appropriate input scaling equations so that the ADC results are correct.
- Must be able to specify the delay between successive ADC conversion operations, if supported by the PMIC. For PMIC chips that do not support this feature, the device driver should return a NOT_SUPPORTED status.
- Provide support for a pulse generator that is synchronized with the ADC conversion. For PMIC chips that do not support this feature, return a NOT_SUPPORTED status.
- Provide a complete IOCTL interface to initiate an ADC conversion operation and to return the conversion results.
- Provide support for a polling method to detect when the ADC conversion has been completed.

Note that 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, as previously indicated, this device driver is not responsible for SPI bus arbitration configuration. The appropriate arbitration settings that are required for this device driver to work properly are expected to have been set during the system boot process.

8.3 Driver 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 using the APIs that are provided with the PMIC protocol driver.

Once an ADC conversion has been started, the calling thread should be suspended until the conversion has been completed. A busy loop should be avoided as this will negatively impact 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.

8.4 Driver Architecture

Figure 8-1 shows the basic architecture for the PMIC digitizer driver. The PMIC protocol driver and the platform's SPI driver provides the necessary interface to read and write the PMIC's hardware control registers.

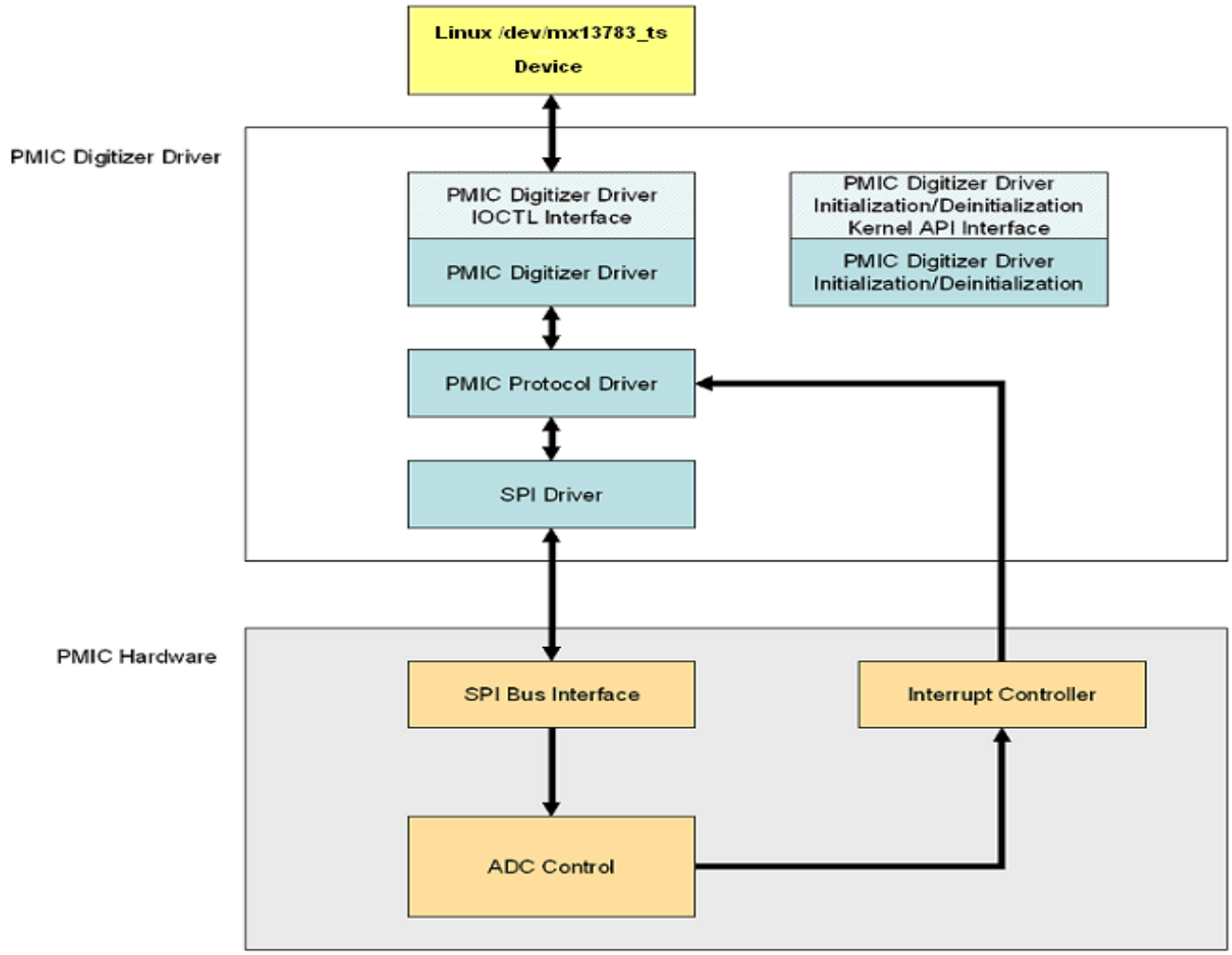


Figure 8-1. PMIC Digitizer Driver Architecture

The PMIC's interrupt controller generates interrupts for the following events:

- ADC end-of-conversion
- High/low level threshold exceeded

8.5 Driver Implementation Details

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 signaled by a hardware interrupt.

8.5.1 Driver Initialization

NOTE

This section does not apply to the i.MX31 3-Stack Board.

The PMIC digitizer driver must also create the appropriate `/dev` character device entry to allow applications to obtain the touchpanel (X,Y) coordinates and pressure measurements. The touchpanel device is only required to support a read operation.

The MC13783— `/dev/mc13783_ts` device is created.

A device-independent softlink, `/dev/ts`, which references the PMIC-specific touchpanel device name is also created, but this is part of a Linux boot script and is not handled by this device driver.

8.5.2 Driver Removal

The PMIC digitizer driver must remove the `/dev` device entry that was created when the driver was loaded.

8.6 Driver Source Code Structure

Table 8-1 lists the source files for the MC13783-specific version of this driver. These are available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/pmic/mc13783`.

Table 8-1. MC13783 Digitizer Driver Source Files

File	Description
`pmic_adc.c`	Implementation of the MC13783 ADC client driver.
`pmic_adc_defs.h`	Hardware definitions and internal functions for the ADC client driver.

The header file for PMIC adc drivers is
`<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/pmic_adc.h`.

8.7 Linux Menu Configuration Options

The following Linux kernel configuration is provided for this module. In order to get to the PMIC ADC 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 -> MXC Support Drivers -> MXC PMIC Support -> MC13783 Client Drivers -> MC13783 ADC support

  Chooses the MC13783 (MC13783) – Specific digitizer driver.

Chapter 9
PMIC Power Management Driver

The PMIC power management device driver for Linux provides enabling and disabling of various low-power modes. The MC13783 regulator driver provides the low-level control of the power supply regulators and selection of voltage levels.

This driver has been deprecated. It has been replaced by the MC13783 regulator driver. It is still used internally by some other drivers but once they have been changed to use the regulator driver, this driver will be removed.

This device driver makes use of the PMIC protocol driver (see Chapter 7, “PMIC Protocol Driver”) to access the PMIC hardware control registers.

9.1 PMIC Features

Using the PMIC chip in a product potentially provides a complete power control and power management strategy. The PMIC chips have built-in switching power supplies and linear voltage regulators that can be configured to power the rest of the platform. These power supplies may also be selectively enabled/disabled, and the voltage levels may be dynamically adjusted to control power consumption.

In addition, there is an internal state machine that can provide automatic power-cut functions and transparent transitions between various low-power operating modes. Full shutdown and automatic restart based on possible external events is also supported.

The IC documentation should be consulted for full details about what power supplies are provided, how they can be configured, and how the internal power control logic is implemented.

9.2 Driver Requirements

The MC13783 PMIC regulator driver is a client of the PMIC protocol driver and regulator core driver. It provides services for regulator control of the PMIC component.

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

9.3 Driver Software Operation

The PMIC power management driver and the MC13783 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's 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.

9.4 Driver Architecture

Figure 9-1 shows the basic architecture of the MC13783 regulator driver. Figure 9-2 shows the basic architecture of the PMIC power management driver, as well as its higher-level interfaces and the connections to the underlying PMIC hardware.

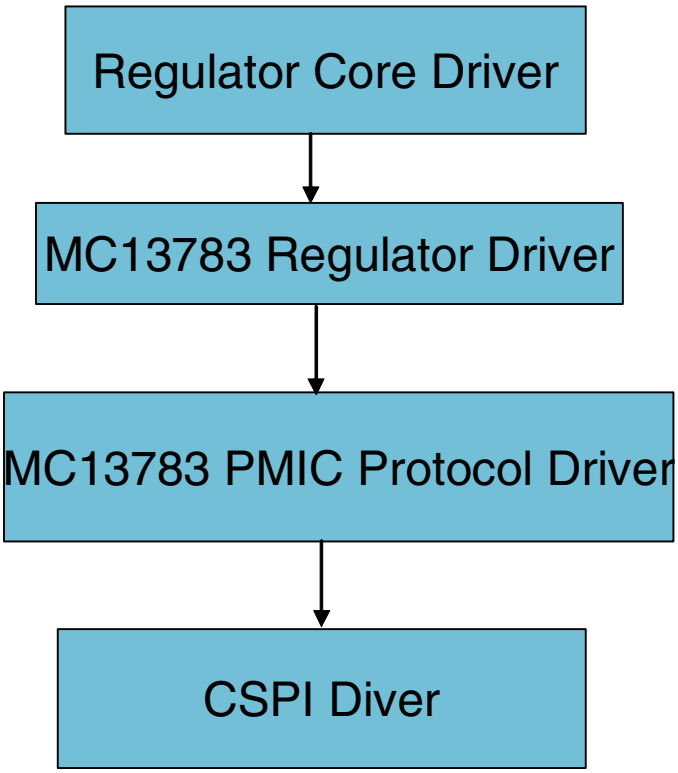


Figure 9-1. MC13783 Regulator Driver Architecture

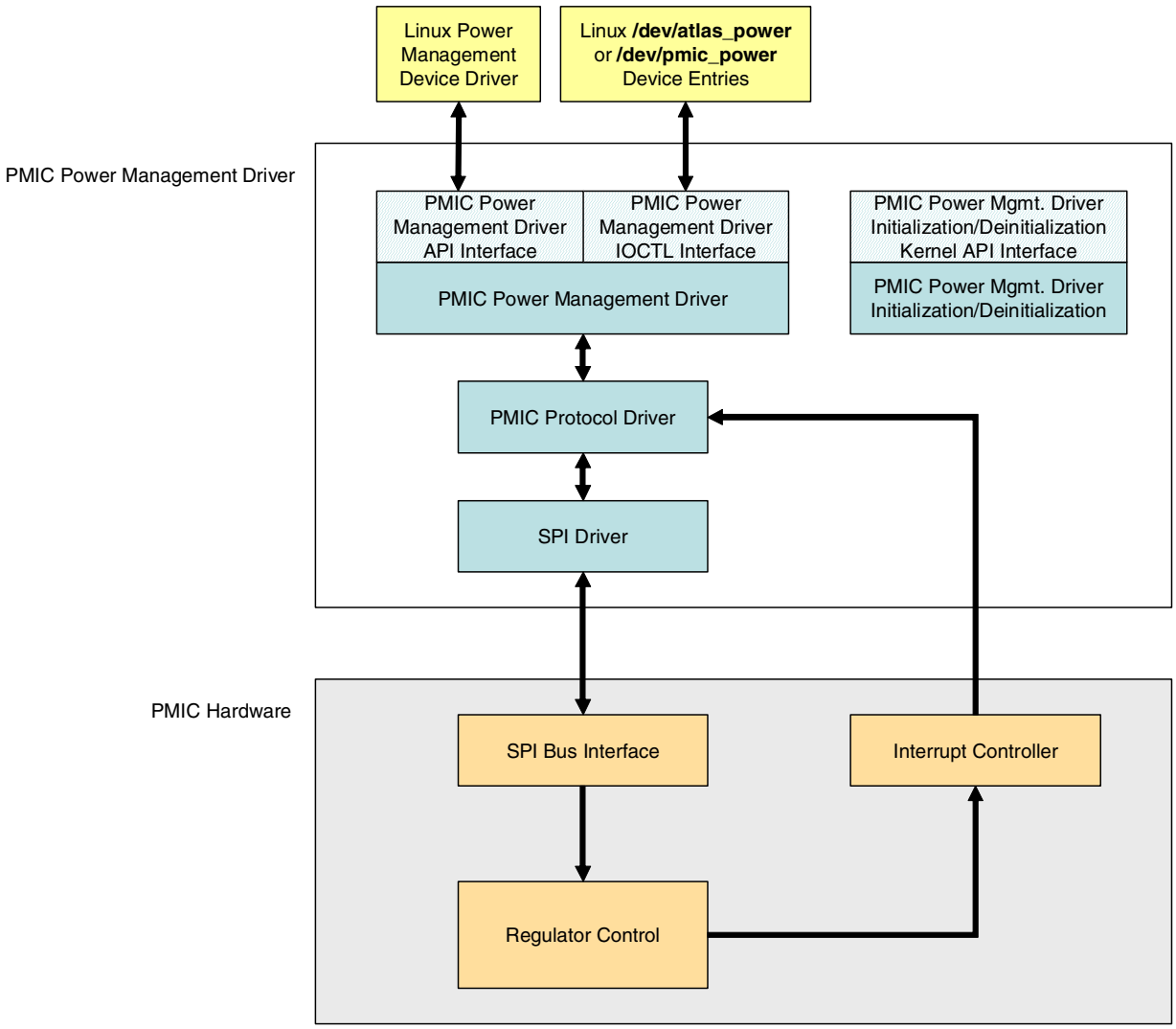


Figure 9-2. PMIC Power Management Driver Architecture

9.5 Driver Implementation Details

The access to the PMIC power management driver and the MC13783 regulator are provided through a set of exported APIs. The exported APIs are meant for use by other kernel-mode device drivers. The IOCTL interface is not provided for general use. An example of how it should work can be found in the unit test module. All IOCTL calls are translated into corresponding internal API calls to perform the requested operation.

All of the power management 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's hardware registers.

9.6 Driver Source Code Structure

The MC13783-specific source files for the power management driver and MC13783 regulator driver are available in the device driver directories:
`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/pmic/mc13783` and
`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/regulator/mc13783`.

Table 9-1. MC13783 Power Management Driver Source Files

File	Description
reg-mc13783.c	Implementation of the MC13783 regulator client driver
pmic_power.c	Implementation of the MC13783 power management client driver
pmic_power_defs.h	Internal header for MC13783 power management client driver.

The header file for PMIC Power drivers is
`<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/pmic_power.h`.

9.7 Driver Configuration

This module is selected using the ltib menu configuration options.

In order to get to the PMICpower configuration, use the command `./ltib -c` when located in the `<ltib dir>`. On the screen that is displayed select **Configure the Kernel** then exit. When the next screen appears select the following options to use the MC13783 regulator and power management drivers.

- Device Drivers > Voltage and Current regulator >MC13783 Regulator Support
- Device Drivers > MXC Support Drivers >MXC PMIC Support > MC13783 Power API support

Then exit and ltib will build the kernel with these drivers enabled.

Chapter 10
PMIC Connectivity Driver

The MC13783 PMIC connectivity driver for Linux provides support for external connectivity of the following types:

- RS-232
- USB OTG
- CEA936

The MC13783 PMIC Connectivity Driver is based on the MC13783 DTS 3.0 specification and the Freescale MC13783 board.

This device driver makes use of the PMIC protocol driver (see Chapter 6, “PMIC Protocol Driver”) to access the PMIC hardware control registers.

10.1 PMIC Features

The PMIC includes transceivers to support both RS-232 and USB On-the-Go (OTG) external connectivity. The MC13783 PMIC also includes support for the CEA936 specification. Due to the limited number of available pin connections, only one of these external connectivity modes can be used at any one time. In the case of the USB OTG transceiver, the specific connections that are made between the PMIC transceiver and the host platform may also affect which USB OTG operating modes are supported.

The PMIC documentation should also be consulted for details about the configuration and use of the RS-232 and USB OTG transceivers.

10.2 Driver Requirements

The PMIC connectivity driver is a client of the PMIC protocol driver and uses it to provide access to the PMIC’s hardware control registers. The PMIC connectivity driver, in turn, must provide a suitable API interface for the Linux UART and USB OTG drivers to support both RS-232 and USB OTG connectivity. The required functionality includes the following:

- Acquisition and release of a connectivity device handle—The current owner of the device handle is granted exclusive access to the PMIC’s transceivers as long as the handle is being held. Any attempts to access or use the connectivity hardware without first successfully acquiring the device handle result in the return of PMIC_ERROR.
- Selection of one of the supported operating modes—For example, RS-232, USB OTG, or CEA-936. Attempting to use an unsupported mode results in a NOT_SUPPORTED return.

NOTE

The list of supported modes may differ from PMIC to PMIC.

- Registration and removal of an event handler callback function—Any attempts to register a callback for an unsupported event results in a NOT_SUPPORTED return.
- Invocation of all registered callback functions—When the matching event has been signaled by the PMIC protocol driver.
- Configuration of the PMIC USB transceiver—As required to communicate with the platform's USB controller. This includes, for example, configuring the operating speed and the transceiver's power supply.
- Configuration of the PMIC USB transceiver—As required to support the additional USB OTG requirements. This includes, for example, setting the Data Line Pulse duration and performing a Host Negotiation Protocol sequence.
- Configuration of the PMIC RS-232 transceiver—As required to support an RS-232 connection.
- Configure the PMIC hardware to support the CEA-936 operating mode— Attempting to use the CEA-936 mode when it is not supported by the underlying PMIC hardware results in a NOT_SUPPORTED return.

NOTE

Currently, this mode is only supported by the MC13783 PMIC.

- Ability to explicitly reset the PMIC's connectivity-related hardware components to their default or power-on state—This function is useful to reinitialize the connectivity hardware to a known state.
- Automatic RS-232 to USB OTG mode switch—As supported by the PMIC, whenever a USB device is attached while idle in RS-232 mode.

The specific hardware register settings that are required to configure the PMIC connectivity components are located in the documentation for each PMIC chip.

10.3 Driver Software Operation

Most of the operations that must be performed by the PMIC connectivity driver involve setting the appropriate values in the PMIC hardware control registers using the APIs that are provided by the PMIC protocol driver. Specific settings for each PMIC can be located within the documentation for each PMIC chip.

Event handler callback functions, if any, are registered using PMIC protocol driver APIs. The PMIC protocol driver interrupt handler automatically invokes all registered callback functions whenever the associated event is signaled by the PMIC hardware.

NOTE

This device driver is not responsible for handling the actual RS-232 or USB OTG data transfer operations.

Higher-level UART or USB OTG drivers handle the transfers, as well as the configuration of the UART and USB OTG controllers. The PMIC chips only provide the transceiver components, an event notification capability, and the connections to any external connectors. The PMIC connectivity driver's role is restricted to transceiver configuration and event notification.

10.4 Driver Architecture

Figure 10-1 shows the basic architecture of the PMIC connectivity driver, as well as its relationship to the Linux UART and USB OTG drivers and the PMIC hardware components. The UART driver configures the RS-232 transceiver, while the USB OTG driver handles the USB OTG transceiver. Only one of these transceivers can be active at any one time.

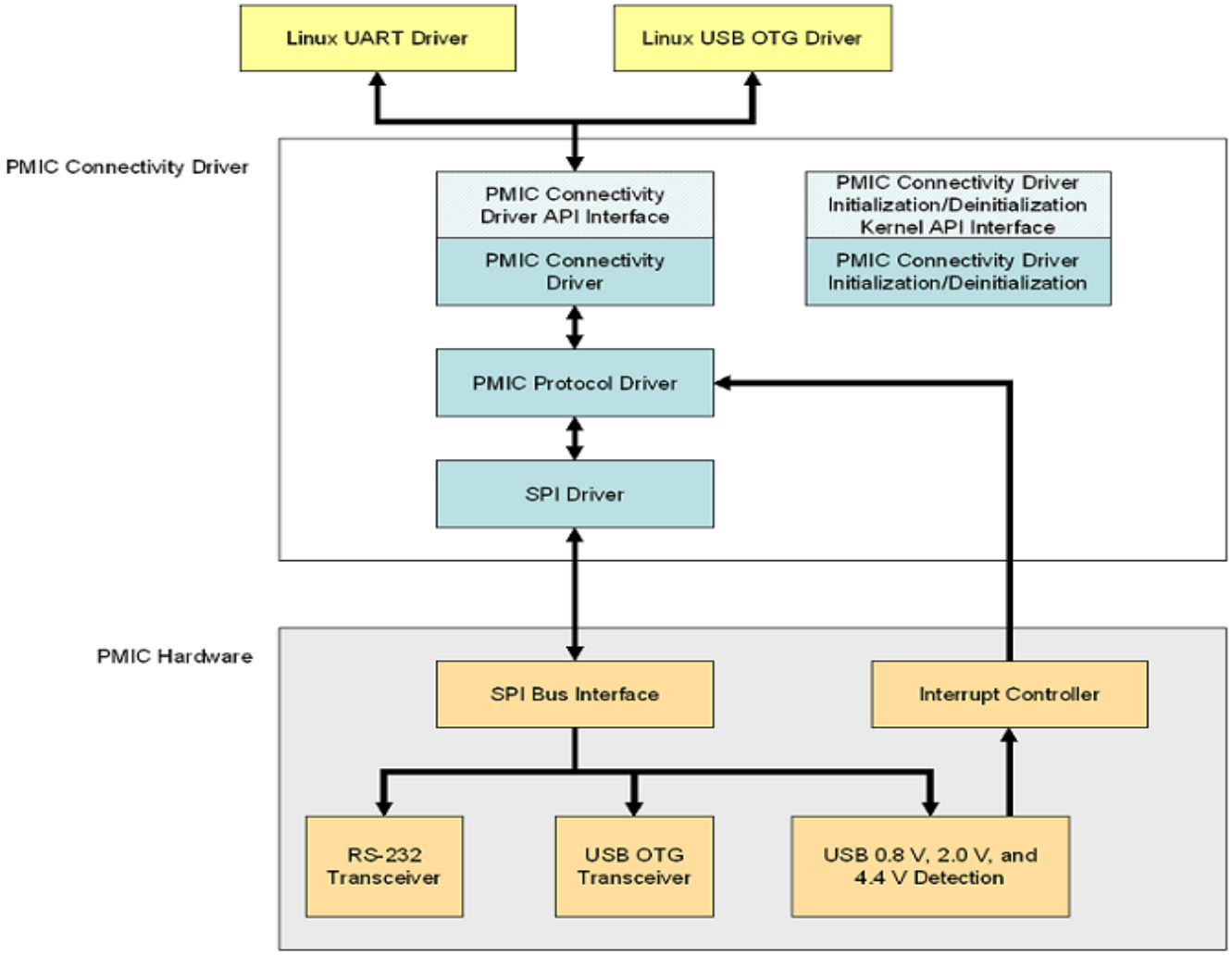


Figure 10-1. PMIC Connectivity Driver Architecture

The only events that are typically reported to the PMIC connectivity driver involve the detection of USB-related state changes. In particular, the voltage level that is measured on the USB signal lines is used to indicate the insertion or removal of a device, the requested operating speed, and whether it operates as a host or a peripheral (for USB OTG devices). Details about the signaling and correct handling of the various USB-related events can be located in the USB specification and the USB OTG supplement.

10.5 Driver Implementation Details

The PMIC connectivity device driver uses a single data structure to track the current state of the device handle and all available PMIC configuration options. All APIs that modify the PMIC hardware also update the data structure, so the device driver state always matches that of the hardware.

A mutex is used to ensure that the state of the device driver and the PMIC hardware are always kept in a consistent state. A mutex is used whenever the system is not operating in an atomic or interrupt context within this driver. In the limited places where the system is operating in an atomic or interrupt context, a spinlock is also acquired. The spinlock is released at the earliest possible time to minimize interrupt handling latencies.

A task is used to perform most of the event handling operations, so as to minimize the time actually spent in the low-level interrupt handling routine.

10.5.1 Driver Initialization

The device driver initialization sequence continues as normal with no special provision for the PMIC connectivity device driver.

NOTE

No device name is created within the `/dev` directory, because this driver does not support any IOCTL interfaces.

10.5.2 Driver Removal

If the device handle is still being held when this driver is removed, then the handle must be forcibly closed and the PMIC connectivity components must be restored to default power-on state, before the deinitialization sequence is completed.

10.6 Driver Source Code Structure

The MC13783-specific source file, `pmic_convity.c`, for this device driver, is available in the device driver directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/pmic/mc13783`. The source file describes the function for MC13783 USB/RS232 connectivity client. The header file for PMIC connectivity drivers is `<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/pmic_convity.h`.

10.7 Driver Configuration

This module can be selected using the ltib menu options.

To get to the PMIC Connectivity 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 PMIC Connectivity driver.

- Device Drivers -> MXC Support Drivers -> MXC PMIC Support -> MC13783 Client Drivers -> MC13783 Connectivity API Support

  Chooses the MC13783–specific connectivity driver.

Chapter 11
PMIC Battery Driver

The PMIC battery device driver for Linux provides support for controlling the PMIC battery interface circuits. This device driver makes use of the PMIC protocol driver (see Chapter 6, "PMIC Protocol Driver") to access the PMIC hardware control registers.

11.1 PMIC Features

PMIC chips include circuits to automatically detect the presence of a charger and to recharge the system battery. Additional circuits are provided to detect and prevent overcharging. Additional capabilities include:

- Support for USB chargers
- Support for a coin cell charger

Battery voltage levels can also be monitored using the analog-to-digital converter and low voltage or battery end-of-life conditions can be signaled through the use of hardware interrupts.

11.2 Driver Requirements

This module is a client of the PMIC protocol driver and uses it to provide access to the PMIC's hardware control registers. The PMIC battery driver, in turn, must provide an IOCTL interface that applications can use to control and monitor the state of the battery and charger circuits. The required functionality includes the following:

- API for battery charger control including selecting the appropriate charger path
- Configure the charging mode (for example, the charge current level)
- Configure the battery voltage and current level monitoring and end-of-life functions.

The specific hardware register settings that are required to configure the battery control circuits can be located in the documentation for each PMIC chip.

11.3 Driver Software Operation

The PMIC battery driver provides an IOCTL interface through the `/dev/pmic_battery` device. Applications use this driver to access the PMIC battery control registers and circuits. The battery driver actually uses the PMIC protocol driver's APIs to perform the necessary hardware control register read/write operations.

The PMIC protocol driver's APIs are also used to register/deregister event handler callback functions. Event handlers can be registered for any of the supported battery-related event notifications, for example, a battery end-of-life condition, detection of a charger being attached, or a charger over voltage condition.

11.4 Driver Architecture

Figure 11-1 shows the basic architecture of the PMIC battery device driver along with the associated PMIC hardware components.

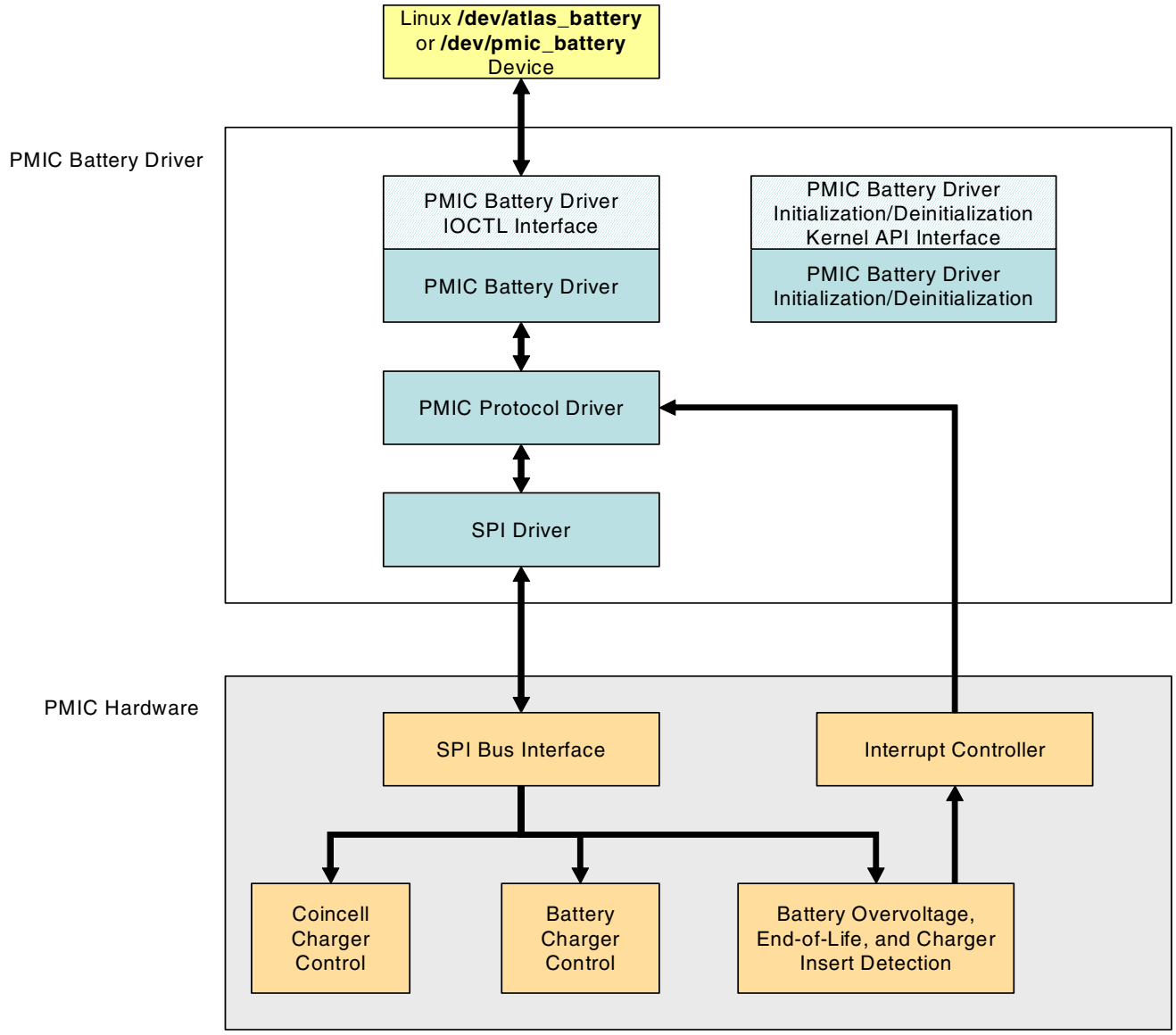


Figure 11-1. PMIC Battery Device Driver Architecture

11.5 Driver Implementation Details

The implementation of the PMIC battery driver is relatively straightforward and involves providing the appropriate IOCTL interface to support the `/dev/pmic_battery` device. Internally, each IOCTL call is translated to the appropriate PMIC hardware control register operations, which are then performed with the aid of the PMIC protocol and SPI drivers.

Event handler callback functions are registered directly with the PMIC protocol driver. The registered event handler is invoked when the corresponding event is detected and the hardware interrupt is received by the PMIC protocol driver.

11.5.1 Driver Initialization

During initialization, register a `/dev/pmic_battery` device to allow application-level access to the device driver using the IOCTL interface.

11.5.2 Driver Deinitialization

The previously registered `/dev/pmic_battery` device entry must be removed when the device driver is unloaded.

11.6 Driver Source Code Structure

Table 11-1 lists the source files for this driver that are available in the directory, `/<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/pmic/mc13783`.

Table 11-1. MC13783 Battery Driver Source Files

File	Description
`pmic_battery.c`	Implementation of the PMIC battery client driver.
`pmic_battery_defs.h`	Define hardware registers for the PMIC battery driver.

The header file for PMIC battery drivers is `/<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/pmic_battery.h`.

11.7 Driver Configuration

This module can be selected using the ltib menu options.

To get to the PMIC Battery device driver, use the command `./ltib -c` when located in the `<ltib dir>`. In the screen displayed, select **Configure Kernel** and exit. When the next screen appears, select the following options to enable the PMIC Battery device driver.

From the menu option, **Device Drivers** > **MXC Support Drivers**>**MXC PMIC Support** in **MC13783 Client Drivers** select **MC13783 Battery API support,** and choose the MC13783-specific version of the battery device driver.

Chapter 12
PMIC Light Driver

The MC13783 PMIC Light Driver for Linux provides access to the PMIC's backlight and LED control circuits. This device driver makes use of the PMIC protocol driver (see Chapter 6, "PMIC Protocol Driver") to access the PMIC hardware control registers.

12.1 PMIC Features

The PMIC chip includes circuits to control the following external components:

- Backlight (for LCD or keypads)
- Color LEDs

The current level and duty cycle can be controlled as required to satisfy a wide variety of operating requirements. The color LEDs can also be configured to flash in a number of different patterns. Complete information about the backlight and LED controls are located in the documentation for each PMIC.

12.2 Driver Requirements

The PMIC light driver provides access to all of the PMIC backlight and LED control circuits. This includes configuring the current levels, duty cycle, and flashing modes. Note that the actual external devices that are attached to the PMIC differ from platform to platform. Therefore, while the light driver must provide access to all of the PMIC supported features, it cannot make any assumptions about the actual nature of the external devices (for example, the color of the LEDs that are attached) and whether they actually exist or not.

Note that the PMIC light driver interface may include functions that are not supported by all PMIC chips. Attempting to use a configuration that is not supported by the current PMIC hardware returns NOT_SUPPORTED.

12.2.1 Backlight Control Functions

The PMIC backlight circuits are intended to support the control of the backlight level for an LCD display and/or the keypad. The device driver supports the following operations:

- Enable/disable the backlight
- Set/get the backlight current level
- Set/get the backlight duty cycle
- Set/get the backlight cycle time
- Configure the backlight ramp up and ramp down settings
- Configure the backlight strobe settings

12.2.2 LED Control Functions

The LED control circuits supplement the backlight circuits by providing the ability to control additional light sources for signaling purposes and for other special effects. The LED channels are labeled as being R, G, and B because one typical application would be to attach red, green, and blue LEDs, respectively, to each channel. However, this is not a required configuration, and other types of LEDs may be used with these circuits. The device driver supports the following operations:

- Enable/disable each individual colored LED circuit
- Select either colored LED or funlight operating modes
- Set/get the colored LED current level
- Set/get the colored LED blink pattern
- Set/get the funlight current level
- Set/get the funlight duty cycle
- Set/get the funlight cycle time
- Configure the funlight ramp settings
- Configure the funlight strobe settings
- Enable/disable audio modulation

12.3 Driver Software Operation

The operation of the PMIC light driver is fairly simple, and only involves configuring the PMIC hardware control registers as required. Access to the PMIC hardware control registers uses the PMIC protocol driver, which in turn, uses the SPI driver.

As no standard Linux device driver exists to control backlight and external LEDs, applications must use the documented IOCTL interface to access the PMIC light driver. Note, however, that not all of the available backlight and LED control functions are supported by a specific PMIC chip. The device driver returns NOT_SUPPORTED if an attempt is made to use a configuration or function that is not supported by the underlying PMIC hardware.

The PMIC-specific control register settings that are required to configure the various backlight and LED control circuits are located in the documentation for each PMIC.

NOTE

No interrupt or notification events are associated with the PMIC light driver.

12.4 Driver Architecture

Figure 12-1 shows the basic PMIC light driver architecture along with the PMIC hardware components that are being used.

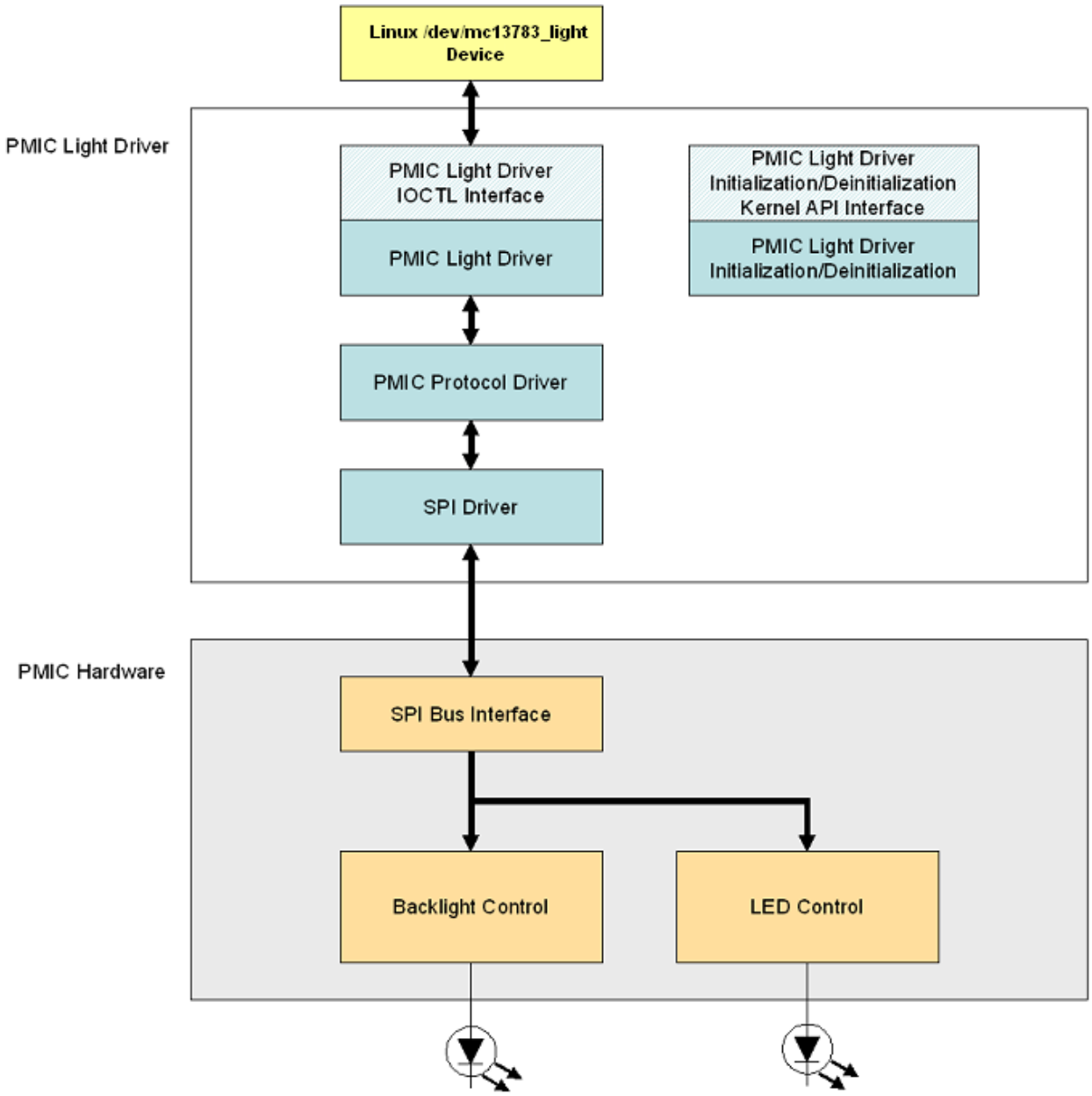


Figure 12-1. PMIC Light Driver Architecture

12.5 Driver Implementation Details

Configuring the PMIC light driver includes configuring parameters, such as duty cycle, current level, ramp-up/ramp-down profiles, and so on, for the various backlight and LED circuits. The appropriate control register settings are located in the documentation for the PMIC chip.

12.5.1 Driver Initialization

To initialize this driver, open the `/dev/pmic_light` device to allow application-level access to the device driver using the IOCTL interface.

12.5.2 Driver Deinitialization

When the device driver is unloaded, remove the `/dev/pmic_light` device.

12.6 Driver Source Code Structure

Table 12-1 lists the source files for the MC13783-specific version of this driver that are available in the device driver directory, `/<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/pmic/mc13783`.

Table 12-1. MC13783 Light Driver Source Files

File	Description
`pmic_light.c`	Implementation of the MC13783 light client driver.
`pmic_light_defs.h`	Definitions for the MC13783 light client driver.

The header file for PMIC Light drivers is as follows:
`/<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/pmic_light.h`.

12.7 Driver Configuration

This module can be selected using the ltib menu options.

To get to the PMIC Light 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 PMIC Light driver.

- Device Drivers > MXC Support Drivers > MXC PMIC Support > MC13783 Client Drivers > MC13783 Light and Backlight support

  Choose the MC13783–specific version of the light driver.

Chapter 13
PMIC Real Time Clock (RTC)

The PMIC RTC for Linux provides access to the PMIC's RTC control circuits. This device driver makes use of the PMIC protocol driver (see Chapter 6, "PMIC Protocol Driver") to access the PMIC hardware control registers.

13.1 PMIC Features

The PMIC chip is used for the following functions:

- Real-time clock control
- Wait alarm event

13.2 Driver Requirements

The PMIC RTC driver is a client of the PMIC protocol driver. It provides services for real time clock control of PMIC component.

13.3 Driver Software Operation

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

13.4 Driver Architecture

Figure 24-1 shows the basic PMIC RTC driver architecture along with the PMIC hardware components that are being used.

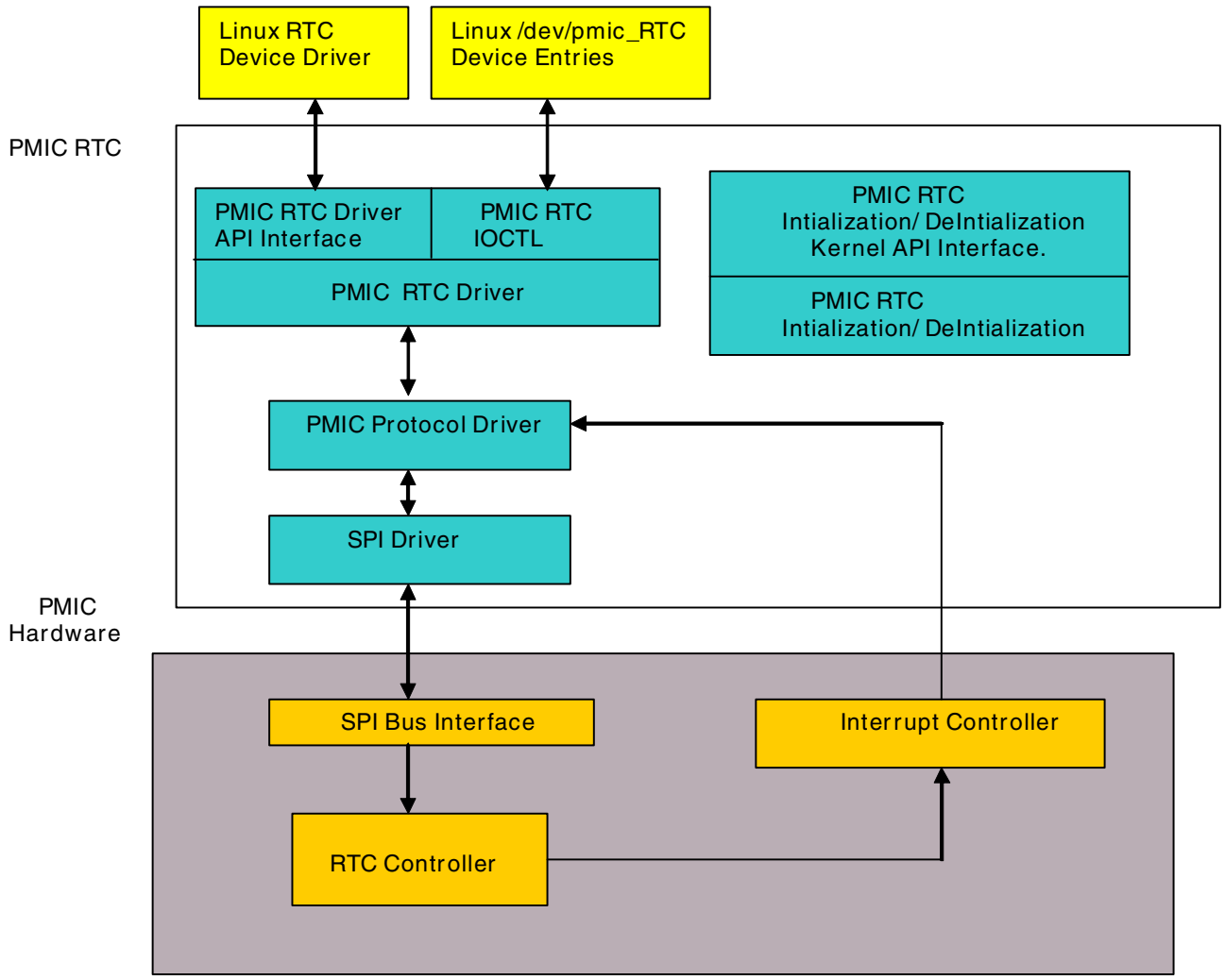


Figure 13-1. PMIC RTC Driver Architecture

13.5 Driver Implementation Details

The device driver supports the following operations:

- 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

13.5.1 Driver Initialization

To initialize this driver, open the `/dev/pmic_rtc` device to allow application-level access to the device driver using the IOCTL interface.

13.5.2 Driver Deinitialization

When the device driver is unloaded, remove the `/dev/pmic_rtc` device.

13.6 Driver Source Code Structure

Table 13-1 lists the source files for the MC13783-specific version of this driver that are available in the device driver directory, `/<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/pmic/mc13783`.

Table 13-1. MC13783 RTC Driver Source Files

File	Description
`pmic_rtc.c`	Implementation of the MC13783 RTC client driver.
`pmic_rtc_defs.h`	Definitions for the MC13783 RTC client driver.

The header file for PMIC RTC drivers is as follows:

`linux/include/<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/pmic_rtc.h.`

13.7 Driver Configuration

This module can be selected using the ltib menu options.

To get to the PMIC RTC 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 PMIC RTC driver.

- Device Drivers > MXC Support Drivers > MXC PMIC Support > MC13783 Client Drivers > MC13783 Real Time Clock (RTC) support

  This is the configuration option to choose the MC13783-specific version of the RTC driver.

Chapter 14
i.MX31 Low-level Power Management Driver

The low-level Power Management (PM) driver implements dynamic frequency scaling (DFS) techniques and low-power modes. Dynamic voltage and frequency scaling (DVFS) is described in Chapter 15, "Dynamic Voltage Frequency Scaling (DVFS) Driver."

14.1 Overview

DFS is used to change the frequency when the Dynamic Power Management (DPM) level decides to change the operating point to meet the power requirements. This is done when the system is in RUN mode to conserve power. Low-power modes, such as WAIT, DOZE, and STOP are implemented to save power. In all these cases, power consumption is achieved by reducing the frequency and increasing the severity of clock gating.

14.1.1 Hardware Operation

The DFS operation and low-power modes on the MCU side are controlled by software using the clock controller module (CCM). The features of CCM are as follows:

- PLL control
- Dynamic frequency change (DFS) – Using dividers to change core frequency on the fly and PLL scaling to lock PLL
- Clock gating for various modules during low-power modes
- Low-power modes

14.1.2 Software Operation

For DFS operation, software is responsible for setting the desired frequency of ARM, AHB (MAX clock) and IP using either PLL scaling or using integer scaling (dividers). Core frequency depends on MAX clock and the PLL clock. For changing frequency using dividers, software should set the desired divider values in the divider register to enable frequency change. For PLL scaling, software should set the desired frequency value using PDF, MFD, and MFN registers in the CCM. For WAIT, DOZE, STOP and DSM low-power modes, software should disable interrupts before executing a wait-for-interrupt (WFI) instruction and re-enable interrupts afterwards.

14.2 Requirements

The Low-level PM driver API requires DPM to make the appropriate calls and pass the required arguments. The MCU clock domain is partitioned into four synchronous clocks and two sub-domains. The main clock of this domain is called `mcu_main_clk`, and it is the output of the MCU clock switch unit.

- mcu_clk (`ipg_clk_arm`) is the clock of the ARM platform. The target frequency of this clock is 532 MHz. This clock is generated from MCU BRM with a division factor as defined by the BRMM bits in PDR0.
- max_clk sub-domain (`ipg_clk_ahb`) is the clock domain of the internal ARM platform peripherals like the cross bar switch and chip modules. Clocks in this domain are generated from the max postdivider with a division factor as defined by the MAX_PDF bits in PDR0 register. These clocks should be an integer multiple (value of between 1 and 8) of the mcu_main_clk. Maximum target frequency of these clocks is 133 MHz.
- hsp_clk is the clock for the IPU. This clock is generated from the hsp postdivider with a division factor as defined by the HSP_PDF bits in PDR0 register. These clocks should be an integer multiple (value of between 1 and 8) of the mcu_main_clk. Maximum target frequency of this clock is 133 MHz for 1.2V supply.
- ipg_clk sub-domain is the clock domain of certain parts of the IP peripherals. These clocks are generated from the ipg postdivider with a division factor defined by the IPG_PDF bits in the PDR0 register. These clocks should be an integer multiple (either 1 or 2) of the max_clk. Maximum target frequency of these clocks is 62.5 MHz.
- nfc_clk (`ipg_clk_nfc_20m`) is the clock for Nand Flash controller. This clock is generated from the nfc postdivider with a division factor as defined by the NFC_PDF bits in the PDR0 register.
- ckil_mcu_sync_ipg is the clock for the peripheral modules. They require a 32 KHz clock
- ipg_clk_gacc_mbx_clk is the clock for the MBX module. It is 1/2 of the ipg_ahb_clk, which is 66 MHz.

14.3 Hardware Issues

i.MX31 silicon does support DSM, but there is no driver available on SDK1.3.

14.4 Source Code Structure

Table 14-1 lists the source files for i.MX31 available in the directory
`<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3/`

Table 14-1. PM Driver Source Files

File	Description
`mxc_pm.c`	Source file with all the implementation
`crm_regs.h`	Header File with all register and bit definitions for CCM module

The header file, `mxc_pm.h` (PM header file that contains the API declaration), associated with low-level PM driver is available in `<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/.`

14.5 Programming Interface

The following set of APIs are currently provided for frequency scaling and low-power modes:

- mxc_pm_intscale

  - Performs all the steps required to enable scaling on the fly using PCDR0 divider
  - DPM passes the required Core, AHB, and IPG frequency
- mxc_pm_pllscale
  - Performs all the steps required to enable PLL scaling
  - DPM passes the required Core, AHB, and IPG frequency
- mxc_pm_lowpower
  - Implements all the steps required to put the system under STOP, DOZE, or WAIT mode

Chapter 15
Dynamic Voltage Frequency Scaling (DVFS) Driver

The Linux Dynamic Voltage Frequency Scaling (DVFS) (designed as part of the CCM module) device driver allows simple S/W dynamic voltage frequency scaling. The frequency of the MCU clock domain and voltage of the chip can be changed on the fly with all modules, including MCU, continue running. The voltage of the chip can be changed by setting of DVS0 and DVS1 pins connected to the MC13783 Power and Audio Management IC (PMIC). The frequency of MCU clock domain can be changed by switching to alternate PLL clock (MCU or SR PLL's) with previous locking to required frequency or just changing post dividers division factors.

The software module is comprised of a Linux driver that allows privileged users to control and monitor the DVFS operation.

15.1 Hardware Operation

15.1.1 DVFS

The DFVS module is a power management module designed as part of the CCM module. The purpose of the DFVS module is to detect the appropriate operation frequency for the IC, considering the frequency of idle mode in the ARM core and considering other signals, using weights for such signals set by the user. The DFVS module generates an ARM interrupt or SDMA event when the frequency must be changed. The DFVS module records a log buffer for power patterns analysis and can generate an ARM interrupt or SDMA event each predefined time quantum for frequency change according to the log buffer.

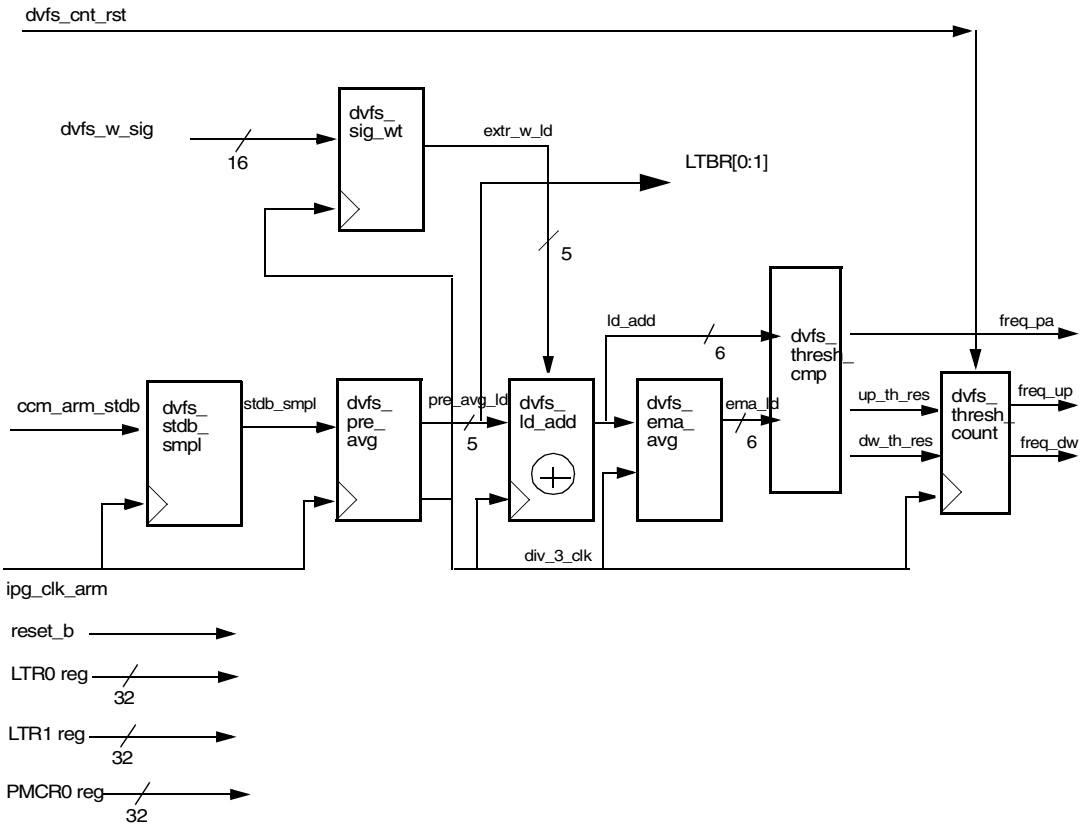


Figure 15-1. DVFS Load Tracking Module Block Diagram

The `dvfs_stdb_smpl` block samples the `ccm_arm_stdb` signal (ARM11 STANDBYWFI signal - idle state indicating) by `ipg_clk_arm` (ARM11 system clock). The `dvfs_pre_avg` block performs simple, non-overlapping averaging, reducing the sampling clock frequency and provide a level-based average index of the tracked CPU load. The `dvfs_sig_wt` block samples the 16 general purpose load signals, multiply each one of them by appropriate weight and sum products. The `dvfs_ld_add` block sums the CPU load, tracked by idle/non-idle signal and the load, detected from the additional load signals, weighted by `signal_weighting` block. The `dvfs_ema_avg` (EMA - Exponential Moving Average) block calculates an exponential moving average of the tracked CPU load. The `dvfs_thres_cmp` block compares the CPU load value to programmable threshold levels. The `dvfs_thres_count` block counts consecutive threshold overflows of `dw_th_res` and `up_th_res` (outputs of `threshold_comp` block).

15.1.2 Software Operation

The DVFS device driver is designed to monitor and control the DVFS hardware module, and perform the transitions between IC working points.

15.2 Source Code Structure

Table 15-1 lists the source files and headers available in the following directory:

`<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3/`

Table 15-1. Source Code Files

File	Description
`dvfs_v2.c`	Linux DVFS functions.

15.3 Linux Menu Configuration Options

None, DVFS is included by default.

15.3.1 Board Configuration Options

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

Chapter 16
Dynamic Process and Temperature Compensation (DPTC) Driver

The Dynamic Process Temperature Compensation (DPTC) Driver manages the DPTC power management technique. This technique reduces power consumption by adjusting the supply voltages according to the specific process case, chip fabrication and ambient temperature.

The DPTC hardware module (designed as part of the CCM module) monitors the current operating point using four reference circuits that test the chip process under the current ambient temperature.

The software module is a Linux driver that allows privileged users to control and monitor the DPTC operation.

16.1 Hardware Operation

The DPTC module is a power management module designed as part of the CCM module. The purpose of the DPTC module is to detect the minimum operation voltage for the IC, regarding process corner case and temperature for a given frequency. The DPTC module receives predefined values for process speed performance measurement and generates an interrupt if a supply voltage value update is required.

Figure 16-1 shows a block diagram of the DPTC hardware operation.

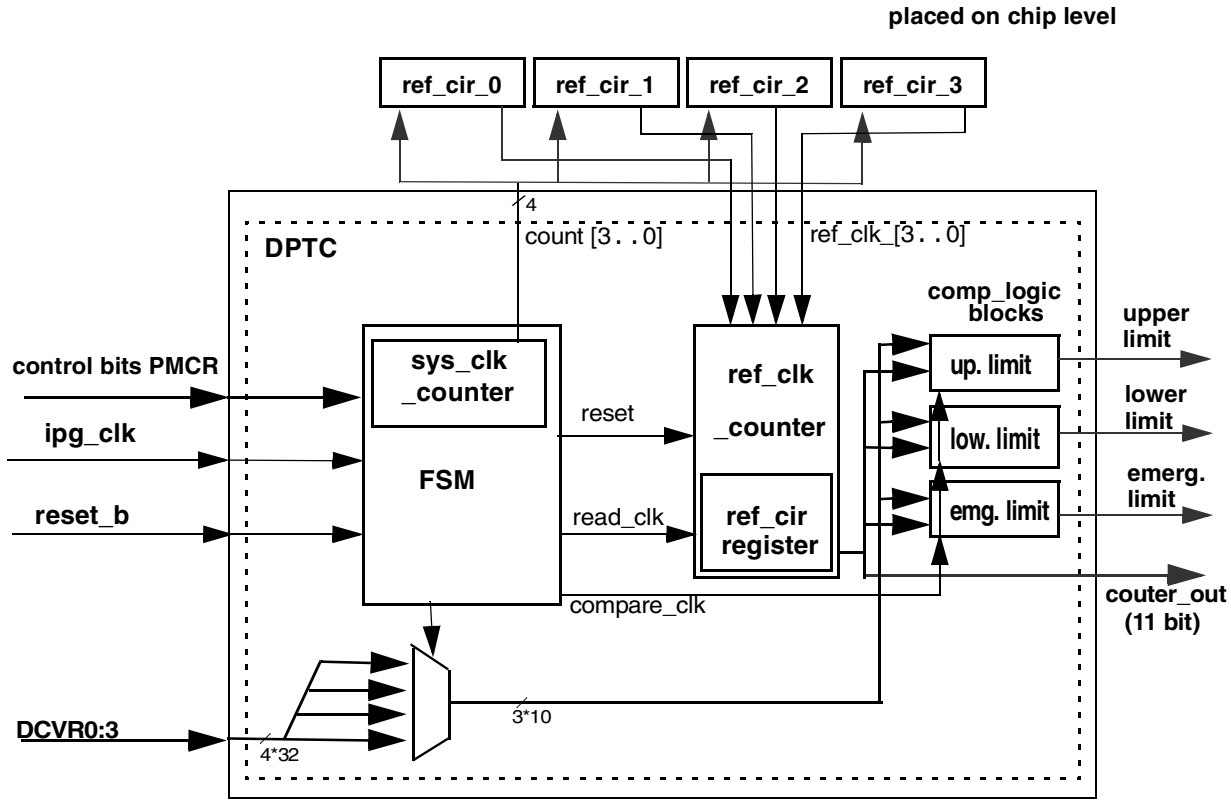


Figure 16-1. DPTC Hardware Module Design

The DPTC module contains four reference circuits (ref_cir_0 - ref_cir_3), control module (FSM module), counter and a comparison block.

The FSM module manages the operation of the DPTC module. On DPTC module enable, FSM selects one reference circuit (each circuit tests a different process parameter). The selected reference circuit then produces a clock signal (`ref_clk`), which is counted by the `ref_clk_counter`. After the measurement is completed, the `ref_clk_counter` value is compared with three threshold values: upper limit, lower limit and emergency limit. If one of the thresholds is exceeded, an interrupt is triggered.

On receiving an interrupt, the DPTC driver checks which of the thresholds was exceeded and changes the IC voltage and DPTC thresholds accordingly.

Figure 16-2 shows the FSM control loop.

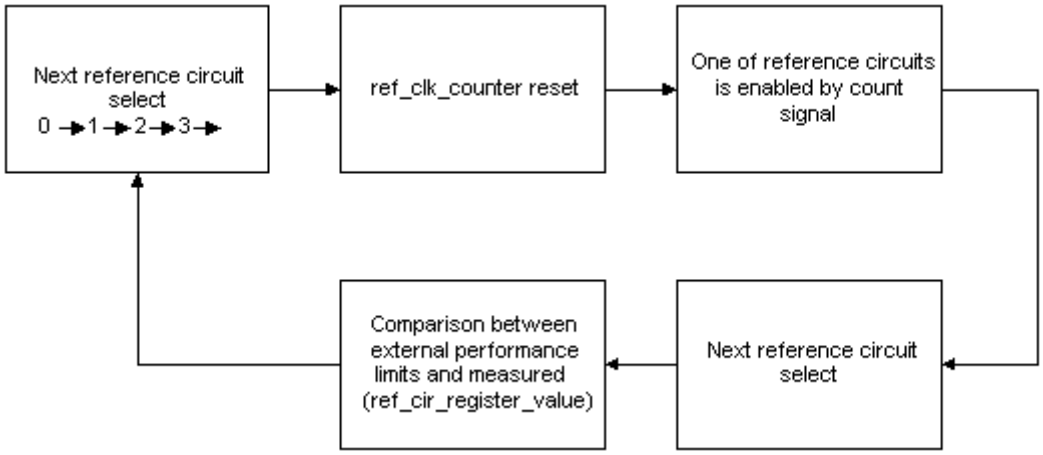


Figure 16-2. FSM Control Loop

16.2 Software Operation

The DPTC device driver is designed to monitor and control the DPTC hardware module, and it performs the transitions between IC working points. Figure 16-3 shows the DPTC driver high level software design.

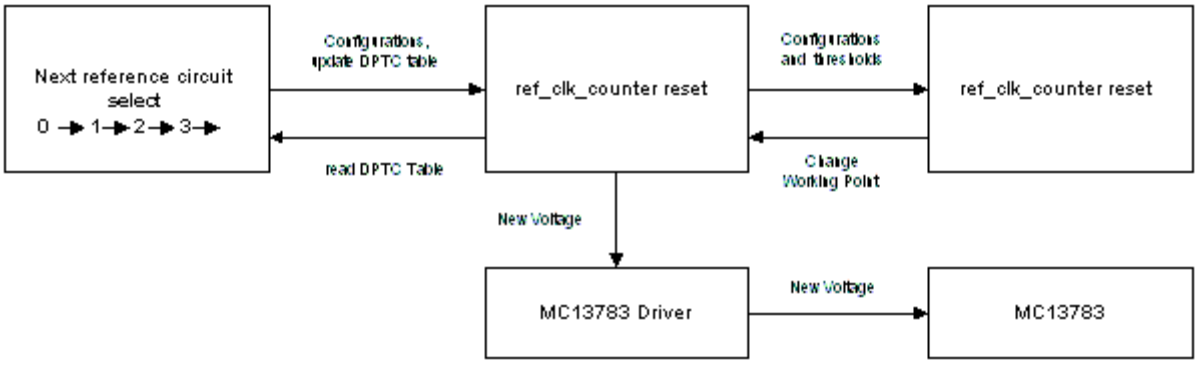


Figure 16-3. DPTC Driver Software Design

Driver operations are as follows:

1. DPTC user space software enables the DPTC driver.
2. The driver configures and enables the DPTC controller.
3. The DPTC controller measures the current IC working point and signals an interrupt if there is a need to move to another working point.
4. On receiving an interrupt, the DPTC driver calculates the new working point.
5. Using the DPTC lookup table, the driver calculates the new IC voltage and updates the current IC voltage through the MC13783 driver.
6. The driver writes new thresholds to the DPTC controller.
7. The DPTC controller starts a new measurement.

16.2.1 DVFS and DPTC – MC13783 Interaction

The DVFS and DPTC drivers use MC13783 to change the voltage of the chip.

On a DPTC working point change request, the driver sets values for four different voltages on the MC13783:

- SW1A SW setting
- SW1A DVF setting
- SW1B DVS setting
- SW1B STANDBY setting

The change is done using MC13783 Power API functions, through SPI. After the voltage change, the DPTC is enabled when it gets a power-ready interrupt from MC13783. This signal comes from the PWRRDY pin of the MC13783, and it is connected to the GPIO1_5 pin of the multimedia application processor.

On a DVFS frequency change request, the driver selects one of the four voltages according to the new frequency. The change is done by writing to the DVSUP[0-1] bits of the CCM PMCR0 register. These bits are connected to the DVFS0 and DVFS1 output pins of the multimedia application processor, and these pins are connected to the DVSSW1A and DVSSW1B pins of the MC13783.

16.3 Requirements

The DPTC driver implements the following requirements:

- [R-DPTC-1] The DPTC driver allows a privileged user to control the DPTC operation and contains the following features:
  - Enable/Disable module
- [R-DPTC-2] On DPTC interrupt, the driver updates the current IC voltage according to the DPTC controller measurements.

16.4 Source Code Structure

Table 16-1 lists the source files and headers available on the following locations:

```
<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3/
<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/plat-mxc
```

Table 16-1. Source Code Files

File	Description
`mach-mx3/dptc.c`	DPTC table
`plat-mxc/dptc.c`	DPTC driver

16.5 Configuration

DPTC driver is included by default.

Chapter 17
CH7024 TV Encoder (TV-Out) Driver

The CH7024 is a TV encoder device targeting handheld, portable video applications, such as digital still cameras and similar portable embedded systems. The device is able to encode the video signals and generate synchronization signals for NTSC and PAL standards. Supported TV output formats are NTSC-M, NTSC-J, NTSC-433, PAL-B/D/G/A/I, PAL-M, PAL-N, and PAL-60.

17.1 TV-Out Driver Overview

The CH7024 takes in digital graphics input, which is the output of the IPU Synchronous Display Controller (SDC), and converts it to TV output. In the IPU SDC controller, only one set of synchronous display signals can be output at the same time. The i.MX platform puts the LCD and CH7024 signal control and data pins together, therefore, the framebuffer cannot be displayed on both the LCD and TV-out simultaneously. There needs to be a dynamic switch between the LCD and TV-out display output devices. The CH7024 registers get configured through its I^2C port. At present the driver only supports PAL-B/D/G/A/I and NTSC-M output formats in SDTV mode.

CH7024 supports two operating modes, SDTV encoder (NTSC/PAL) with non-interlaced input and SDTV encoder (NTSC/PAL) with interlaced input. In the first mode CH7024 can take non-interlaced data from graphics controller and encode it to analog NTSC and PAL waveforms. In the second mode it can take interlaced data from sources and perform SDTV encoding. The driver supports the first operating mode with non-interlaced input.

The TV-out driver implements an I^2C client driver and a framebuffer driver in the Linux kernel. The I^2C client driver implements the configurations to CH7024 registers through the I^2C interface. The framebuffer driver implements IPU SDC configurations and the digital graphics input to CH7024.

The driver is enabled by selecting the tvout option under the graphics parameters in the kernel configuration.

17.1.1 Hardware Operation

The CH7024 provides a digital interface to most GCCs (In i.MX, it is the synchronous LCDC, for example, IPU SDC). It accepts computer-generated digital graphics input in RGB or YCrCb format. The CH7024 receives initialization and basic configuration information through its I^2C-compatible SIO port with simple register Read/Write commands. The valid outputs are SDTV (PAL/NTSC). The driver implements the SDTV (PAL/NTSC).

There is no specific hardware operation for CH7024 hardware.

17.1.2 Software Operation

The driver implements the TV-Encoder SDTV output format configuration (NTSC or PAL).

The driver switches the current display output device from LCD to TV-out (power off the LCD panel, disable the current SDC output, setup the CH7024 and reconfigure the SDC to output appropriate signal to CH7024). Then the TV-Out framebuffer device is used by applications.

CH7024 registers are accessed through the I^2C interface. The driver uses the common kernel I^2C client driver to configure the CH7024 registers. The I^2C client driver can not be accessed directly in user space.

The TV-out architecture diagram is shown in Figure 17-1.

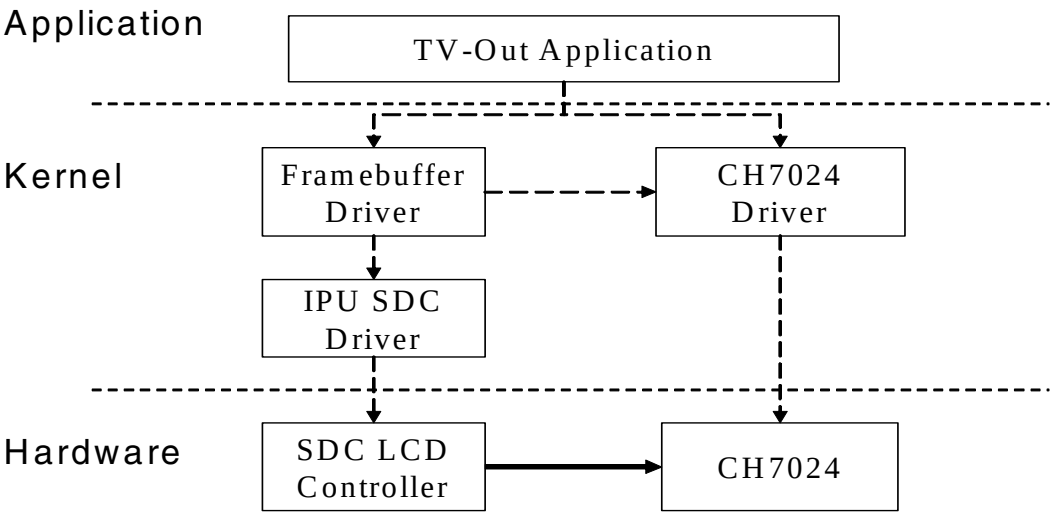


Figure 17-1. TVout Driver in the Architecture

17.2 Source Code Structure Configuration

Table 17-1 describes the source files associated with the TV-out driver, which are available in the directory `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/video/mxc`.

Table 17-1. TV-Out Driver Source File

File	Description
ch7024.c	Source file for CH7024 TV-Out driver

Table 17-2 describes the source files associated with the framebuffer drivers which use TV-out driver are available in the directory `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/video/mxc`.

Table 17-2. Framebuffer Driver Source Files

File	Description
mxcfb.c	Source file for LCD framebuffer driver. Provides SDC LCD disable/enable interface to mxcfb_tvout module for output device switching.

17.3 Driver Configuration

This module can be selected using the ltib menu options.

To get to the CH7024 TV Out Encoder 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 CH7024 TV Out Encoder driver.

- Device Drivers > Graphics support > CH7024 TV Out Encoder
  Chooses the CH7024 TV Out Encoder driver.

Chapter 18
Image Processing Unit (IPU) Drivers

The image processing unit (IPU) is designed to support video and graphics processing functions in the MXC architecture and to interface with video and still image sensors and displays.

18.1 IPU Hardware Operation

The detailed hardware operation of the IPU is discussed in the hardware documentation.

18.2 IPU Software Operation

Figure 18-1 depicts the interaction between the different graphics/video drivers and the IPU.

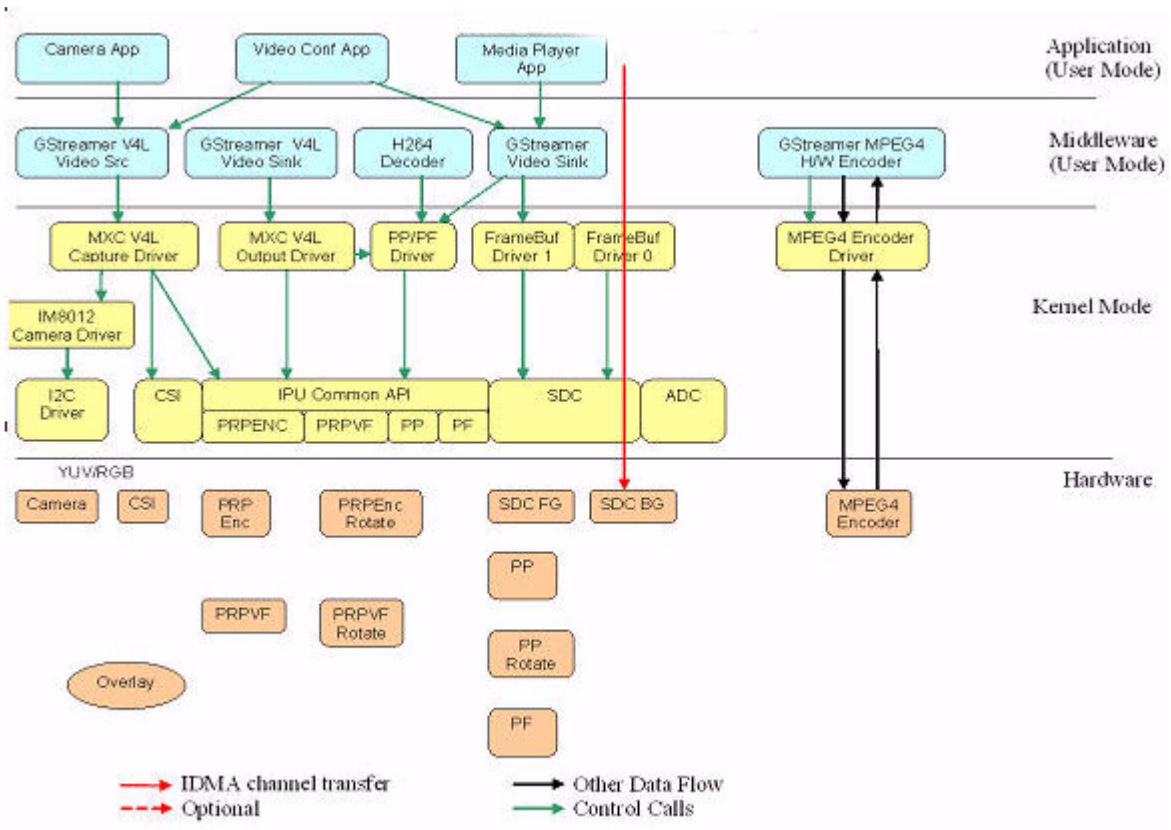


Figure 18-1. Graphics/Video Drivers Software Interaction

The IPU drivers are sub-divided as follows:

- Device drivers – include the frame buffer driver for SDC, the frame buffer driver for Epson LCD, 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-2.6.26/drivers/video/mxc` directory of the Linux kernel.

  The V4L2 device drivers are available in the `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/media/video` directory of the Linux kernel.
- Low-level library routines – interfaces 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-2.6.26/drivers/mxc/ipu` directory of the Linux kernel.

18.2.1 IPU Frame Buffer Drivers Overview

The frame buffer device provides an abstraction for the graphics hardware. It represents the frame buffer of 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, usually located in the `/dev` directory, for example, `/dev/fb*`.

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.

18.2.1.1 IPU Frame Buffer Hardware Operation

The frame buffer interacts with the IPU hardware driver module.

18.2.1.2 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-2.6.26/include/linux/fb.h`. The following are a few of the ioctl 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's 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).

All this hardware abstraction makes the implementation of application programs easier and more portable. The Qt/Embedded server works completely on `/dev/fb*` and thus is not required to know, for example, the organization of the color registers of the hardware. 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-2.6.26/drivers/video/mxc/mxcfb.c`) interacts tightly with the generic Linux frame buffer driver (`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/video/fbmem.c`).

18.2.1.3 SDC Frame Buffer Driver

The SDC Frame buffer screen driver implements a Linux standard Frame buffer driver API for synchronous LCD panels or those without memory. The SDC 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 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 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. By default, the frame buffer driver supports the Sharp QVGA panel.

The Frame buffer interacts with the IPU Driver using custom APIs:

- 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:

- Support for Sharp QVGA, Sharp/NEC/EPSON/CH7024-TVEncoder VGA panels and CLAA WVGA panels
- Configurable screen resolution
- Configurable RGB 16(VGA/WVGA only support 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.

18.2.1.4 ADC Frame Buffer Driver

The ADC Frame buffer screen driver implements a Linux standard Frame buffer driver API for asynchronous or smart LCD panels. The ADC Frame buffer screen driver is the top level kernel video driver that interacts with the kernel and user level applications. This 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 depends on the VT console for switching from serial to graphics mode.

The Frame buffer interacts with the IPU Driver using custom APIs:

- Initialization of panel interface settings for serial or parallel mode
- Initialization of IPU channel settings for ADC commands and data
- Control of IPU auto-refresh and/or bus snooping for automatic update of panel memory

The following features are supported:

- Support for Epson L2F60012P00 dual mode 176x220 panel
- Configurable RGB 16, 24 or 32 bits per pixel frame buffer
- 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

18.2.2 IPU backlight Driver

The IPU backlight driver implements IPU PWM backlight control for SDC and ADC panel. It exports a sys control file under `/sys/class/backlight/mxc_ipu_bl.0/brightness` to user space. The max backlight intensity value is 255 with default of 127.

18.2.3 Video for Linux 2 (V4L2) APIs

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

The V4L2 capture device includes 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.

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

The V4L2 Output driver uses the IPU post-processing functions for video output. The driver implements the standard V4L2 API for output devices.

18.2.3.1 V4L2 Capture Device

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-2.6.26/driver/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, while `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 has turned on the AE and AWB turn on. The frame rate may change, depending on light sensor samples.

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 degree rotation
- 4—90 degree rotation clockwise
- 5—90 degree rotation clockwise and vertical flip
- 6—90 degree rotation clockwise and horizontal flip
- 7—90 degree rotation counter-clockwise

Figure 18-2 shows a block diagram of V4L2 Capture API interaction.

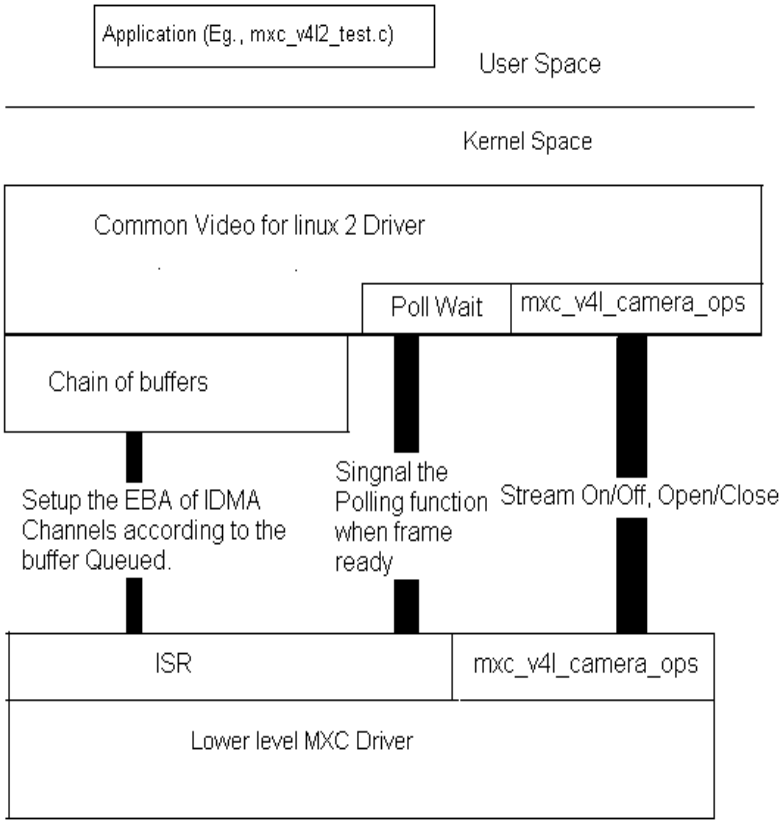


Figure 18-2. Video4Linux2 Capture API Interaction

18.2.3.2 Use of the V4L2 Capture APIs

A sample V4L2 capture process is shown in the following procedure:

1. The application sets the capture pixel format and size by `ioctl` VIDIOC_S_FMT.
2. The application sets the control information by `ioctl` VIDIOC_S_CTRL for rotation usage.
3. The application 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. The application memory maps the buffer to its user space.
5. The application queues buffers using the ioctl command VIDIOC_QBUF.
6. The application 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. The application takes the buffer from the queue using the `ioctl` VIDIOC_DQBUF. This `ioctl` blocks until it has been signaled by the ISR driver.
8. The application stores the buffer to a Ycrcb file.

9. The application replaces the buffer in the queue of the V4L2 driver by executing VIDIOC_QBUF again.

V4L2 still image capture:

1. The application sets the capture pixel format and size by executing the `ioctl` VIDIOC_S_FMT.
2. The application reads one frame still image with YUV422.

V4L2 overlay support use case:

1. Application set the overlay window by `ioctl` VIDIOC_S_FMT.
2. Application turn on overlay task by `ioctl` VIDIOC_OVERLAY.
3. Application turn off overlay task by `ioctl` VIDIOC_OVERLAY.

18.2.3.3 V4L2 Output Device

V4L2 output device support can be selected during kernel configuration. The driver is available at `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/media/video/mxc/output/mxc_v4l2_output.c`.

The following V4L2 features are supported by the driver:

- Direct output to the SDC foreground overlay plane (no ARM processor intervention, and synchronized to LCD refresh)
- Support for color keying alpha blending of the frame buffer and overlay planes
- Support for linking post-processing resize and CSC, rotation, and display IPU channels for 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
- Support for RGB 16, 24, and 32 bit, YUV 4:2:0 and 4:2:2 planar, and YUV 4:2:2 interleaved input formats
- Support for TV output (features TBD)

These features are supported using custom APIs:

- Output to user buffer instead of overlay display
- Programmable rotation

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 degree rotation
- 5—90 degree rotation and vertical flip
- 6—90 degree rotation and horizontal flip
- 7—90 degree rotation with horizontal and vertical flip

18.2.3.4 Use of the V4L2 Output APIs

The following procedure shows a sample V4L2 capture use case that uses the V4L2 output APIs:

1. The application sets the capture pixel format and size using `ioctl` VIDIOC_S_FMT.
2. The application sets the control information using `ioctl` VIDIOC_S_CTRL, for rotation.
3. The application 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. The application memory maps the buffer to its user space.
5. The application executes the `ioctl` VIDIOC_DQBUF.
6. The application passes the data that requires post-processing to the buffer.
7. The application queues the buffer using the `ioctl` command VIDIOC_QBUF.
8. The application starts the stream by executing `ioctl` VIDIOC_STREAMON.

18.2.4 MPEG4/H.264 Post Filter Driver

The Post-filtering driver provides a custom user API for IPU post-filtering functions. The following features are supported by the driver:

- Support for MPEG4 dering and/or deblock

- Support for H264 deblock
- Support for intra-frame pause and resume (H.264 only)
- Synchronous and asynchronous operation
- Support for driver-allocated or user-allocated buffers

The post-filter driver implements ioctls for initialization, release, buffer allocation, and beginning the processing for a frame.

18.3 IPU Source Code Structure Configuration

Table 18-1 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-2.6.26/drivers/mxc/ipu
<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/video/mxc
<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/media/video/mxc
<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/video/backlight
```

Different ioctls for both Display panels will be provided in future releases.

Table 18-1. IPU Source and Header File List

File	Description
drivers/mxc/ipu/ipu_adc.c	ADC configuration driver
drivers/mxc/ipu/ipu_common.c	Configuration functions for ADC and SDC
drivers/mxc/ipu/ipu_csi.c	CMOS sensor interface functions
drivers/mxc/ipu/ipu_ic.c	IPU library functions
drivers/mxc/ipu/ipu_sdc.c	SDC configuration driver
drivers/mxc/ipu/ipu_device.c	IPU driver device interface and fops functions.
drivers/media/video/mxc/capture/mc521da.c	Camera sensor driver for sMC521DA
drivers/media/video/mxc/capture/ov2640.c	Camera sensor driver for OV2640
drivers/media/video/mxc/capture/mt9v111.c	Camera sensor driver for MT9V111
drivers/media/video/mxc/capture/ipu_prp_enc.c	Pre-processing encoder driver
drivers/media/video/mxc/capture/ipu_prp_vf_adc.c	Pre-processing view finder (adc) driver.
drivers/media/video/mxc/capture/ipu_prp_vf_sdc.c	Pre-processing view finder (sdc foreground) driver.
drivers/media/video/mxc/capture/ipu_prp_vf_sdc_bg.c	Pre-processing view finder (sdc background) driver.
drivers/media/video/mxc/capture/ipu_still.c	Pre-processing still image capture driver
drivers/mxc/ipu/pf/mxc_pf.c	Post filtering driver
drivers/video/mxc/mxcfb.c	Framebuffer driver for SDC
drivers/video/mxc/mxcfb_epson.c	Framebuffer driver for ADC
drivers/video/mxc/mxcfb_epson_qvga.c	Framebuffer driver for ADC QVGA
drivers/video/mxc/mxcfb_epson_vga.c	Framebuffer driver for SDC VGA

Table 18-1. IPU Source and Header File List(Continued)

File	Description
drivers/video/mxc/mxcfb_claa_wvga.c	Framebuffer driver for SDC WVGA
drivers/video/mxc/mxcfb_modedb.c	Parameter settings for Framebuffer devices
drivers/media/video/mxc/capture/mxc_v4l2_capture.c	V4L2 capture device driver
drivers/media/video/mxc/output/mxc_v4l2_output.c	V4L2 output device driver
drivers/video/backlight/mxc_ipu_bl.c	IPU backlight control driver

Table 18-2 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-2.6.26/drivers/mxc/ipu
<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc
<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/media/video/mxc
```

Table 18-2. IPU Global Header File List

File	Description
drivers/mxc/ipu/ipu_param_mem.h	Helper functions for IPU parameter memory access
drivers/mxc/ipu/ipu_prv.h	Header file for Pre-processing drivers
drivers/mxc/ipu/ipu_regs.h	IPU register definitions
drivers/media/video/mxc/capture/mt9v111.h	Header file for MT9V111 sensor driver
include/asm-arm/arch-mxc/mxc_pf.h	Header file for Post filtering driver
include/asm-arm/arch-mxc/mxcfb.h	Header file for framebuffer driver for SDC
drivers/media/video/mxc/capture/ipu_prp_sw.h	Header file for IPU PRP use case driver.
drivers/media/video/mxc/capture/mxc_v4l2_capture.h	Header file for V4L2 capture device driver
drivers/media/video/mxc/output/mxc_v4l2_output.h	Header file for V4L2 output device driver

18.4 IPU Linux 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 Drivers > 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 Drivers > 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. By default, this option is M for i.MX31 platforms.
- CONFIG_MXC_IPU_PRP_VF_SDC—Option for the IPU:

  ```
  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 adapter > MXC Video For Linux Camera > MXC
  Camera/V4L2 PRP Features support > Pre-Processor VF SDC library
  ```

  By default, this option is Y for all.
- CONFIG_MXC_IPU_PRP_VF_ADC—Options for the IPU:

  Use case driver for the rotation

  ```
  CSI > IC > MEM MEM > IC (ROT) > MEM MEM > ADC
  ```

  or for smart sensors

  ```
  CSI > IC > ADC.
  ```

  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 VF SDC library
  ```

  By default, this option is Y 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 Y for all.
- CONFIG_MXC_IPU_PF—This is configuration option for MXC MPEG4/H.264 Post Filter Driver. This option is dependent on “MXC_IPU” option. In menuconfig, this option is available under:

  ```
  Device Drivers > MXC support drivers > MXC MPEG4/H.264 Post Filter Driver
  ```

  By default, this option is Y 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 Frame buffer 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 Frame buffer 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 Frame buffer 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 Frame buffer 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 Frame buffer 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 Frame buffer support > Synchronous Panel Framebuffer > FS453 TV Out Encoder
  ```
- CONFIG_FB_MXC_ASYNC_PANEL-This configuration option selects the asynchronous panel framebuffer. This option is dependent on “CONFIG_FB_MXC” option. In `menuconfig`, this option is available under:

  ```
  Device Drivers > Graphics support > MXC Frame buffer support > Asynchronous Panels
  ```

By default, this option is N for all architectures.

- CONFIG_FB_MXC_EPSON_PANEL—This configuration option selects the Epson panel. This option is dependent on “CONFIG_FB_MXC_ASYNC_PANEL” option. In `menuconfig`, this option is available under

  ```
  Device Drivers > Graphics support > MXC Frame buffer support > Asynchronous Panels >
  Asynchronous Panel Type > Epson 176x220 Panel
  ```

  By default this option is N for all architectures.

18.5 IPU Programming Interface

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

Chapter 19
MBX Driver

MBX refers to an integrated 3D graphics accelerator–the ARM MBX R-S accelerator. A base driver for the MBX is provided by ARM Inc. and has been ported by Freescale. Various documents on the user-API for the MBX driver are available. The link to these documents is provided in this chapter.

19.1 Hardware Operation

The MBX R-S 3D Graphics Core has the following features:

- Deferred texturing
- Screen tiling
- Flat and Gouraud shading
- Perspective correct texturing
- Specular highlights
- Floating-point Z-buffer
- 32-bit ARGB internal rendering and layer buffering
- Full tile blend buffer
- Z-load and store mode
- Per-vertex fog
- 16-bit RGB textures, 1555, 565, 4444, 8332, 88
- 32-bit RGB textures, 8888
- YUV 422 textures
- PVR-TC compressed textures
- One-bit textures for text acceleration
- Point, bilinear, trilinear, and anisotropic filtering
- Full range of OpenGL and *Direct3D* (D3D) blend modes
- Dot3 bump mapping
- Alpha test
- Zero-cost full-scene anti-aliasing
- 2Dvia3D

19.2 Software Operation

The MBX drivers are based on proprietary code and can be released only as kernel modules. These kernel modules are a part of the filesystem released along with the Linux BSP and can be loaded using a script

that has been provided. The MBX driver supports the OpenGLES 1.1 standard. An SDK and other related documentation is available for developers from http://www.imgtec.com/powervr/insider/sdkdownloads/. The PowerVR OpenGL ES SDK provides a set of documentation, source code and utilities that help developers create applications using the OpenGL ES graphics library on PowerVR platforms.

19.3 Requirements

This MBX driver conforms with the OpenGL ES Application Programmer Interface (API) for user space applications

19.4 Source Code Structure

The MBX module driver source code is not available as part of the standard Linux BSP release. Contact Freescale support to access source code for the driver.

The following header files are needed to build the OpenGLES application:

`http://www.khronos.org/opengles/headers/1_1/egl.h`

`http://www.khronos.org/registry/gles/api/1.1/gl.h`

`http://www.khronos.org/registry/gles/api/1.1/glplatform.h`

Table 19-1 lists the modules and libraries associated with MBX.

Table 19-1. MBX Related File List

File	Description
libclcdc.so, libGLES_CM.so, libpvrmmap.so, libsrv_um.a, libswcamera.so	MBX related libraries that are part of the Linux BSP filesystem. These libraries are located in /usr/lib/
pvr.ko, clcdc.ko	Kernel level modules for the MBX driver and the display driver. These modules are located in /lib/modules/2.6.26-*/kernel/drivers/char/
rc.pvr	Initialization script to load the MBX drivers. The script is present in /etc/rc.d/init.d/.
`services_test`, `egl_test`	Test binaries for MBX driver. Theses are located in /usr/local/bin/

19.5 Configuration

19.5.1 Linux Menu Configuration Options

This module can be selected using the ltib menu options.

To get to the MBX use the command `./ltib -c` when located in the `<ltib dir>`.

- Package list > mbx-bin

This package provides proprietary binary kernel modules, libraries, and test code built from the MBX OpenGL ES (GX200) DDK.

For the MBX demo to work correctly, the kernel framebuffer depth needs to be set to 16 bpp and the overlay framebuffer should not be selected.

19.5.2 MBX Filesystem Setup

The MBX can be initialized using the `rc.pvr` script in the `/etc/rc.d/init.d/` directory. To load the drivers, type `/etc/rc.d/init.d/rc.pvr start` on the terminal console. To unload the driver, type `/etc/rc.d/init.d/rc.pvr stop`.

19.6 Programming Interface

19.6.1 User Space API

Refer to the API related documents from the PowerVR developer's site (http://www.pvrdev.com/) and the Khronos site (http://www.khronos.org/) for other OpenGL resources.

Chapter 20
Hantro VGA Video Encoder Driver

20.1 Overview

The integrated hardware VGA video encoder is from Hantro. The encoder is operated through an Application Programming Interface (API). The encoder API usage can be seen graphically in Figure 20-1.

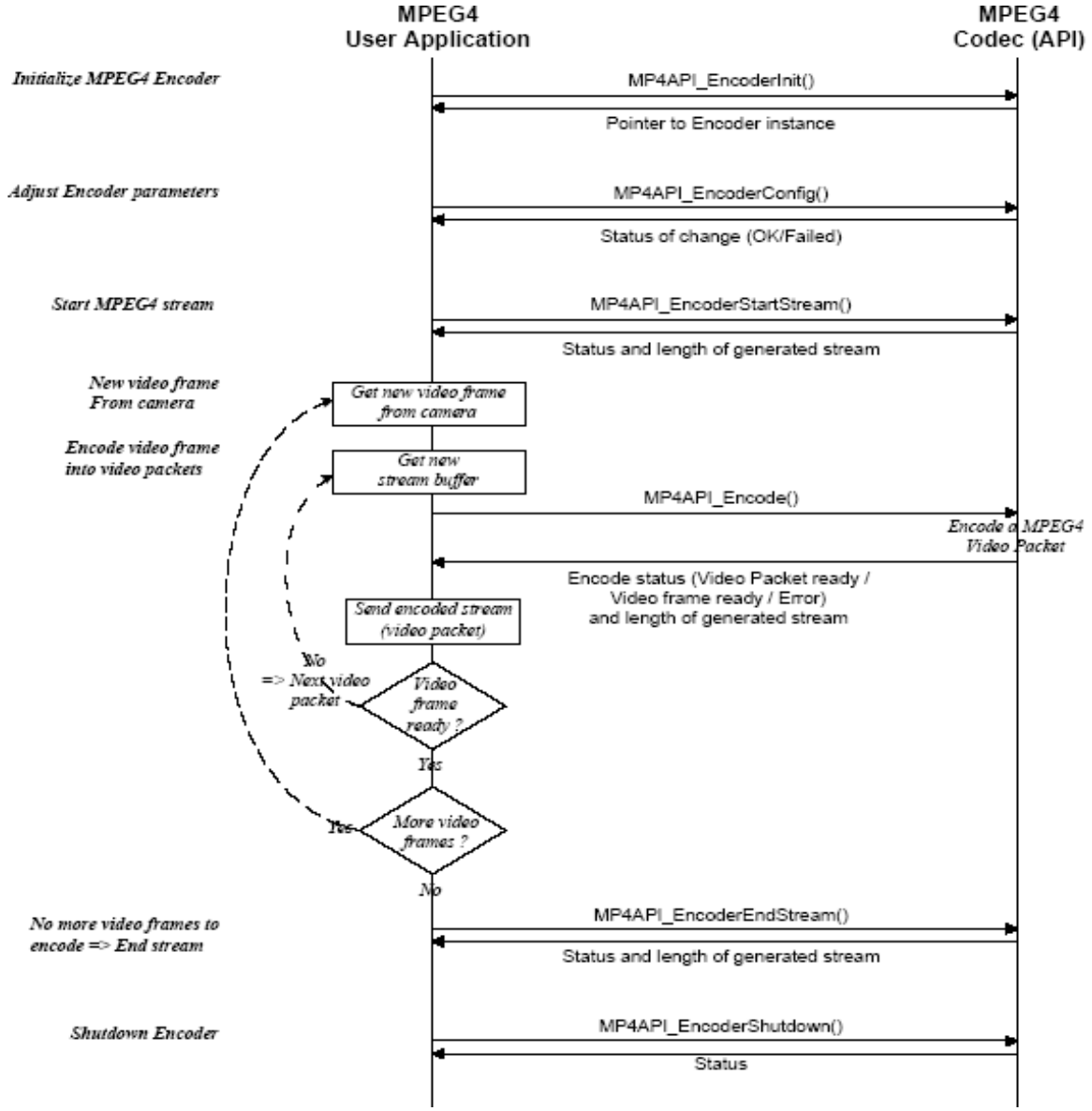


Figure 20-1. Encoder API Usage

Figure 20-1 is a combination of the message sequence chart and the flow chart showing a standard operation sequence.

20.1.1 Hardware Operation

The internal registers of Hantro Encoder are mapped to the user space, which are accessed by the API provided from Hantro.

20.1.2 Software Operation

As shown in Figure 20-2, The encoder kernel module is a character driver which is loaded into the kernel to use Hantro Encoder API library. The static library (Encoder API) provided by Hantro uses this kernel module to get device data. The kernel module allocates memory and maps the internal registers and memory buffers to user space.

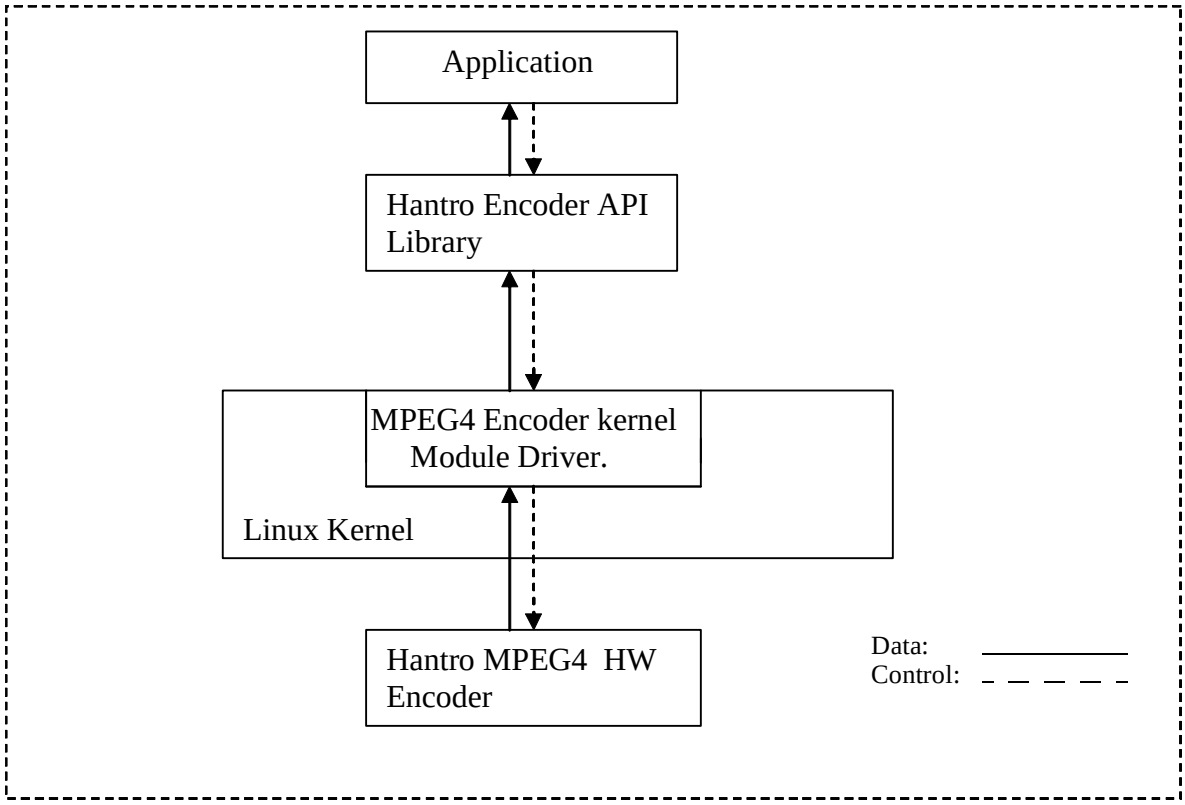


Figure 20-2. Block Diagram of Encoder

20.2 Requirements

The following requirements have been met in the Encoder driver:

- The driver supports MPEG4 API version 3.1 for H.263 Encoder (reference from *User Manual version 1.0, Hantro products, 2004*).

- This MPEG4 Encoder kernel driver module is used by Encoder library provided by Hantro (libmeg4enc.a).
- The example application provided with Hantro Encoder API manual works.

20.3 Source Code Structure

Table 20-1 lists the source files available in the source directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/hmp4e`.

Table 20-1. FIRI File List

File	Description
`mxc_hmp4e.h`	Header file for Encoder driver
`mxc_hmp4e.c`	Encoder driver source

20.4 Configuration

The Encoder driver is accessed by the following interface:

```
$/dev/hmp4e
MPEG4 Encoder memory map           : Base address:53FC8000 to 53FCBFFF.
MPEG4 Encoder IRQ pin no           : 5.
MPEG4 Encoder IO size              : (35 * 4) bytes.
MPEG4 Encoder Buffer size          : 1048576 bytes.
```

20.4.1 Linux Menu Configuration Options

The Linux kernel configuration, `CONFIG_MXC_HMP4E`, is provided for this module. This is the configuration option for the Hantro Encoder driver. In the `menuconfig` this option is available under Device Drivers > MXC support drivers > MXC MPEG4/H.264 Post Filter Driver.

Chapter 21
OmniVision Camera Driver (OV2640)

The OV2640FSL is an on-board camera sensor and lens module designed for mobile applications where low power consumption and small size are of the utmost importance. The camera driver is located under the Linux V4L2 architecture. It implements the V4L2 capture interfaces.

Applications cannot use the camera driver directly. Instead, the applications use the V4L2 capture driver to open and close the camera for preview and image capture, controlling the camera, getting images from camera, and starting the camera preview.

21.1 Hardware Operation

The OV2640FSL uses the serial camera control bus (SCCB) interface to control the sensor operation. It works as an I^2C client, and CSI interface of IPU works as the I^2C master, which uses I^2C bus to control camera's operation.

The CSI interface of IPU also provides the sensor clock to the camera when the camera is working so that IPU can get image data from camera through CSI interface. The pixel clock, horizontal reference output and vertical synchronization output generated from camera are used by CSI interface to get image data from camera.

Refer to OV2640 and OV2640FSL datasheet to get more information on the sensor. Refer to the datasheet for the platform to get more information on CSI and IPU.

21.2 Software Operation

The camera driver implements V4L2 capture interface, and applications use V4L2 capture interface to operate the camera. The supported operations of V4L2 capture are preview, capture stream mode, capture still mode, rotation, and resize.

The supported picture formats are RGB565, RGB24, BGR24, RGB32, BGR32, YUV422P, UYVY, and YUV420.

21.3 Source Code Structure

Table 21-1 lists the camera driver source files available in the `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/media/video/mxc/capture` directory.

Table 21-1. Camera File List

File	Description
`ov2640.c`	camera driver implementation

21.4 Linux Menu Configuration Options

The Linux kernel configuration option, CONFIG_MXC_CAMERA_OV2640, is provided for the module. This is the configuration option for the OV2640 camera driver. In menuconfig, this option is available under Device Drivers > Multimedia device > Video For Linux > Video Capture Adapter > MXC Camera/V4L2 PRP Features support. This option is dependent on the CONFIG_VIDEO_MXC_IPU_CAMERA option. By default, this option is M.

Chapter 22
Advanced Linux Sound Architecture (ALSA) Sound Driver with PMIC Hardware Support

This section explains the Advanced Linux Sound Architecture (ALSA) driver. Additional documentation on ALSA can be found at www.alsa-project.org.

ALSA has the following components:

- ALSA `utils` (`aplay. arecord, alsamixer`) – open source utilities that invoke APIs of the ALSA user space library to access the kernel drivers and hardware.
- User space ALSA library (`libasound`)
- Kernel drivers – hardware abstractions that directly map to some hardware entity. Anything else that can be done in software (such as resampling, mixing, snooping, and so on) is handled in user space as plug-ins.

ALSA can work in the following modes:

- Native or ALSA mode in which the applications go through a user space library. Here the applications do not perform operations on the device files directly.
- OSS emulation mode in which the kernel ALSA driver emulates OSS for all practical purposes. OSS compatible applications can directly perform operations on the device files. In this case the compatibility between OSS style and ALSA is completely handled by the ALSA middle layer.

ALSA provides following types of interfaces to user space:

- Operational interface through `/dev/snd/` (PCM components for capture and playback, control components, MIDI devices, sequencer devices and a timer)
- Status and configuration interface through `/proc/asound`

22.1 ALSA Features and Components

The sections below describe the ALSA sound driver as applicable to Linux platforms based on Freescale's i.MX family of processors. This audio driver was ported to provide ALSA and OSS compatible applications with the means to perform audio playback and recording functions using the audio components provided by Freescale's PMIC chips.

The operational interfaces exported through `/dev/snd/` in this release of BSP are as follows:

- PCM interfaces for playback (2)
- PCM interface for recording (1)
- Control interface for mixer operations (1)

22.1.1 Current BSP Release Support

- 8 kHz through 96 kHz Stereo and Mono playback on `/dev/snd/pcmC0D0p` in native mode and `/dev/sound/dsp` in OSS emulation mode
- 8 kHz and 16 kHz mono playback on `/dev/snd/pcmC0D1p` in native mode
- 8 kHz and 16 kHz mono recording on `/dev/snd/pcmC0D0c` in native mode
- Mixer operations to control input/output devices, playback/recording gains, balance and mono adder configurations on `/dev/snd/controlC0`
- Playback Stream Mixing, that is, mixing of two audio streams during playback. The audio driver supports mixing of two audio streams. Mixing can be achieved in two ways:
  - Analog Mixing – mixing the two streams in the analog domain after the DAC or the CODEC that is, after the streams have been decoded and before being passed to the output.
  - Digital Mixing – mixing the two streams (mono) before they are decoded when they are still in digital format. The two audio streams are mixed in the SSI and then the combined stream is routed to the VCODEC for playback.

22.1.2 PCM Components

ALSA exports PCM devices: pcmC0D0p, pcmC0D0c, pcmC0D1p, and pcmC0D2p (if mixing is enabled) in `/dev/snd` and dsp and adsp in `/dev/sound` for OSS compatibility.)

- C0 indicates sound card 0
- D0 or D1 indicates PCM device ID (There can be multiple PCM devices attached to one sound card)

Please note that these device files in `/dev/snd` and `/dev/sound` map to the same hardware but are exported differently as per ALSA native and OSS emulation requirements.

Each PCM component maps to PCM device in the kernel that can have one playback and one capture stream. Each stream can have multiple substreams.

In cases where the audio chip supports four identical DACs, they can be represented as one playback stream with four substreams and allocating a substream upon device open is handled by the ALSA middle layer.

22.1.3 Control Components

ALSA exports one control component in `dev/snd` as controlC0. The same control can be reached in OSS emulation mode with the help of /dev/mixer device.

Controls are registered with a sound card as a linked list of `Kcontrol` structures identified by index, name and interface. These control components can be accessed with `amixer` and `alsamixer` utilities. Currently controls have been provided to vary playback volume, recording gain, playback balance, mono adder configuration, output device selection and input device selection.

22.2 Hardware Operation

The ALSA sound driver provides interfaces between audio applications that run in user mode and the hardware. The platform components that are used by the ALSA sound driver include the following:

- The Digital Audio MUX – selects the path for transferring the digital audio stream to and from the PMIC. Reconfiguring the Digital Audio MUX can direct a digital audio stream to either the Voice CODEC or the Stereo DAC. The Digital Audio MUX can also be used to select an audio stream from either the ARM or DSP cores, but this feature is not currently implemented.
- The DMA controller – transfers the digital audio data between a user-supplied data buffer and the Synchronous Serial Interface (SSI) FIFO while minimizing any additional CPU overhead. The ALSA sound driver internally allocates and manages the DMA channels, as well as handling all DMA-related interrupt events.
- The SSI controller – transmits and receives digital audio data in conjunction with the PMIC. The SSI can be configured to operate in master mode, and the PMIC in slave mode, or vice versa. The difference is that the master device generates appropriate clock signals to control the flow of data. Using the PMIC in master mode and the SSI in slave mode is recommended, because then the PMIC can generate the necessary clock signals using its own on-board clock sources, without any dependencies or concerns about possible side-effects on other components that may be sharing the same clock signal. Also PMIC generated clocks are more precise.

At least one SPI interface provides the ARM core with read/write access to the PMIC's control registers.

In terms of the PMIC, the following audio-related components are configured and used by the OSS sound driver:

- The Voice CODEC – Provides both playback and recording capabilities.
- The Stereo DAC – Provides a playback capability.

Various output devices and phantom ground circuits can be used to select and configure appropriate output path. Various input devices and microphone bias circuits can be used to select and configure appropriate input path.

The on-board PLL and clock source are used to generate appropriate clock signals for the SSI bus when the PMIC is configured in master mode.

Gain settings for voice codec and stereo DAC:

- Balance gain to be applied to L and R channels
- Mono adder configuration to keep L and R separate or added or phase inverted with respect to one another

For the hardware connection on i.MX31 3-Stack board, only one SSI bus is connected to PMIC audio hardware. So Voice CODEC and Stereo DAC share one SSI bus. Thus audio recording and playback can not be executed simultaneously.

22.3 Software Operation

In brief, the software performs the following steps:

22.3.1 Initialization

- Allocate sound card instance
- Create two PCM devices to support playback on ST-DAC (stereo DAC), playback on voice codec and recording on voice codec
- Pre allocate buffers for PCM components and set playback and capture operations as applicable
- Initialize the control components
- Enable clocks and power management functions
- Finally, register the sound card with all added components with ALSA driver. At this point access to all device files is enabled

ALSA middle layer expects the following ops (something like Linux fops) to be implemented by the audio chip abstraction layer

- Open (Opens a substream for playback or recording. Here generally the low level hardware devices are also opened. ALSA also assigns a substream for the required operation at this stage)
- Close
- IOCTL
- Hardware params (Typically audio hardware configuration in terms of DMA is done over here)
- Hardware free
- Prepare (The low level audio chip, such as SSI and DAM are configured and made ready for playback or recording)
- Trigger (The operation is started for the first time over here. If the driver supports pause/resume operation, it is implemented as part of this function)
- Pointer (This function is expected to return the current position of the DMA pointer)

22.3.2 Device Open

- ALSA allocates a free substream for the operation to be performed
- Open the low level hardware device
- Assign the hardware capabilities to ALSA runtime information. (Runtime structure contains all the hardware, DMA, software capabilities of an opened substream)
- Configure DMA read or write channel for operation
- Configure SSI and DAM hardware
- Configure PMIC audio hardware
- Trigger the transfer

After triggering for the first time, the subsequent DMA reads and writes are configured by the DMA callback.

22.3.3 Digital Mixing

Digital Mixing involves mixing the two streams by configuring SSI to use two channel mode so that data is transmitted alternately from FIFO 0 and FIFO 1. One stream is written to TXFIFO0 and other to TXFIFO1. So the two streams can be mixed as the SSI TX fetches data alternately from FIFO 0 and FIFO 1 in two channel mode. This is routed to VCODEC for playback.

22.4 Source Code Structure

Table 22-1 shows the PMIC-independent source files that are used to build the ALSA sound driver. In addition, these source files define the audio features, capabilities, and sound card interface that is to be supported by the underlying audio hardware. A particular sound card need not support all of the features and capabilities that are defined and there are means available to query the underlying sound card driver for exactly what is supported. All of the source files listed in Table 22-1 are available in the `<ltib_dir>/rpm/BUILD/linux-2.6.26/sound/arm` directory in the Linux kernel source tree.

Table 22-1. PMIC Independent Source Files

File	Description
`mxc-alsa-pmic.c`	Main file that abstracts PMIC audio hardware from ALSA and implements all ALSA callback functions
`mxc-alsa-mixer.c`	Implements and manages control components
`mxc-alsa-common.h`	Common APIs and enums used between `mxc-alsa-pmic.c` and `mxc-alsa-mixer.c`
`mxc-alsa-pmic.h`	Header File

Chapter 23
Digital Audio Multiplexer (AUDMUX) Driver

The digital audio multiplexer (AUDMUX) driver provides multiple and simultaneous interfaces between internal/external ports and peripherals. With AUDMUX, resources do not need to be hard-wired and can be effectively shared in different configurations. The AUDMUX interconnections allow multiple, simultaneous audio/voice/data flows between the ports in point-to-point or point-to-multipoint configurations.

AUDMUX includes two types of interfaces. Internal ports connect to the processor serial interfaces and external ports connect to off-chip audio devices and serial interfaces of other processors. A desired connectivity is achieved by configuring the appropriate internal and external ports.

23.1 Hardware Operation

The Digital Audio Multiplexer (AUDMUX) Driver module configures and deals with the hardware registers for the AUDMUX module.

- At most three internal ports
- Four external ports
- Full 6-wire SSI interfaces for asynchronous receive and transmit
- Configurable 4-wire (synchronous) or 6-wire (asynchronous) peripheral interfaces
- Independent Tx/Rx Frame sync and clock direction selection for host or peripheral
- Each host interface can be connected to any other host or peripheral interface in a point-to-point or point-to-multipoint (network mode)
- Transmit and Receive Data switching to support external network mode
- CE Bus network mode to provide synchronous switching on RxD

For more information, see the chapter on Audio Multiplexer in the documentation for the multimedia applications processor.

23.2 Software Operation

The AUDMUX driver is a hardware abstraction located between its client (the audio driver) and the multimedia applications processor registers. The purpose of this low level API is only to set and read registers. Figure 23-1 shows the block diagram for AUDMUX driver interactions.

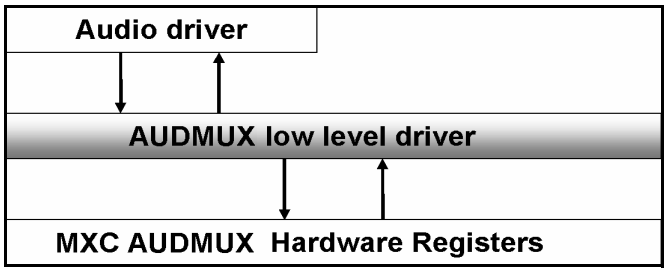


Figure 23-1. AUDMUX Driver Interactions

23.3 Requirements

The AUDMUX module's implementation meets the following requirements:

- The AUDMUX module implements each of the functions required by such a module to interface to Linux and configure all hardware registers related to this module.

23.4 Source Code Structure

Table 23-1 lists the source files available in the device directory:

`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/dam`.

Table 23-1. AUDMUX Source Files

File	Description
`dam.h`	Header file providing external API
`dam.c`	AUDMUX version 2 registers access implementation
`dam_v1.c`	AUDMUX version 1 registers access implementation

23.4.1 Linux Menu Configuration Options

The Linux kernel configuration option, CONFIG_MXC_DAM is provided for this module. In order to get to the dam configuration, use the command `./ltib -c` when located in the <ltib dir>. In the screen, select **Configure Kernel**, exit, and a new screen will appear.

This configuration option is for the Digital Audio Multiplexer (AUDMUX) Driver. In `menuconfig`, this option is available under Device Drivers > MXC support drivers > MXC Digital Audio Multiplexer support > DAM support.

23.5 Programming Interface (Exported API)

The AUDMUX exported API allows the user to process standard AUDMUX operations.

Table 23-2. AUDMUX Exported Functions

Function	Description
`dam_select_mode()`	This function selects the operation mode of the port.
`dam_select_RxClk_direction()`	This function controls Receive clock signal direction for the port.
`dam_select_RxClk_source()`	This function controls Receive clock signal source for the port.
`dam_select_RxD_source()`	This function selects the source port for the RxD data.
`dam_select_RxFS_direction()`	This function controls Receive Frame Sync signal direction for the port.
`dam_select_RxFS_source()`	This function controls Receive Frame Sync signal source for the port.
`dam_select_TxClk_direction()`	This function controls Transmit clock signal direction for the port.
`dam_select_TxClk_source()`	This function controls Transmit clock signal source for the port.
`dam_select_TxFS_direction()`	This function controls Transmit Frame Sync signal direction for the port.
`dam_select_TxFS_source()`	This function controls Transmit Frame Sync signal source for the port.
`dam_set_internal_network_mode_mask ()`	This function sets a bit mask that selects the port from which of the RxD signals are to be ANDed together for internal network mode. Bit 6 represents RxD from Port7 and bit0 represents RxD from Port1. 1 excludes RxDn from ANDing. 0 includes RxDn for ANDing.
`dam_set_synchronous()`	This function controls whether or not the port is in synchronous mode. When the synchronous mode is selected, the receive and the transmit sections use common clock and frame sync signals. When the synchronous mode is not selected, separate clock and frame sync signals are used for the transmit and the receive sections. The default value is the synchronous mode selected.
`dam_switch_Tx_Rx()`	This function swaps transmit and receive signals from (Da-TxD, Db-RxD) to (Da-RxD, Db-TxD). This default signal configuration is Da-TxD, Db-RxD.
`dam_reset_register()`	This function resets the two registers of the selected port.

The exact description of this API is available in the generated doxygen `api_output` directory. The whole API documentation, including internal functions, is available in the generated doxygen `full_output` directory.

23.6 Interrupt Requirements

No interrupts are generated by the digital audio multiplexer.

Chapter 24
Synchronous Serial Interface (SSI) Driver

The synchronous serial interface (SSI) driver manages a full-duplex serial port that allows the multimedia applications processor to communicate with a variety of serial devices. These serial devices can be standard CODECs, digital signal processors (DSPs), microprocessors, peripherals, and popular industry audio codecs that implement the inter-IC sound bus standard (I2S) and Intel AC97 standard.

The SSI is typically used to transfer samples in a periodic manner. The SSI consists of independent transmitter and receiver sections with independent clock generation and frame synchronization. It supports the configuration of all SSI block registers.

24.1 Hardware Operation

SSI includes the following features:

- Independent (asynchronous) or shared (synchronous) transmit and receive sections with separate or shared internal/external clocks and frame syncs, operating in Master or Slave mode.
- Normal mode operation using frame sync.
- Network mode operation allowing multiple devices to share the port with as many as thirty-two time slots.
- Gated Clock mode operation requiring no frame sync.
- Two sets of Transmit and Receive FIFOs. Each of the four FIFOs is 8x24 bits. The two sets of Tx/Rx FIFOs can be used in Network mode to provide two independent channels for transmission and reception.
- Programmable data interface modes such as I2S, LSB, MSB aligned.
- Programmable word length (8, 10, 12, 16, 18, 20, 22, or 24 bits).
- Program options for frame sync and clock generation.
- Programmable I2S modes (Master, Slave or Normal). Oversampling clock `ccm_ssi_clk` available as output from SRCK in I2S Master mode.
- AC97 support.
- Completely separate clock and frame sync selections for the receive and transmit sections. In the AC97 standard, the clock is taken from an external source and frame sync is generated internally.
- External `ccm_ssi_clk` input for use in I2S Master mode. Programmable oversampling clock (`SYS_CLK/ccm_ssi_clk`) of the sampling frequency available as output in master mode at SRCK, when operated in sync mode.
- Programmable internal clock divider.
- Time Slot Mask Registers for reduced CPU overhead for both Tx and Rx.

- SSI power-down feature.
- Programmable wait states for CPU accesses.
- IP Interface for register accesses, compliant to SRS 3.0.2 standard.

For more information, see the chapter on SSI in the multimedia applications processor documentation.

24.2 Software Operation

The SSI driver is a hardware abstraction located between its client (Audio driver) and the multimedia applications processor registers.

The purpose of this low level API is only to set and read registers. Figure 24-1 shows a block diagram of the software interaction.

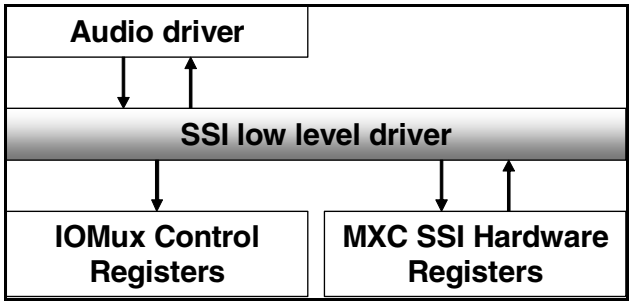


Figure 24-1. SSI Driver Interactions

24.3 Requirements

The SSI module implements each of the functions required by an SSI module to interface to Linux and configure all hardware registers related to this module.

24.4 Source Code Structure

Table 24-1 lists the source files available in the device's directory:

`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/ssi`.

Table 24-1. SSI Source File List

File	Description
`registers.h`	MXC registers definition header file
`ssi_types.h`	Header file providing SSI specific types
`ssi.h`	Header file providing external API
`ssi.c`	SSI registers access implementation

24.4.1 Linux Menu Configuration Options

The Linux kernel configuration option, CONFIG_MXC_SSI, is provided for this module. In order to get to the ssi configuration, use the command `./ltib -c` when located in the <ltib dir>. In the screen, select Configure Kernel, exit, and a new screen will appear.

This configuration option is for the multimedia applications processor SSI driver used for the MXC SSI ports. In `menuconfig`, this option is available under Device Drivers > MXC support drivers > MXC SSI support > SSI support.

24.5 Programming Interface (Exported API)

The SSI Exported API allows the user to process standard SSI operations. The exact description of this API is available in the generated doxygen `api_output` directory. The whole API documentation, including internal functions, is available in the generated doxygen `full_output` directory.

Table 24-2. SSI Exported Functions

Function	Description
`ssi_ac97_frame_rate_divider()`	This function controls the AC97 frame rate divider.
`ssi_ac97_get_command_address_register()`	This function gets the AC97 command address register.
`ssi_ac97_get_command_data_register()`	This function gets the AC97 command data register.
`ssi_ac97_get_tag_register()`	This function gets the AC97 tag register.
`ssi_ac97_mode_enable()`	This function controls the AC97 mode.
`ssi_ac97_tag_in_fifo()`	This function controls the AC97 tag in FIFO behavior.
`ssi_ac97_read_command()`	This function controls the AC97 read command.
`ssi_ac97_set_command_address_register()`	This function sets the AC97 command address register.
`ssi_ac97_set_command_data_register()`	This function sets the AC97 command data register.
`ssi_ac97_set_tag_register()`	This function sets the AC97 tag register.
`ssi_ac97_variable_mode ()`	This function controls the AC97 variable mode.
`ssi_ac97_write_command()`	This function controls the AC97 write command.
`ssi_clock_idle_state()`	This function controls the idle state of the transmit clock port during SSI internal gated mode.
`ssi_clock_off()`	This function turns off/on the ccm_ssi_clk to reduce power consumption.
`ssi_enable()`	This function enables/disables the SSI module.
`ssi_get_data()`	This function gets the data word in the Receive FIFO of the SSI module.
`ssi_get_status()`	This function returns the status of the SSI module (SISR register) as a combination of status.
`ssi_i2s_mode()`	This function selects the I2S mode of the SSI module.

Table 24-2. SSI Exported Functions (Continued)

Function	Description
ssi_interrupt_disable()	This function disables the interrupts of the SSI module.
ssi_interrupt_enable()	This function enables the interrupts of the SSI module.
ssi_network_mode()	This function enables/disables the network mode of the SSI module.
ssi_receive_enable()	This function enables/disables the receive section of the SSI module.
ssi_rx_bit0()	This function configures the SSI module to receive data word at bit position 0 or 23 in the Receive shift register.
ssi_rx_clock_direction()	This function controls the source of the clock signal used to clock the Receive shift register.
ssi_rx_clock_divide_by_two()	This function configures the divide-by-two divider of the SSI module for the receive section.
ssi_rx_clock_polarity()	This function controls which bit clock edge is used to clock in data.
ssi_rx_clock_prescaler()	This function configures a fixed divide-by-eight clock pre-scaler divider of the SSI module in series with the variable pre-scaler for the receive section.
ssi_rx_early_frame_sync()	This function controls the early frame sync configuration.
ssi_rx_fifo_counter()	This function gets the number of data words in the Receive FIFO.
ssi_rx_fifo_enable()	This function enables the Receive FIFO.
ssi_rx_fifo_full_watermark()	This function controls the threshold at which the RFFx flag will be set.
ssi_rx_flush_fifo()	This function flushes the Receive FIFOs.
ssi_rx_frame_direction()	This function controls the direction of the Frame Sync signal for the receive section.
ssi_rx_frame_rate()	This function configures the Receive frame rate divider for the receive section.
ssi_rx_frame_sync_active()	This function controls the Frame Sync active polarity for the receive section.
ssi_rx_frame_sync_length()	This function controls the Frame Sync length (one word or one bit long) for the receive section.
ssi_rx_mask_time_slot()	This function configures the time slot(s) to mask for the receive section.
ssi_rx_prescaler_modulus()	This function configures the prescale divider for the receive section.

Table 24-2. SSI Exported Functions (Continued)

Function	Description
ssi_rx_shift_direction()	This function controls whether the MSB or LSB will be received first in a sample.
ssi_rx_word_length()	This function configures the Receive word length.
ssi_set_data()	This function sets the data word in the Transmit FIFO of the SSI module.
ssi_set_wait_states()	This function controls the number of wait states between the core and SSI.
ssi_synchronous_mode()	This function enables/disables the synchronous mode of the SSI module.
ssi_system_clock()	This function allows the SSI module to output the SYS_CLK at the SRCK port.
ssi_transmit_enable()	This function enables/disables the transmit section of the SSI module.
ssi_two_channel_mode()	This function allows the SSI module to operate in the two channel mode.
ssi_tx_bit0()	This function configures the SSI module to transmit data word from bit position 0 or 23 in the Transmit shift register.
ssi_tx_clock_direction()	This function controls the direction of the clock signal used to clock the Transmit shift register.
ssi_tx_clock_divide_by_two()	This function configures the divide-by-two divider of the SSI module for the transmit section.
ssi_tx_clock_polarity()	This function controls which bit clock edge is used to clock out data.
ssi_tx_clock_prescaler()	This function configures a fixed divide-by-eight clock prescaler divider of the SSI module in series with the variable prescaler for the transmit section.
ssi_tx_early_frame_sync()	This function controls the early frame sync configuration for the transmit section.
ssi_tx_fifo_counter()	This function gets the number of data words in the Transmit FIFO.
ssi_tx_fifo_empty_watermark()	This function controls the threshold at which the TFEx flag will be set.
ssi_tx_fifo_enable()	This function enables the Transmit FIFO.
ssi_tx_flush_fifo()	This function flushes the Transmit FIFOs.
ssi_tx_frame_direction()	This function controls the direction of the Frame Sync signal for the transmit section.
ssi_tx_frame_rate()	This function configures the Transmit frame rate divider.

Table 24-2. SSI Exported Functions (Continued)

Function	Description
`ssi_tx_frame_sync_active()`	This function controls the Frame Sync active polarity for the transmit section.
`ssi_tx_frame_sync_length()`	This function controls the Frame Sync length (one word or one bit long) for the transmit section.
`ssi_tx_mask_time_slot()`	This function configures the time slot(s) to mask for the transmit section.
`ssi_tx_prescaler_modulus()`	This function configures the prescale divider for the transmit section.
`ssi_tx_shift_direction()`	This function controls whether the MSB or LSB will be transmitted first in a sample.
`ssi_tx_word_length()`	This function configures the Transmit word length.

24.6 Interrupt Requirements

The SSI module generates interrupts but this driver is only a hardware abstraction. The interrupt requirements depend on the client which will use the API.

Chapter 25
NAND Flash Memory Technology Device (MTD) Driver

25.1 Overview

The NAND Flash Memory Technology Device (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 taken care of by the generic layer provided by the Linux MTD subsystem for NAND devices.

25.1.1 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 can't be executed from there. 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. For details, see Section 25.6, “Device-Specific Information.”

25.1.2 Software Operation

The MTDs in Linux cover 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 filesystem 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/.

25.2 Requirements

This NAND Flash MTD driver implementation should meet the following requirements:

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

25.3 Source Code Structure

Table 25-1 lists the source files available for the NAND MTD driver. These files are under the `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mtd/nand` directory.

Table 25-1. NAND MTD File List

File	Description
`mxc_nd.c`	Hardware-specific layer for NAND MTD driver for NFC version 1.
`mxc_nd.h`	Register declaration for NFC version 1
`mxc_nd2.c`	Hardware-specific layer for NAND MTD driver for NFC version 2 and above
`mxc_nd2.h`	Register declaration for NFC version 2 and above

25.4 Configuration

The NAND MTD driver has the following Linux menu configuration options.

25.4.1 Linux Menu Configuration Options

In the `menuconfig` the following options are available under `Memory Technology Device (MTD) support -> NAND Device Support -> MXC NAND Support`:

- CONFIG_MTD_NAND_MXC - This is the configuration option for the NAND MTD driver for the i.MX processors having NFC hardware version 1.

- CONFIG_MTD_NAND_MXC_V2 - This is the configuration option for the NAND MTD driver for the i.MX processors having NFC hardware version 2.

25.5 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.

25.6 Device-Specific Information

For more information on NFC hardware, refer to the L3 specifications of i.MX processors.Table 25-2 lists the NFC hardware version on different i.MX platforms.

Table 25-2. NFC Hardware Version across i.MX platforms

NFC Version	Platforms/SoC
1	i.MX31

Chapter 26
Low-Level Keypad Driver

The low-level keypad driver interfaces with the keypad port hardware (KPP) in the i.MX application processors. The keypad driver is implemented as a standard Linux 2.6 keyboard driver, modified for the i.MX application processors.

The keypad driver supports the following features:

- Interrupt-driven scan code generation for keypress and release on a keypad matrix.
- The keypad is supported 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 the other input devices are loaded or not.

26.1 Hardware Operation

The i.MX application processors keypad device supports a keypad matrix with as many as 8 rows and 8 columns. Any pins that are not being used for the keypad are available as general purpose input/output pins.

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 key press event, but in Low power mode it is done even when there is no MCU clock.

26.2 Software Operation

The keypad driver generates scan-codes for keypress and release 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 keypresses and releases.
3. The keypad scan timer function is called every 10ms to scan for any keypress or release on the keypad.
4. The scancode 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 which is shown in Figure 26-1. The `mxc_kpp_scan_matrix` function is called for every keypress and release interrupt.

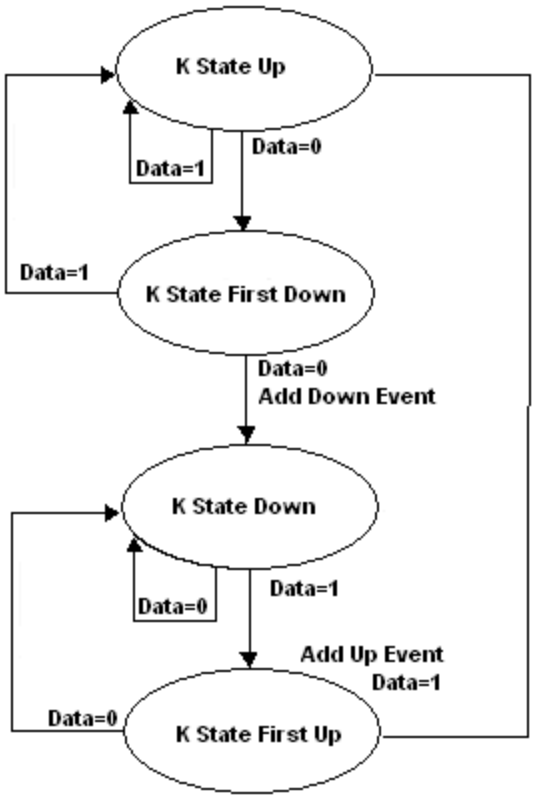


Figure 26-1. Keypad Driver State Machine

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

The keypress keycodes 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-2.6.26/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 and non-blocking reads and also `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 happened,
- ‘code’ i.MX keycode for keypress or release,
- ‘value’ equals ‘0’ for key release and ‘1’ for key press.

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;
```

Refer to the Device-Specific Information section Table 26-3 for mapcodes and scancodes.

26.3 Reassigning Keycodes

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

26.4 Requirements

The keypad driver meets the following requirements:

- Rreturns the input keycode for every key that is pressed or released.
- Implements support for an interrupt driver for keypress or release.
- Implements support for blocking and non-blocking reads.
- Is implemented as a standard input device.

26.5 Source Code Structure

The source files are available in the `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/input/keyboard` directory, the `<ltib_dir>/rpm/BUILD/linux-2.6.26/include/linux` directory, and the `<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx*` directories where * is the platform desired. It is 3 for i.MX31.

The source code is comprised of the following files:

Table 26-1. Source Code

File	Description
drivers/input/keyboard/mxc_keyb.c	low-level driver implementation
drivers/input/keyboard/mxc_keyb.h	driver structures, control register address definitions
include/linux/input.h	generic Linux keycode definitions
arch/arm/mach-mx3/mx3_3stack.c	contains the platform-specific keymapping[] keycode array

26.6 Driver Configuration

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 Keypad port (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 8 rows and 8 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 26-1).
- Debounce delay: The user can configure the debounce delay by changing the variable `KScanRate` defined in `mxc_keyb.c`

26.7 Programming Interface

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

26.8 Interrupt Requirements

Table 26-2 lists the keypad interrupt timer requirements.

Table 26-2. 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 msec	10 msec

26.9 Device-Specific Information

Table 26-3 shows key connections, key scancodes, and key mapcodes of the keys on the keypad for a specific platform.

Table 26-3. Key Connections for Keypad

Key	Row	Column	Scancode	Linux Key Code	Platform
UP	0	0	0	KEY_UP	i.MX31 3-stack
DOWN	0	1	1	KEY_DOWN	i.MX31 3-stack
RIGHT	1	0	8	KEY_RIGHT	i.MX31 3-stack
LEFT	1	1	9	KEY_LEFT	i.MX31 3-stack
ENTER	1	2	10	KEY_ENTER	i.MX31 3-stack
MENU1	2	0	16	KEY_F6 (APP1)	i.MX31 3-stack
MENU2	2	1	17	KEY_F8 (APP2)	i.MX31 3-stack
MENU3	2	2	18	KEY_F9 (APP3)	i.MX31 3-stack
MENU4	2	3	19	KEY_F10 (APP4)	i.MX31 3-stack

Chapter 27
SMSC LAN9217 Ethernet Driver

The SMSC LAN9217 Ethernet driver interfaces SMSC LAN9217-specific functions with the standard Linux kernel network module. The LAN9217 is a full-featured, single-chip 10/100 Ethernet controller designed for embedded applications where performance, flexibility, ease of integration, and system cost control are required. The LAN9217 has been specifically architected to provide the highest performance possible for any 16-bit application. The LAN9217 is fully IEEE 802.3 10BASE-T and 802.3 100BASE-TX compliant, and supports HP Auto-MDIX.

The SMSC LAN9217 Ethernet driver has the following features:

- Efficient PacketPage Architecture can operate in I/O and memory space, and as a DMA slave.
- Supports full duplex operation.
- Supports on-chip RAM buffers for transmission and reception of frames.
- Supports programmable transmit features like automatic retransmission on collision and automatic CRC generation.
- EEPROM support for configuration.
- Supports MAC address setting.
- Supports obtaining statistics from the device, such as transmit collisions.

This network adapter can be accessed through the `ifconfig` command with interface name (`eth0`). The probe function of this driver is declared in `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/net/Space.c` to probe for the device and to initialize it during boot.

27.1 Hardware Operation

The SMSC LAN9217 Ethernet controller interfaces the system to the LAN network.

A brief overview of the device functionality is provided here. For details, see *LAN9217 Ethernet Controller Data Sheet*.

The LAN9217 includes an integrated Ethernet MAC and PHY with a high-performance SRAM-like slave interface. The simple, yet highly functional host bus interface provides glue-less connection to most common 16-bit microprocessors and microcontrollers as well as 32-bit microprocessors with a 16-bit external bus. The LAN9217 includes large transmit and receive data FIFOs to accommodate high latency applications. In addition, the LAN9217 memory buffer architecture allows the most efficient use of memory resources by optimizing packet granularity.

27.2 Software Operation

The SMSC LAN9217 Ethernet Driver has the functions listed below.

- Module initialization—Initializes the module with the device specific structure
- Driver entry points—Provides standard entry points for transmission
- Interrupt servicing routine
- Miscellaneous routines—Setting and programming MAC address

27.3 Requirements

The Ethernet driver meets the following requirements:

- The module provides all the entry points to interface with the Linux kernel 2.6 net module.
- This Ethernet driver implements the default data configuration function to set the MAC address and interface media used in case of EEPROM failure.
- This module follows Linux kernel coding style by Linus Torvalds. This is included in Linux distributions as the file Documentation/CodingStyle.

27.4 Source Code Structure

Table 27-1 lists the source files available in the `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/net` directory:

Table 27-1. Ethernet File List

File	Description
`smsc911x.h`	Header file defining registers.
`smsc911x.c`	Linux driver for Ethernet LAN controller.

27.5 Linux Menu Configuration Options

The Linux kernel configuration option, CONFIG_SMSC911X, is provided for this module. This is the Ethernet driver used for the SMSC LAN9117 chip. In `menuconfig`, this option is located under `Device Drivers -> Network Device Support -> Ethernet 10 or 100 Mbit -> SMSC LAN911x/LAN921x families embedded ethernet support`.

Chapter 28
WLAN Driver

The APM6628 is a full-featured Wi-Fi 802.11b/g and Bluetooth v2.0+EDR combo module that simultaneously provides Wi-Fi and Bluetooth connections.

The WLAN driver is used to drive the APM6628 module to implement Wi-Fi functionality. The APM6628 module adopts the CSR UniFi V5 solution. The UniFi chip is connected to SDHC2 controller of i.MX31.

The UniFi driver implements the Wi-Fi module of the APM6628. This driver provides access to a network using an access point (AP), which is a standard Ethernet interface in Linux. The user level tools communicate with the UniFi driver using the Linux Wireless Extension. The user can use the Linux Wireless Tools (WT) to configure Wi-Fi.

28.1 Hardware Operation

The APM6628 provides the Wi-Fi functionality needed to interface the system to a LAN network. In the i.MX31 3-Stack board, it is interfaced to the i.MX application processor through SDHC controller. The SDHC port is SDHC2.

28.1.1 Register Access

The APM6628 accesses its registers using two methods: SDIO and SPI. The i.MX boards use SDIO to access the APM6628 with the CMD52 command channel and the CMD53 data channel as follows:

- For APM6628 register access, the driver uses the CMD52 access to each of the on-chip registers and memory locations directly.
- For APM6628 data access, the driver uses the CMD53 to transfer blocks of data directly to or from the on-chip MMU buffers.

28.1.2 Transmission

The driver uses the CMD52/53 to transfer packets to the device.

28.1.3 Reception

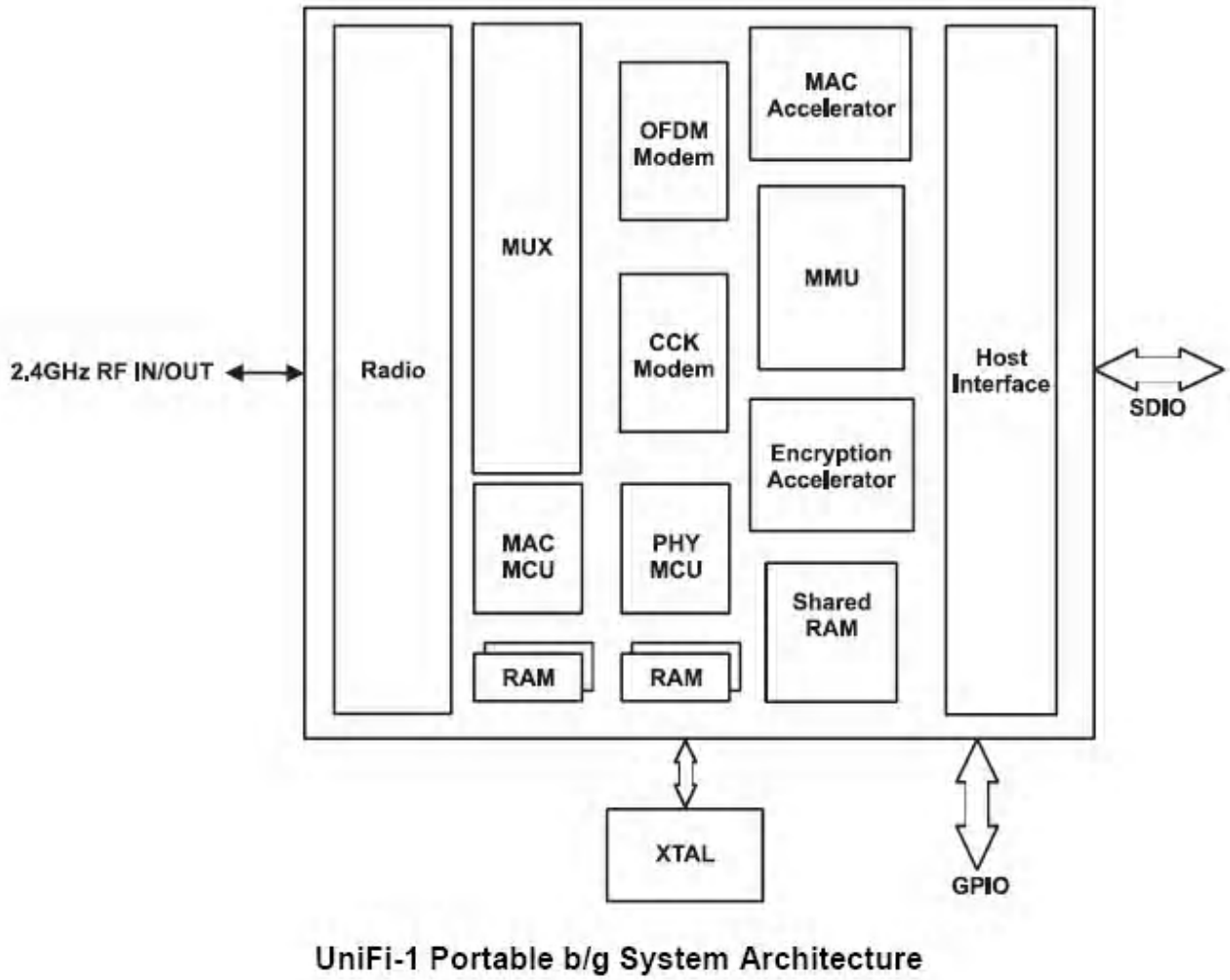
The driver uses the CMD52/53 to receive packets from the device.

28.1.4 Encryption and Decryption

There are two phases of encryption or decryption. In the first phase, the driver uses CMD52/53 to transmit original data to the device. In the second phase, the driver uses CMD52/53 to read back the data which has been transmitted.

Figure 28-1 illustrates system architecture.

Figure 28-1. Wi-Fi System Architecture



28.1.5 Conflicts with other Peripherals

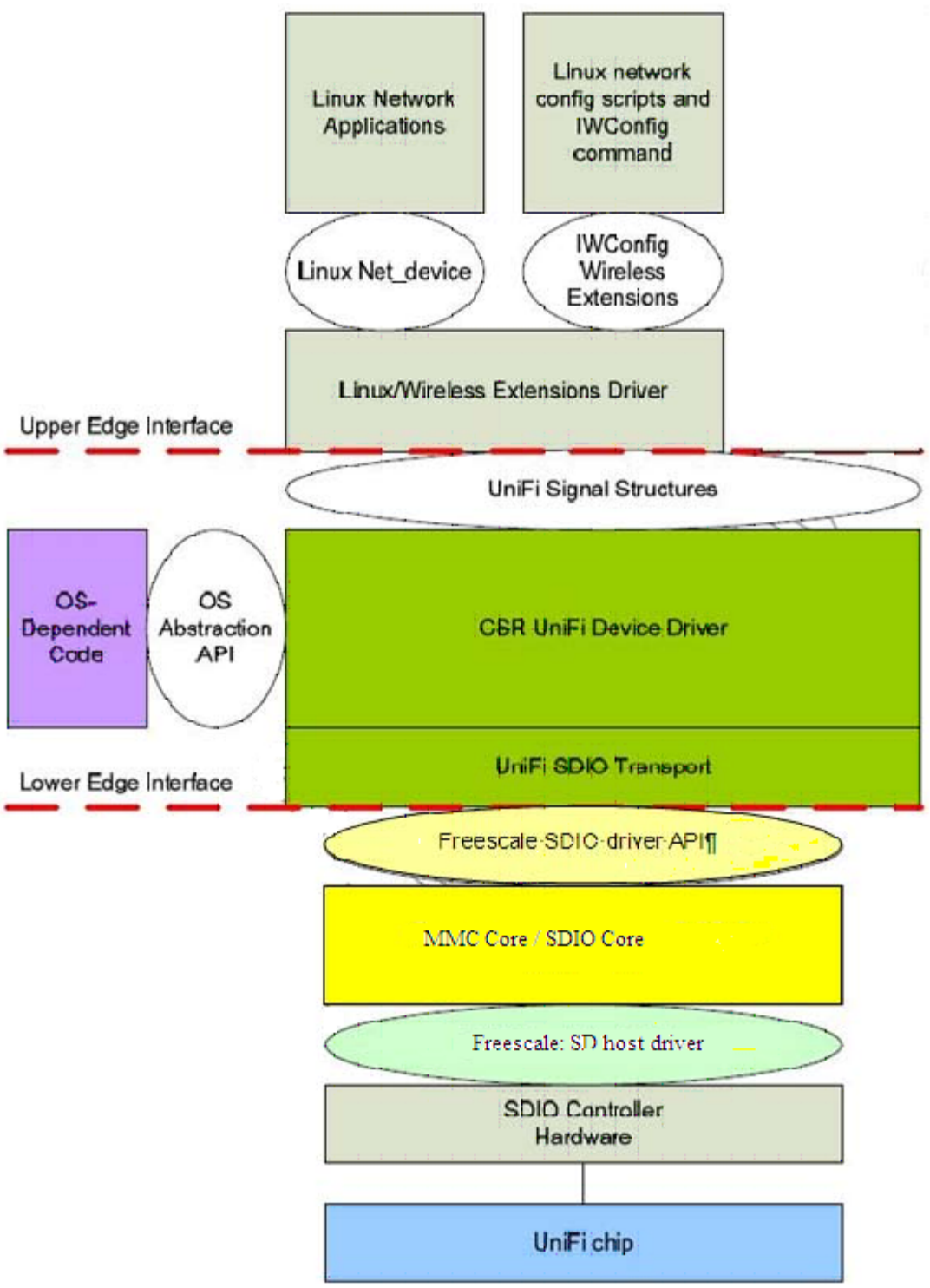
Wi-Fi shares one reset pin with the Bluetooth module in the i.MX31 3-Stack board. Normally, the Wi-Fi module starts prior to the Bluetooth module, and resets the entire APM6628 chip upon startup.

28.2 Software Operation

The CSR UniFi driver implements the Wi-Fi module of the APM6628. This driver provides access to a network using an access point (AP), which simulates a standard Ethernet interface in Linux. The software

operates the Wi-Fi device using Linux Wireless Extension, and configures Wi-Fi using Linux Wireless Tools (WT). Figure 28-2 illustrates the software architecture.

Figure 28-2. Software Architecture



The main component is the miniport driver. The driver provides the means to configure the UniFi device for connecting to a wireless network to send and receive data. A suitable wireless client, such as Microsoft Wireless Zero Configuration can:

- Actively scan for wireless networks in the local area.
- Connect to an unsecured or WEP-enabled infrastructure or ad-hoc network.
- Connect to WPA-enabled networks using pre-shared key (PSK).
- Start an unsecured or WEP-enabled ad-hoc network (IBSS).

The device driver conforms to the following:

- The NDIS 5.1 specification (defined by the IEEE 802.11 Network Adapter Design Guidelines for Windows XP) for integration into the Windows operating system.
- The UniFi Host Interface Protocol Specification for the exchange of signal primitives with the UniFi WLAN card.
- Network connections are set up using a wireless LAN client. The client issues a set of NDIS defined 802.11 OIDs to the miniport driver so that it can configure the UniFi device appropriately.

28.3 Configuration

There are two configurations: kernel configuration and WPA configuration.

28.3.1 Linux Configuration

The following Linux kernel configuration options are provided for the driver:

- Networking > Wireless > Generic IEEE 802.11 Networking Stack
- Networking > Wireless > Wireless extensions

Select these options for wireless support. By default, these options are enabled for all architectures.

28.3.2 WPA Configuration

The following configurations are provided for the WPA supplicant:

```
CONFIG_WIRELESS_EXTENSION=y
CONFIG_IEEE8021X_EAPOL=y
CONFIG_EAP_PSK=y
CONFIG_CTRL_IFACE=y
CONFIG_L2_PACKET=linux
```

28.4 Programming Interface

The Freescale SDIO structure exports an interface to get the `mmc_host` structure pointer to the UniFi driver. The UniFi driver uses this structure to issue the process of discovery of the SDIO card, and adds code to enable or disable the power supply.

Chapter 29
Security Drivers

The security drivers provide several APIs that facilitate access to various security features in the processor. The secure controller (SCC) consists of two modules, a secure RAM module and a secure monitor module. The SCC's key encryption module (KEM) has a security feature of storing encrypted data in the on-chip RAM (Red data = plain text, Black data = encrypted), with a total size of 2 kBytes. This module is needed in cases where data must be stored securely in external memory in encrypted form. This module has a feature of clearing the secure RAM during intrusion. The system also includes:

- Random Number Generator (RNGA), which generates random numbers
- Run-Time Integrity Checker (RTIC), which hashes the data during run-time

The security design covers the following modules:

- Boot Security
- SCC (Secure RAM, Secure Monitor)
- Algorithm Integrity Checker
- Security Timer
- Key Encryption Module (KEM), Zeroization module
- RNGA
- RTIC

29.1 Hardware Security Modules

The platform has several different security blocks, and the details of the individual blocks are mentioned in the following sections.

29.1.1 Boot Security

During boot, the boot pins must be set to enable the processor to boot internally. The SCC module must be enabled by blowing specific fuses. By booting in this manner, the data in the Flash (kernel image) can be assured of integrity. Any violation in the data integrity raises an alarm.

29.1.2 SCC-Secure RAM

Figure 29-1 shows the SCC-Secure RAM and its modules. Individual modules are described in the following sections.

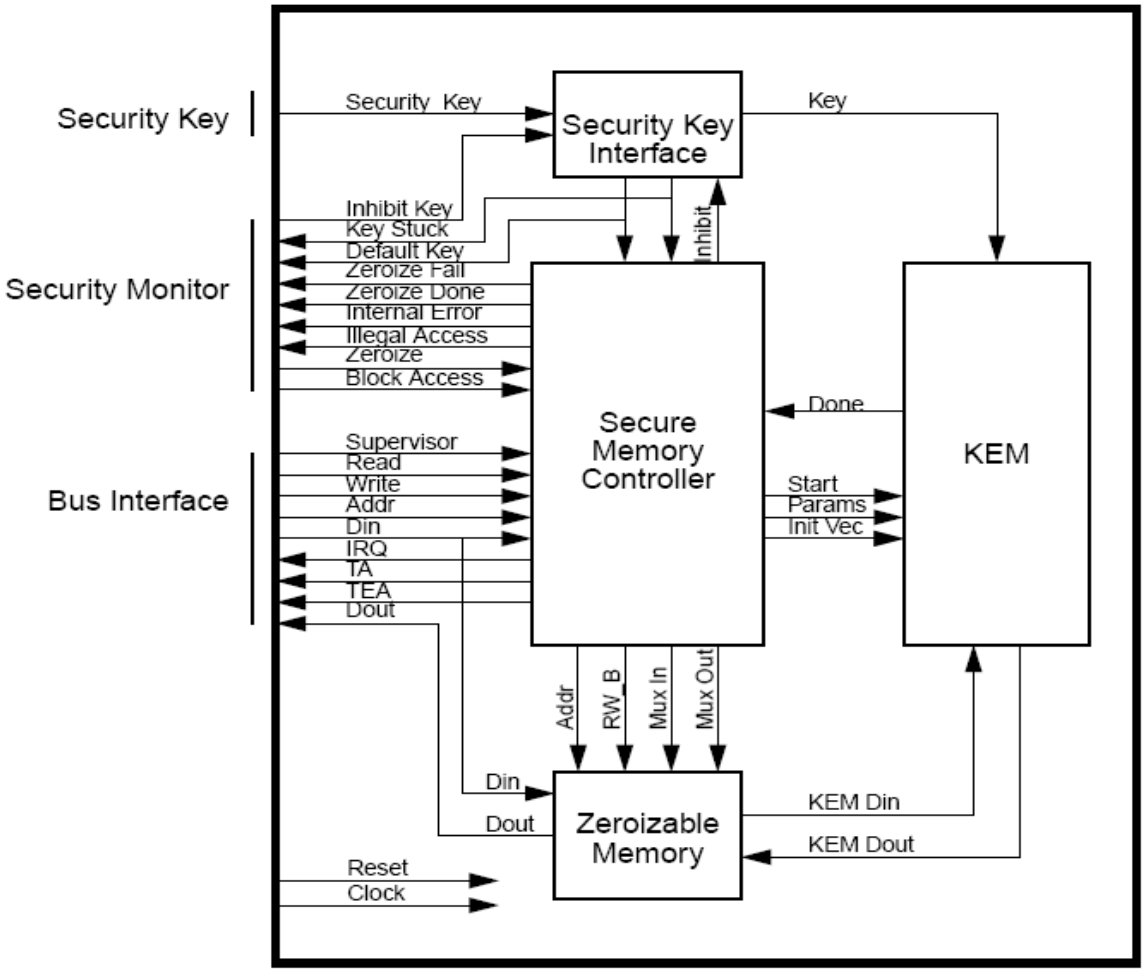


Figure 29-1. Secure RAM Block Diagram

29.1.3 SCC—Key Encryption Module (KEM)

The key encryption module (KEM) uses the 3DES algorithm and a 168-bit key for encryption of data. The key is programmed during manufacture and is accessible only to the encryption module. It is not accessible on any bus external to the secure memory module. The data in the external RAM is stored in an encrypted format. The data is encrypted using 3DES algorithm so that it can be decoded only using the SCC module.

29.1.4 SCC—Zeroizable Memory

The memory module can be multiplexed in and out of the RAM to allow the memory controller to switch paths according to the Secure RAM state and the host read and write accesses. When zeroing sections of memory, only the memory controller has access. When encrypting or decrypting, only the KEM module

has access. When the Secure RAM is in the Idle state, then the host can access the memory. The Zeroize Done signal is also used to reset the encryption module and the memory controller. While the Zeroize Done signal is low, any attempted access by the host is ignored. When the Zeroize signal is asserted, or when the Zeroize Memory bit in the Interrupt Control register is set, not only is the Red and Black memory initialized, but most of the registers are also reset. The Red Start, Black Start, Length, Control, Error Status, Init Vector 0, and Init Vector 1 registers are cleared. The encryption engine is also reset. The Zeroization takes place whenever there is a security violation like external bus intrusion. The Red and Black memory area is usually cleared during system boot-up.

29.1.5 SCC—Security Key Interface Module

The Security Key Interface module uses a 168-bit encryption key. The physical structures for the encryption key reside elsewhere. The Secret Key Interface contains a key mux to select between the encryption key and the default key and test the logic to determine the validity of the encryption key. In the Secure state the encryption key is used. In the Non-Secure state, the default key prevents unauthorized access to SCC-encrypted data and is useful for test purposes.

29.1.6 SCC—Secure Memory Controller

The Secure Memory controller implements an internal data handler that moves data in and out of the KEM, a memory clear function, and all of the supervisor-accessible Control and Status registers.

29.1.7 SCC—Security Monitor

The Security Monitor (SMN) is a critical component of security assurance for the platform. Specifically, it determines when and how Secure RAM resources are available to the system, and it also provides mechanisms for verifying software algorithm integrity. This block ensures that the system is running in such a manner as to provide protection for the sensitive data that is resident in the SCC. The Security Monitor consists of five main sub-blocks: The Secure State Controller, the Security Policy, the Algorithm Integrity Checker (AIC), the Security Timer, and the Debug Detector.

Figure 29-2 shows a block diagram of the SMN.

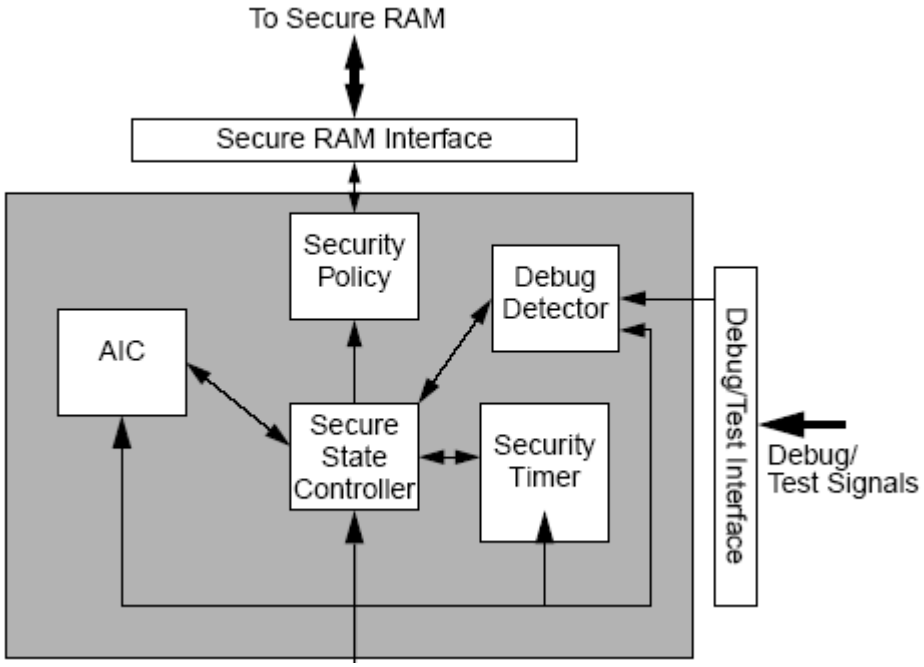


Figure 29-2. Security Monitor Block Diagram

29.1.8 SCC—Secure State Controller

The Secure State Controller, shown in Figure 29-3, is a state machine that controls the security states of the chip.

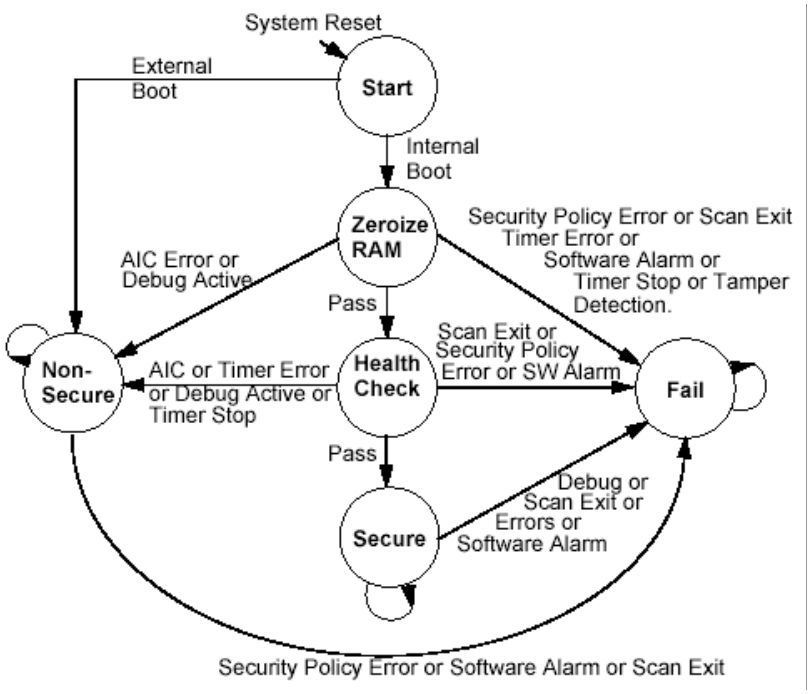


Figure 29-3. Secure State Controller State Diagram

29.1.9 SCC—Security Policy

The Security Policy block uses state information from the Secure State Controller along with inputs from the Secure RAM to determine what access to the Secure RAM is allowed based on the policy table, which is available in the corresponding platform's L3 specification document.

29.1.10 SCC—Algorithm Integrity Checker (AIC)

The Algorithm Integrity Checker (AIC) is used in conjunction with software to provide assurance that critical software (such as a software encryption algorithm) operates correctly. It is also an integral part of the power-up procedure as it must be used to achieve a secure state.

29.1.11 SCC—Secure Timer

The Secure Timer is a 32-bit programmable timer. It is used in conjunction with the Secure State Controller during power-up to ensure that the transition to the Secure state happens in the appropriate amount of time. After power-up, the timer can be used as a watchdog timer for any time-critical routines or algorithms. If the timer is allowed to expire, it generates an error.

29.1.12 SCC—Debug Detector

The debug detector monitors the various debug and test signals and informs the status to the Secure state controller. The secure state controller gets an alert when debug modes, such as JTAG and scan are active. The debug detector status register can be read by the host processor to determine which debug signals are currently active. Refer to the SCC section in the corresponding platform's L3 spec of the hardware documentation for more information on the SCC-Debug Detector.

29.1.13 Random Number Generator Accelerator (RNGA)

The RNGA module is used to generate 32-bit random numbers. This module is designed to comply with the FIPS-140 standards for randomness and non-determinism. The random bits are generated by clocking shift registers with clocks derived from ring oscillators. The configuration of the shift registers ensures statistically good data (that is, data that looks random). The oscillators, with their unknown frequencies, provide the required entropy needed to create random data. There are different modes of operation of the RNGA:

- Normal Mode
- Secure Mode
- Verification Mode
- Oscillator Frequency Test Mode
- Sleep Mode
- Scan Mode

These modes can be achieved by setting appropriate bits in the RNGA set of registers. Secure Mode is functionally equivalent to normal mode. It is provided for applications requiring higher assurance. For details about how to enter the different modes, refer to the RNGA chapter in the IC documentation.

29.1.13.1 Run-Time Integrity Checker (RTIC)

RTIC is used to ensure the integrity of the peripheral memory contents and to assist with boot authentication. This module has the ability to check the memory contents during system boot and during run-time execution. If the memory contents at run-time fail to match the hash signature, an error in the security monitor is triggered. The RTIC communicates over two interfaces: the IP Skyblue (slave) and AHB-Lite (master). The IP-slave interface is used to read/write to the RTIC address space. The RTIC contains a DMA controller to perform reads of the peripheral memory block(s) on the AHB bus. For more information about the RTIC, refer to the RTIC chapter in the appropriate platform's L3 spec.

29.2 Software Security Modules

Besides the hardware security modules, there is optional, specialized software that helps to deliver security. This includes the RNGA (Random Number Generator) and the RTIC (Run-Time Integrity Checker) modules.

29.2.1 SCC Common Software Operations

The SCC driver is only available to other kernel modules. That is, there is no node file in `/dev`. Thus, it is not possible for a user-mode program to access the driver, and it is not possible for a user program to access the device directly.

With the exception of scc_monitor_security_failure(), all routines are synchronous, which means they will not return to their caller until the requested action completes, or fails to complete. Some of these functions could take some time to perform, depending upon the request.

Routines are provided to:

- Encrypt or decrypt secrets—`scc_crypt()`
- Trigger a security-violation alarm—`scc_set_sw_alarm()`
- Get configuration and version information—`scc_get_configuration()`
- Zero areas of memory—`scc_zeroize_memories()`
- Work on wrapped and stored secret values—`scc_alloc_slot()`, `scc_dealloc_slot()`, `scc_load_slot()`, `scc_decrypt_slot()`, `scc_encrypt_slot()`, and `scc_get_slot_info()`
- Monitor the Security Failure alarm—`scc_monitor_security_failure()`
- Stop monitoring Security Failure alarm—`scc_stop_monitoring_security_failure()`
- Write registers of the SCC—`scc_write_register()`
- Read registers of the SCC—`scc_read_register()`

The driver does not allow storage of data in either the Red or Black memories. Any decrypted information is returned to the user. If the user wants to use the information at a later point, the encrypted form must again be passed to the driver, and it must be decrypted again.

The SCC encrypts and decrypts using Triple DES with an internally stored key. When the SCC is in Secure mode, it uses its secret, unique-per-chip key. When it is in Non-Secure mode, it uses a default key. This ensures that secrets stay secret if the SCC is not in Secure mode.

Not all functions that could be provided in a 'high level' manner have been implemented. Among the missing are interfaces to the ASC/AIC components and the timer functions. These and other features must be accessed through scc_read_register() and scc_write_register(), using the #define values provided.

29.2.2 Random Number Generator Accelerator (RNGA)

- The FSL SW API is provided in both user mode and kernel mode.
- Blocking calls and callback and non-callback non-blocking support are provided.
- Random number support allows the generation of an arbitrary number of bytes of random data, as well as the addition of additional entropy to the hardware random number generator.
- No other cryptographic functions (hashing, symmetric encryption, and so on) are supported by this driver.
- There is a debug-only interface to read/write RNG registers.

29.2.3 Run-Time Integrity Checker (RTIC)

The Run-Time Integrity Checker is intended to serve as a Hash accelerator. It has the ability to verify the memory contents during system boot (Hash-Once) and during run time (Run-Time) execution. The contents of both contiguous and non-contiguous memory blocks can be checked using the RTIC module.

The following APIs can be used to access RTIC module:

- rtic_configure_mode—Configures the mode of operation of the RTIC; that is, Run-Time or Hash-Once. The parameter specifying which memory blocks need to be enabled must also be passed; that is, Memory A,B,C,D. RTIC does not support enabling multiple memory blocks that are not grouped together (that is, you cannot enable only memory blocks A and C without enabling memory B).
- rtic_start_hash—Starts the hashing process of either Run-Time or Hash-Once. If the Run-Time mode is selected then Run-Time memory registers need to be enabled before the start of hashing or vice-versa.
- rtic_configure_mem_blk—Enables the configuration of the start address and block length of the memory content to be hashed. Start address indicates the starting location from where the data in the memory is to be hashed. The start address and block length should be aligned to a 4-byte boundary. The number of blocks that need to be hashed is loaded in the block count register. There are four memory blocks available. The user can configure any one of these four memory blocks by passing their appropriate address and block length to be hashed.
- rtic_hash_result—Reads the 160-bit hash result from the RTIC memory blocks A, B, C, D Hash Result Register.
- rtic_get_status—Reads the status register of the RTIC.
- rtic_get_control—Reads the control register of the RTIC.
- rtic_configure_interrupt—Enables or disables the interrupt for the RTIC module.
- rtic_get_faultaddress—Reads the fault address register of the RTIC.

If the RTIC is selected for non-interrupt based configuration (polling mode), then other operations are blocked during hashing until the hashing is done. If the RTIC is in HASH ONCE mode, it becomes idle

after hashing is done. If RTIC is in RUN TIME CHECK mode, the RTIC Control register cannot be changed while the RTIC is busy. Thus, the Suspend and Resume functions are not implemented in RUN TIME CHECK mode.

29.3 Requirements

Requirements for SCC:

- SCC provides an interface to check whether the SCC fuse is blown or not (SCC Disabled/Enabled).
- It provides interface to configure the Red & Black memory area addresses and number of blocks to be encrypted/decrypted.
- SCC provides an interface to load the data to be encrypted.
- SCC provides an interface to load the data to be decrypted.
- SCC provides an interface to start the Ciphering mechanism.
- SCC provides an interface to report back the status of the KEM module.
- SCC provides an interface to zero blocks in the Red/Black memory area.
- SCC provides an interface to check for the boot type, that is, Internal or External.
- SCC provides an interface to raise a software alarm.
- SCC provides an interface to report back the status of the Zeroize module.
- SCC provides an interface to configure the AIC's start and end algorithm sequence number.
- SCC provides an interface to check the sequence of the algorithm.
- SCC provides an interface to find the next sequence number given the current sequence number.
- SCC provides an interface to configure the Security Timer.
- SCC provides an interface to report back the status of the Security Timer module.

Requirements for RNGA:

- Provides an interface to configure the RNGA module.
- Provides an interface to enter the initial seed number to the RNGA module.
- Provides an interface to read the random number generated from the RNGA module.
- Provides an interface to report back the status of the RNGA module.

Requirements for RTIC:

- Provides an interface to configure the RTIC module.
- Provides an interface to data during run-time mode.
- Provides an interface to hash data during one-time mode.
- Provides an interface to report back the status of the RTIC module.

29.4 Source Code Structure

This section contains the various files that implement the Security modules. Table 29-1 lists the headers and some source files associated with the security driver.

- The C source files are available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/security` directory.
- Header files are available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc`.
- The RNG driver also depends on the header files in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/security/sahara2/include`.

Table 29-1. SCC File List

File	Description
`Makefile`	Used to compile, link and generate the final binary image.
`rng/rng_driver.c`	Contains the core driver.
`rng/shw_driver.c`	Contains the `shw` API.
`mxc_scc_driver.h`	Header file related to SCC module interface.
`mxc_scc_internals.h`	Header file which contains definitions needed by the SCC driver. This is intended to be the file that contains most or all of the code or changes needed to port the driver.
`mxc_scc2_driver.h`	Header file related to SCCV2 module interface.
`scc2_internals.h`	Header file with SCCV2 driver-related definitions.
`rng/include/`	Contains the include files.

29.5 Configuration

This section provides the configurations required to execute the security system during boot-up.

29.5.1 Linux Kernel Configuration Options

The following Linux kernel configurations are provided for this module. In order to get to the security configuration, use the command `./ltib -c` when located in the <ltib dir>. In the screen select configure kernel, exit and a new screen will appear.

- CONFIG_MXC_SECURITY_SCC—Use the SCC module. In `menuconfig`, it is available under `Device Drivers > MXC Support drivers ->MXC Security Drivers > MXC_SCC_Driver`. By default, this option is Y for platform.
- CONFIG_MXC_SECURITY_RNGA—Use the RNGA module core API's. In `menuconfig`, it is available under `Device Drivers > MXC Support drivers ->MXC Security Drivers > MXC_RNG_Driver`. By default, this option is Y for platform.
- CONFIG_RNGA_TEST_DRIVER—Debug the RNGA module. In `menuconfig`, it is available under `Device Drivers > MXC Support drivers > MXC Security Drivers > MXC RNG Driver > MXC RNG debug register`. By default, this option is N for platform. Please note that this configuration

should be enabled for `rnga_read_register()` and `rnga_write_register()` functions to be defined and exported. This may affect inserting the test driver modules, which might assume the availability of these functions.

- CONFIG_MXC_SECURITY_RTIC—Use the RTIC module core API's. In `menuconfig`, it is available under `Device Drivers > MXC Support drivers > MXC Security Drivers > MXC RTIC driver`. By default, this option is Y for platform.
- CONFIG_RTIC_TEST_DEBUG—Debug the RTIC module. In `menuconfig`, it is available under `Device Drivers > MXC Support drivers > MXC Security Drivers > MXC RTIC driver > MXC RTIC module debugging`. By default, this option is N for platform.

29.5.2 Source Code Configuration Options

NOTE

This section does not apply for the i.MX31 3-Stack Board.

29.5.2.1 Board Configuration Option

To Configure the SCC, perform the following steps:

1. Install Icepick and point it to license file.
2. Blow the following fuses to SCC key 0 - SCC Key 20. Please refer to the IC documentation for register details.

```
SCC Key0   = 0x77
SCC Key1   = 0xff
SCC Key2   = 0x3a
SCC Key3   = 0x76
SCC Key4   = 0x02
SCC Key5   = 0xb0
SCC Key6   = 0x0a
SCC Key7   = 0x0d
SCC Key8   = 0x90
SCC Key9   = 0x76
SCC Key10  = 0xf8
SCC Key11  = 0x07
SCC Key12  = 0x13
SCC Key13  = 0x9e
SCC Key14  = 0x36
SCC Key15  = 0xd3
SCC Key16  = 0xfa
SCC Key17  = 0x00
SCC key18  = 0x00
SCC Key19  = 0x9d
SCC Key20  = 0xfe
```

Follow the instructions below to program the SCC key using Icepick:

1. Run Icepick
2. openSocket <IP Address of ICE>
3. initZas
4. source util_fuse_<platform>.tcl

5. init_iim
6. blow_fuse bank row bit

   Step 6 command will write desired fuse. Here, the parameters passed to blow_fuse are bank, row and bit. For information about parameters to be passed refer to the appropriate platform's L3 specification.

   Example: The following example shows how to program the value 0x77 into SCC Key0.

```
blow_fuse 1 1 0
blow_fuse 1 1 1
blow_fuse 1 1 2
blow_fuse 1 1 4
blow_fuse 1 1 5
blow_fuse 1 1 6
```

7. sense_fuse bank row bit

   The command in Step 7 will read the desired fuse value.

8. Write the following ASC Sequence in the debugger script (init_sdram.txt)

```
setmem /32 0x53FAD008 =0x00005CAA
setmem /32 0x53FAD00C =0x00002E55
setmem /32 0x53FAD010 =0x00002E55
```

9. Configure the boot mode pins SW7-1 and SW7-2 to Internal Boot.

29.6 Interrupt Requirements

There are no interrupt requirements in this module, as it provides only an API interface to underlying hardware.

29.7 Usage Example

RTIC:

To hash the data from memory during run-time or one time.

To hash data in contiguous or non-contiguous Flash memory locations.

1. For Run-Time hashing of data in the memory, do the following:

```
rtic_runtime_hash(start_address, block_length)
{
     /* Pass the parameter for start_address as physical address */
      rtic_config_mem_blk(start_address, block_length, Mem Block=A/B/C/D);   /* Set
the addr, length
                                                                  * and mem block */
        rtic_configure_mode(mode = Run-Time, Mem Block=A/B/C/D);/* Configure the
RTIC for mode, Run-Time or Hash-Once.*/
        rtic_configure_interrupt(irq_enable);/* Configure the RTIC for interrupt
enable/disable*/
        rtic_start_hash(start hash);/* Start hashing of data*/
        while (rtic_status() != RTIC_STAT_HASH_ERR);/* check the status for any
error occurrence */
      if(rtic_get_status == RTIC_STAT_HASH_DONE)
     /* HAshing for RUN-TIME is success */
        rtic_hash_result(* hash_result);/* Read the 160 bit hash data from the
register*/
```

```
}
```

2. For One-Time hashing of data in the memory, do the following:

```
rtic_onetime_hash(start_address, block_length)
{
     /* Pass the parameter for start_address as physical address */
         rtic_config_mem_blk(start_address, block_length, Mem Block = A/B/C/D);  /*
Configure start addr,
                                                                    block length*/
         rtic_config_mode(mode = one_time);/* Configure to one time hash mode*/
         rtic_config_interrupt(irq_enable);/* Configure the RTIC for interrupt
enable*/
         rtic_start_hash(start_hash);/* Start the hashing of data*/
         while (rtic_get_status()!=HASH_DONE);/* check the status for any error
occurrence (if done for polling mode)*/
         rtic_hash_result(* hash_result);/* Read the 160 bit hash data from the
register*/
}
```

Chapter 30
Inter-IC (I^2C) Driver

The MXC I^2C driver for Linux has two parts: an I^2C bus driver and an I^2C chip driver. The I^2C bus driver is a low level interface that is used to talk to the I^2C bus, while the I^2C chip driver acts as an interface between other device drivers and the I^2C bus driver.

I^2C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices.

30.1 I^2C Bus Driver Overview

The I^2C bus driver is invoked only by the MXC 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 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:

- Compatibility with the I^2C bus standard
- Supports bit rates up to 400 kbps
- Starts and stops signal generation/detection
- Acknowledge bit generation/detection
- Interrupt-driven, byte-by-byte data transfer
- Supports standard I^2C master mode

The I^2C slave mode may be supported by a separate driver.

30.2 I^2C Client Driver Overview

The I^2C client 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 their 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 MXC I^2C bus driver and calls the appropriate function in the I^2C bus driver to do the data transfer. This driver provides the following functions to other device drivers:

- A read function to read the device registers
- A write function to write to the device registers

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

30.3 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.
- An address is received that matches its own specific address in slave-receive mode.
- Arbitration is lost.

30.4 Software Operation

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

30.4.1 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`. That field is a pointer to the `i2c_algorithm` structure that describes how data is transferred over the MXC 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.

On startup, the MXC I^2C bus adapter is registered with the I^2C core when the driver is loaded. Certain MXC 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.

30.4.2 I^2C Client Driver Software Operation

The MXC I^2C chip driver controls an individual I^2C device that lives on the MXC 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 MXC 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.

30.5 Requirements

The MXC I^2C driver meets the following requirements:

- Supports the I^2C communication protocol.
- Supports the I^2C master mode of operation.

- Does not support the I^2C slave mode of operation.

30.6 Source Code Structure

Table 30-1 lists the I^2C bus driver source files available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/i2c/busses`.

Table 30-1. I^2C Bus Driver Files

File	Description
`mxc_i2c.c`	I^2C bus driver source file

30.7 Configuration

30.7.1 Linux Menu Configuration Options

In order to get to the I^2C configuration, use the command `./ltib -c` when located in the `<ltib dir>`. In the screen, select `Configure Kernel`, exit, and a new screen will appear.

The `I2C_MXC` Linux kernel configuration is provided for this module. This option is available under `Device Drivers > I2C support > I2C Hardware Bus support > MXC I2C support`.

30.8 Programming Interface

The I^2C device driver could use the standard SMBus interface to read and write the registers of the device connected to the MXC I^2C bus. For more information, see `<ltib_dir>/rpm/BUILD/linux-2.6.26/include/linux/i2c.h`.

30.9 Interrupt Requirements

The I^2C module generates many kinds of interrupts. The highest interrupt rate is associated with the transfer complete interrupt.

Table 30-2. I^2C Interrupt Requirements

Parameter	Equation	Typical	Worst-Case
Rate	Transfer Bit Rate/8	25,000/sec	50,000/sec
Latency	8/Transfer Bit Rate	40us	20us

The typical value of the transfer bit-rate is 200 kbps. The worst-case is based on a baud rate of 400 kbps (the maximum supported by the I^2C interface).

30.10 Device-Specific Information

The x in I^2Cx denotes the individual I^2C number.

Table 30-3. Default Configuration

Option	I2C_NR	I2C1_FRQ_DIV	I2C2_FRQ_DIV	I2C3_FRQ_DIV
i.MX31	1	0x17	N/A	N/A

Chapter 31
One-Wire Driver

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

31.1 Hardware Operation

The One-Wire interface provides the communication line to a 1 kbit Add-Only Memory (DS2502). The interface can send or receive one bit at a time. The protocol for accessing the DS2502 is defined by Dallas Semiconductors. The DS2502 holds battery characteristics information. The 1-wire is a peripheral device to the Core and communicates with it through the IP interface.

31.2 Software Operation

The One-Wire module's software implementation is through a One-Wire (OWire) driver.

31.3 Requirements

This OWire implementation meets the following requirements:

- Supports a single OWire memory device connected to the MXC OWire peripheral for read/write bit and read/write byte operations. (For instance, a module serving as a data-link layer.)
- Supports the OWire peripheral in the MXC product for single device detection and selection. (Module serving as a network layer.)
- Interfaces to the OWire peripheral in the MXC at the read/write block and read/write page level. (Module serving as the transport layer.)
- Exposes the single connected OWire device through existing Linux device interface(s).
- Supports open, close, read and write operations.

31.4 Source Code Structure

The OWire module is implemented in
`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/w1/masters/mxc_w1.c`.

Table 31-1. Owire File List

File	Description
`mxc_w1.c`	OWIRE function implementations

31.5 Configuration

31.5.1 Linux Menu Configuration Options

The Linux kernel configuration option, CONFIG_W1_MXC, is provided for this module. In order to get to the one-wire configuration, use the command `./ltib -c` when located in the `<ltib dir>`. In the screen, select **Configure Kernel**, exit, and a new screen will appear.

This is the configuration option for the MXC OWire driver used for the MXC 1-Wire interface. In the `menuconfig`, this option is available under the following:

- Device Divers > Dallas's 1-wire support > 1-wire bus master > Freescale's MXC driver for 1-wire
- Device Divers > Dallas's 1-wire support > 1-wire Slaves > 4kb EEPROM family support (DS2433)

Options `CONFIG_W1` and `CONFIG_W1_DS2433` should also be selected.

Chapter 32
Configurable Serial Peripheral Interface (CSPI) Driver

The configurable serial peripheral interface (CSPI) driver implements a standard Linux driver interface to MXC CSPI Controllers. It is based on SPI Framework by David Brownell. It supports the following features:

- Interrupt- and SDMA-driven transmit/receive of bytes
- Supports multiple master controller interface
- Supports multiple slaves select
- Supports multi-client requests

32.1 Hardware Operation

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

- Master/slave-configurable
- Two chip selects allowing maximum of 4 different slaves each for master mode operation
- Up to 32-bit programmable data transfer
- 8 by 32-bit FIFO for both Tx and Rx data
- Polarity and phase of the Chip Select (SS) and SPI Clock (SCLK) are configurable

32.2 Software Operation

32.2.1 SPI Sub-System in Linux

The CSPI driver layer is located between the client layer (PMIC and SPI Flash are examples of the clients) and the MXC hardware access layer. Figure 32-1 shows the block diagram for SPI subsystem in Linux.

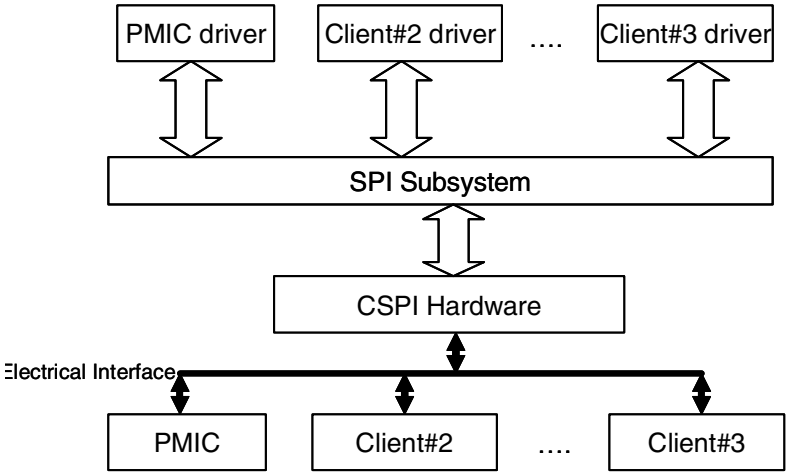


Figure 32-1. SPI Sub-system

SPI requests always go into I/O queues. Requests for a given SPI device are always 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.

All SPI clients must have a protocol driver associated with them. And those must all be sharing the same controller driver. Only the controller driver can interact with the underlying SPI hardware module. Figure 32-2 shows how the different SPI drivers are layered in the SPI subsystem.

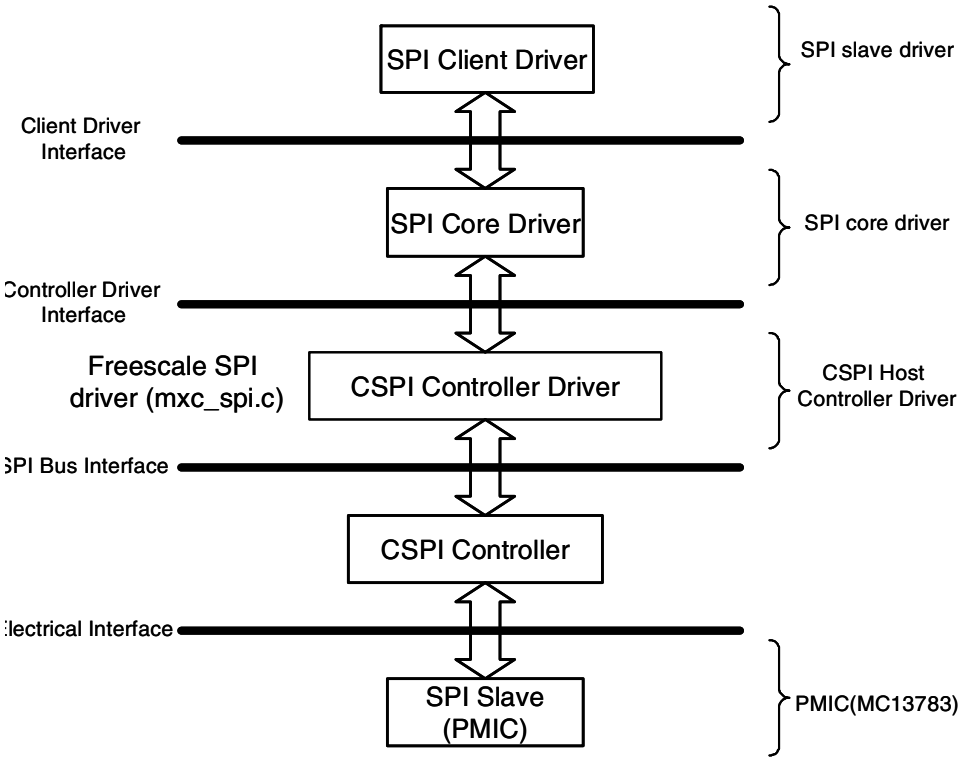


Figure 32-2. Layering of SPI Drivers in SPI subsystem

32.2.2 Limitations

- It does not have SPI Slave logic implementation yet.
- It does not support a single client connected to multiple masters.
- It presently does not implement the user space interface with the help of the device node entry but supports "`sysfs`" interface.

32.2.3 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:

1. The init function `mxc_spi_init( )`—Registers the `device_driver` structure.
2. The 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.
3. The 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.

4. SPI transfer function `mxc_spi_transfer( )`—Handles data transfers operations.
5. SPI setup function `mxc_spi_setup( )`—Initializes the current SPI device.
6. SPI driver ISR `mxc_spi_isr( )`—Called when the data transfer operation is completed and an interrupt is generated.

32.2.4 CSPI Synchronous Operation

Figure 32-3 shows how CSPI provides synchronous read/write operations.

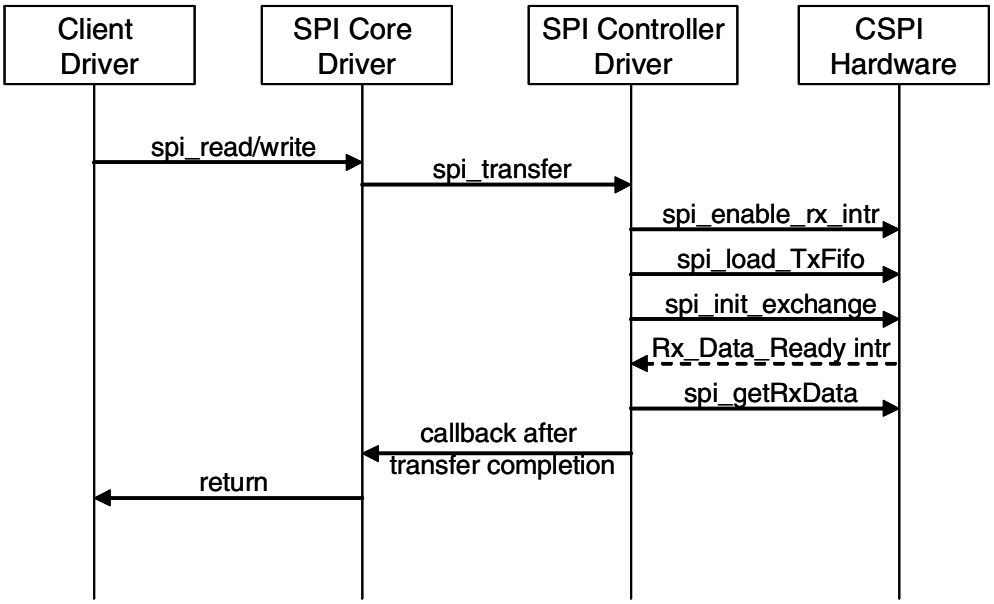


Figure 32-3. CSPI Synchronous Operation

32.2.5 PMIC Access

Figure 32-4 shows the how PMIC can be accessed through the SPI subsystem.

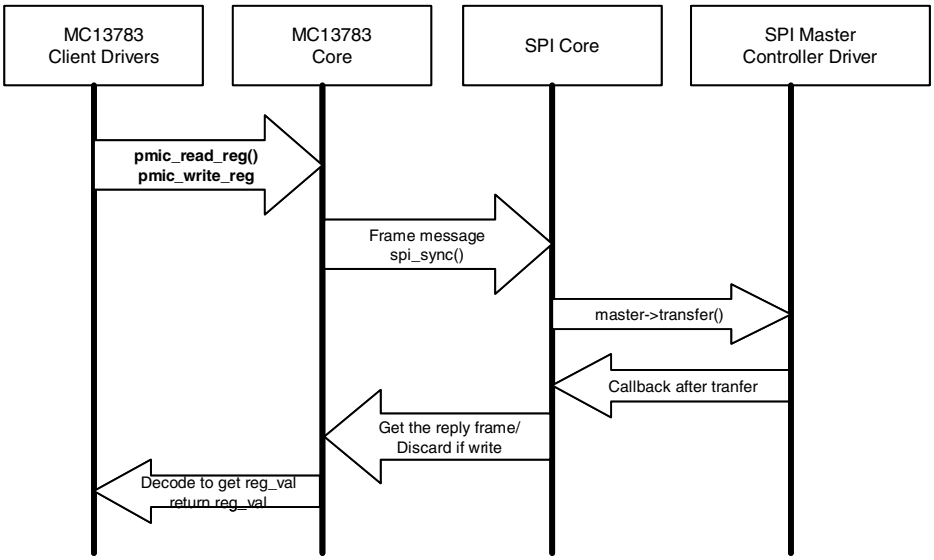


Figure 32-4. PMIC Access through SPI

32.3 Requirements

The CSPI module meets the following requirements:

- Implements each of the functions required by a CSPI module to interface to Linux.
- Provides support for multiple SPI master controllers.
- Provides support to handle multi-client synchronous requests.

32.4 Source Code Structure

Table 32-1 lists the source files available in the devices directory:

`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/spi/.`

Table 32-1. CSPI Source File List

File	Description
`mxc_spi.c`	Freescale SPI Master Controller driver

32.5 Configuration

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

- `CONFIG_SPI`: Build support for the SPI core. In menuconfig, this option is available under Device Drivers> SPI Support > SPI Support.

- CONFIG_BITBANG: This is library code, and is automatically selected by drivers that need it. SPI_MXC selects it. In menuconfig, this option is available under Device Drivers > SPI Support > Bitbanging SPI master
- CONFIG_SPI_MXC: This 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: This is to select the CSPI hardware modules into the build (where n = 1, 2, or 3). In menuconfig, this option is available under Device Drivers > SPI Support > CSPIn.
- CONFIG_SPI_MXC_TEST_LOOPBACK: This is 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.
- CONFIG_SPI_MXC_DMA: This is to select the enable DMA function of CSPI. In menuconfig, this option is available under Device Drivers> SPI Support > MXC CSPI controller as SPI Master. By default this is disabled.

32.6 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.

32.7 Interrupt Requirements

The SPI interface generates interrupts. CSPI interrupt requirements are listed in Table 32-2.

Table 32-2. 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.

32.8 Device-Specific Information

Table 32-3 lists the number of CSPI controllers in different i.MX family platforms.

Table 32-3. CSPI Controllers in i.MX Family Platforms

Platforms (SOC)	No. of CSPI Controllers
i.MX31	3

Chapter 33
MMC/SD/SDIO Host Driver

The MMC/SD/SDIO Host driver implements a standard Linux driver interface to the MMC/Secure Digital Host Controller (SDHC) or the enhanced MMC/SD host controller (eSDHC). The host driver is part of the Linux kernel's MMC framework.

Table 33-1. Available Platforms

module name	Available platforms
SDHC	i.MX31

The MMC driver has the following features:

- 1-bit or 4-bit operation
- Supports card insertion and removal events
- Supports the standard MMC commands
- PIO and DMA data transfers
- Power management

33.1 Hardware Operation

The MMC communication is based on an advanced 7-pin serial bus designed to operate in a low voltage range. The SDHC or eSDHC module supports MMC along with SD memory and I/O functions. The SDHC or 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 SDHC or eSDHC only supports the SD bus protocol.

For the SDHC module, the SDHC command number and the SDHC command argument register allows a command to be issued to the card. The SDHC command and data control register allows the users to specify the format of the data and the 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.

For SDHC, there is an 8-bit x16-bit FIFO to store the response from the card in the SDHC. The SDHC Response FIFO Access register is used to access this FIFO. The SDHC uses two 64-byte data buffers. These buffers are used as temporary storage for data being transferred between the host system and the card, and vice versa. The SDHC data buffer access register bits hold 32-bit data upon a read or write transfer. For reception, follow these steps:

1. The SDHC controller generates an SDMA request when the FIFO is full.

2. Upon receiving this request, SDMA starts transferring data from the SDHC FIFO to system memory by reading the data buffer access register.

To transmit data, follow these steps:

1. The SDHC controller generates an SDMA request whenever the transmit FIFO is empty.
2. Upon receiving this request, the SDMA starts moving data from the system memory to the SDHC FIFO by writing to the Data Buffer Access Register for a number of pre-defined bytes.

The read-only SDHC Status Register provides SDHC operations status, application FIFO status, error conditions, and interrupt status.

For both SDHC and eSDHC modules, when certain events occur in the module, then all have the ability to generate an interrupt as well as setting corresponding Status Register bits. The SDHC interrupt control register and eSDHC interrupt status enable and signal enable registers allow the user to control whether these interrupts should occur.

33.2 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 SDHC or eSDHC.

Figure 33-1 shows how the MMC-related drivers are layered.

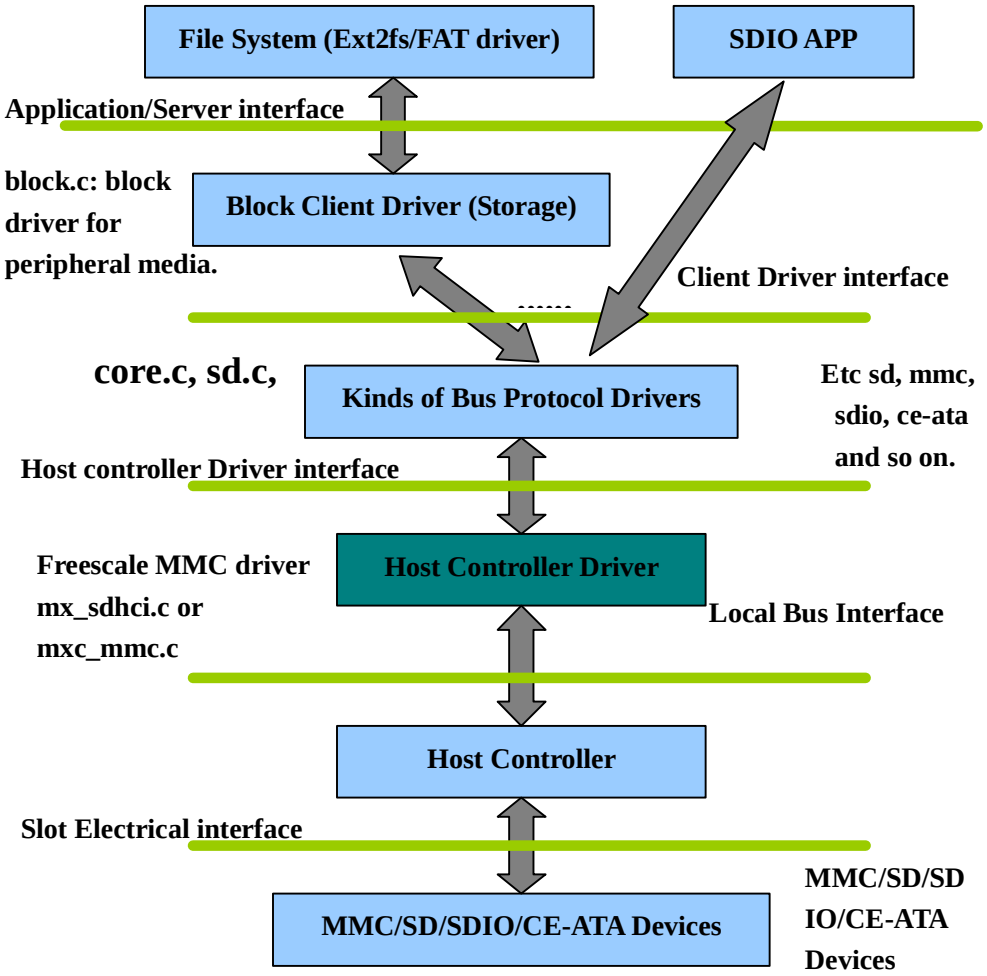


Figure 33-1. Layering of MMC drivers

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

For SDHC:

1. The init function `mxcmci_init()`—Registers the device_driver structure.
2. The probe function `mxcmci_probe()`—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 SDHC I/O pins and resets the hardware. Requests for IRQ and allocates DMA channel along with transfer completion routine `mxcmci_dma_irq()`.
3. `mxcmci_set_ios()`—Sets bus width, voltage level, and clock rate according to core driver requirements.

4. `mxcmci_request()`—Handles both read and write operations. Sets up the number of blocks and block length. It configures an DMA channel, allocates safe DMA buffer and starts the DMA channel. It configures the SDHC command number register and SDHC command argument register to issue a command to the card. This function starts the SDMA and starts the clock.
5. MMC driver ISR `mxcmci_gpio_irq()`—Called when the MMC card is detected or removed.
6. MMC driver ISR `mxcmci_irq()`— Interrupt from SDHC called when command is done or errors like CRC or buffer underrun or overflow occurs.

DMA completion routine `mxcmci_dma_irq()`—Called after completion of a DMA transfer. It informs the MMC core driver of a request completion by calling `mmc_request_done()` API.

33.3 Requirements

- Provides support for multiple SDHC modules.
- Provides all the entry points to interface with the Linux MMC core driver.
- Supports MMC and SDcards.
- Recognizes data transfer errors like command time outs and CRC errors.
- Supports power management.
- Conforms to the Linux coding standards.

33.4 Source Code Structure

Table 33-2 lists the SDHC source files available in the source directory `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mmc/host/`.

Table 33-2. SDHC Driver File List

File	Description
`mxc_mmc.h`	Header file defining registers
`mxc_mmc.c`	SDHC driver

33.5 Linux Menu Configuration Options

In order to get to the mmc configuration, use the command `./ltib -c` when located in the <ltib dir>. In the screen, select **Configure Kernel**, exit, and a new screen will appear.

The following Linux kernel configurations are provided for this module:

- CONFIG_MMC—Build support for the MMC bus protocol. In `menuconfig`, this option is available under Device Drivers > MMC/SD Card support. By default, this option is Y for all architectures.
- 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 support. By default, this option is Y for all architectures.

- CONFIG_MMC_MXC—i.MX MMC driver used for the i.MX SDHC ports. In `menuconfig`, this option is available under Device Drivers > MMC/SD Card Support > Freescale MXC Multimedia Card Interface support. This option is available for the i.MX31 platforms.

33.6 Programming Interface

This driver implements the functions required by the MMC bus protocol to interface with the i.MX SDHC and eSDHC modules. For additional information, see the *BSP API Document.*

Chapter 34
Universal Asynchronous Receiver/Transmitter (UART) Driver

The low-level universal asynchronous receiver transmitter (UART) driver interfaces the Linux serial driver API to all the UART ports. It has the following features:

- Supports interrupt-driven and SDMA-driven transmit/receive of characters.
- Supports standard Linux baud rates up to 4 Mbps.
- Supports transmitting and receiving characters with 7-bit and 8-bit character lengths.
- Supports transmitting 1 or 2 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.
- Supports odd and even parity.
- Supports 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.
- Supports 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.
- Recognize 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 ioctls. 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. While doing this the user should specify the UART type to be 52. This is defined in the `serial_core.h` header file.
- Serial IrDA support.
- Supports power management feature by suspending and resuming the UART ports
- Supports the 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. The number of available UART ports varies from device to device.

Autobaud detection is not supported.

34.1 UART Driver Hardware Operation

Refer to the IC/Hardware Specification 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-words 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.

34.2 UART Driver Software Operation

The Linux OS contains a core UART driver that handles 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 size 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.

34.3 UART Driver Requirements

The UART driver meets the following requirements:

- Supports baud rates up to 4 Mbps.
- Recognizes frame and parity errors only in interrupt-driven mode. The UART driver 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).
- Is able to 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.
- Supports power management.
- Supports interrupt-driven and DMA-driven data transfer.

34.4 UART Driver Source Code Structure

Table 34-1 lists the source files associated with the UART driver that are available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/serial`.

Table 34-1. UART Source And Header File List

File	Description
`mxc_uart.c`	UART low level driver
`serial_core.c`	Core UART driver that is included as part of standard Linux
`mxc_uart_reg.h`	UART file for the register values

Table 34-2 lists the header files associated with the UART driver.

Table 34-2. UART Global Header File List

File	Description
`<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc/mxc_uart.h`	UART header that contains UART configuration data structure definitions
`<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3/board-mx3_3stack.h`	Holds some UART board specific configuration options

The source files, `serial.c/serial.h`, are associated with the UART driver that is available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3`. The source file contains UART configuration data and calls to register the device with the platform bus.

34.5 UART Driver Configuration

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

34.5.1 Linux Menu Configuration Options

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

- CONFIG_SERIAL_MXC—This configuration option is 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 for all architectures.
- CONFIG_SERIAL_MXC_CONSOLE—This configuration option 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 for all architectures.

34.5.2 Source Code Configuration Options

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

34.5.2.1 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 individual UART number is listed in Table 34-5.

34.5.2.2 Board Configuration Options

The following board-specific configuration options for the driver can be set within `board.h`:

- UART Mode (`UARTx_MODE`)—Specifies whether the UART is configured to be in 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.
- MAX_UART_BAUDRATE—Specifies the maximum baud rate to support on the board. Any value up to 1500000 can be specified.

The x in UARTx denotes the individual UART number. The default configuration for each individual UART number is shown in Table 34-5.

34.6 UART Driver 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.

34.7 UART Driver 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 34-3.

Table 34-3. UART Interrupt Requirements

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

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 1 and a transmitter trigger level (Txtl) of 2. The worst-case is based on a baud rate of 1.5 Mbps (max supported by the UART interface) with an Rxtl of 1 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.

34.8 Device Specific Information

34.8.1 UART Ports

The UART ports can be accessed through the device files `/dev/ttymxc0`, `/dev/ttymxc1`, etc. where `/dev/ttymxc0` refers to UART 1. The number of UART ports on a particular platform are listed in Table 34-4.

34.8.2 Board Setup Configuration

Table 34-4. UART General Configuration

Platform	UART_NR	MAX BAUDRATE
i.MX31	5	1500000 (1.5 Mbps)

Table 34-5. UART Active/Inactive Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	1	1	1	0	0	--

Table 34-6. UART IRDA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	NO_IRDA	NO_IRDA	NO_IRDA	NO_IRDA	NO_IRDA	--

Table 34-7. UART Mode Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	MODE_DCE	MODE_DCE	MODE_DTE	MODE_DTE	MODE_DTE	--

Table 34-8. UART Shared Peripheral Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	-1	-1	SPBA_UART3	-1	-1	--

Table 34-9. UART Hardware Flow Control Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	1	0	1	1	1	--

Table 34-10. UART DMA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	0	0	1	0	0	--

Table 34-11. UART DMA RX Buffer Size Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	1024	512	1024	512	512	--

Table 34-12. UART UCR4_CTSTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	16	-1	16	16	16	--

Table 34-13. UART UFCR_RXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	16	16	16	16	16	--

Table 34-14. UART UFCR_TXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	16	16	16	16	16	--

Table 34-15. UART Interrupt Mux Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	INTS_MUXED	INTS_MUXED	INTS_MUXED	INTS_MUXED	INTS_MUXED	--

Table 34-16. UART Interrupt 1 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	INT_UART1	INT_UART2	INT_UART3	INT_UART4	INT_UART5	--

Table 34-17. UART Interrupt 2 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	-1	-1	-1	-1	-1	--

Table 34-18. UART interrupt 3 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX31	-1	-1	-1	-1	-1	--

34.9 Early UART Support

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

Linux kernel 2.6.10 and later kernels have an “early UART” driver that works 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 35
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-telephone interconnects. It features ease-of-use; for example, it supports plug-and-play, port expansion, and any new USB peripheral uses the same type of port.

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

- Full Speed / Low Speed Host Only core (HOST 1)
- High Speed / Full Speed / Low Speed Host Only core (HOST2)
- 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, MTP, and CDC (Communication Devices Class) drivers
- Embedded DMA controller

35.1 Architectural Overview

A USB host system is composed of a number of hardware and software layers. Figure 35-1 illustrates a conceptual block diagram of the building block layers in a host system that work in concert to support USB 2.0.

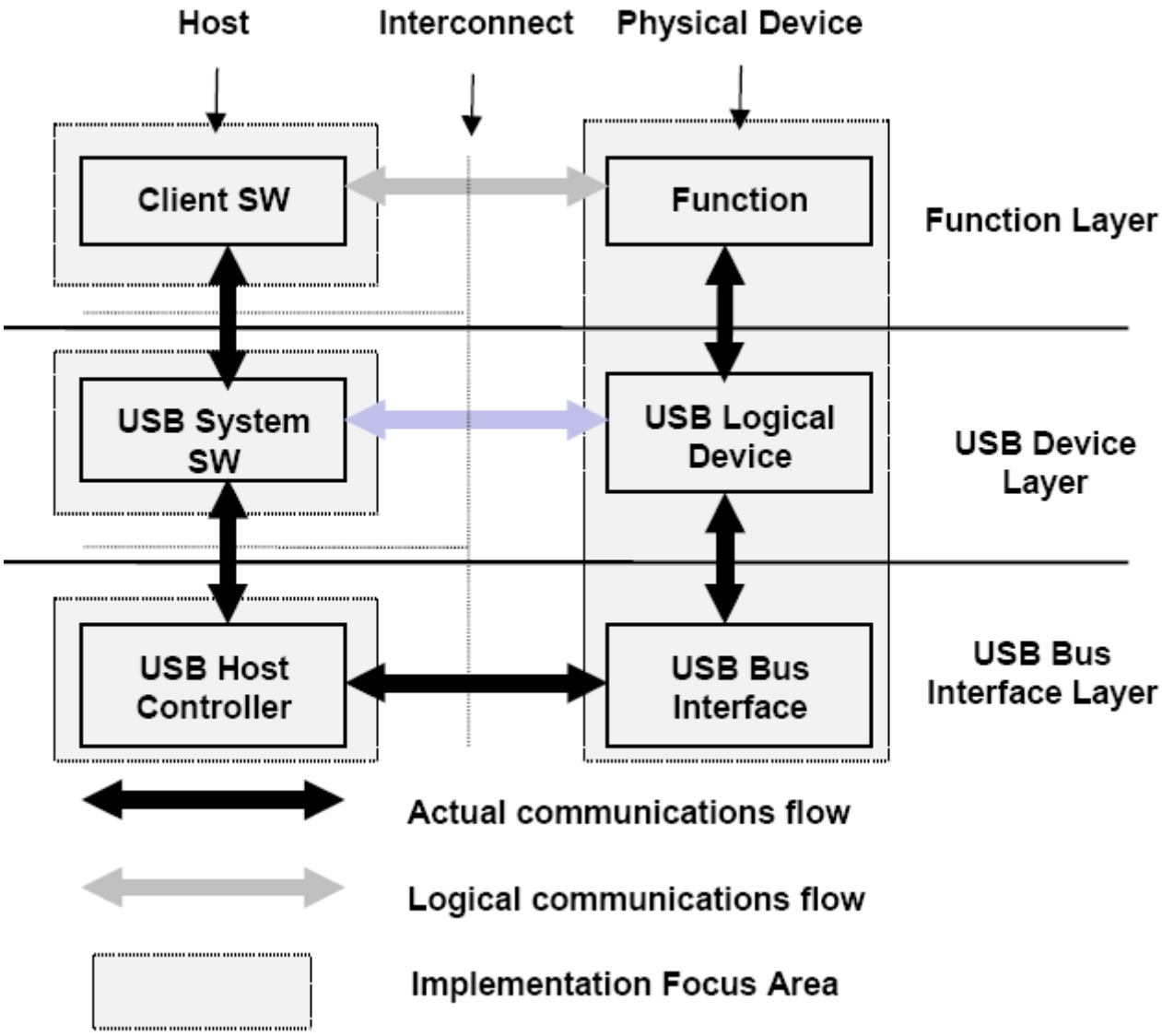


Figure 35-1. Block Diagram

35.2 Hardware Operation

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

35.3 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 endpoint, making it usable.
- `fsl_ep_disable()`—specifies 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.

35.4 Requirements

The USB stack meets the following requirements:

- Supports USB device mode
- Supports mass storage device profile – subclass 8-1. (RBC set)
- Supports USB host mode

- Supports HID host profile – subclasses 3-1-1 and 3-1-2. (USB mouse and keyboard)
- Supports mass storage host profile – subclass 8-1
- Supports Ethernet USB profile – subclass 2
- Supports DC PTP transfer
- Supports MTP device mode

35.5 Source Code Structure

Table 35-1 lists the source files available in the source directory,
`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/usb`.

Table 35-1. USB Driver File List

File	Description
`host/ehci-hcd.c`	host driver source file.
`host/ehci-arc.c`	host driver source file.
`host/ehci-mem-iram`	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`	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.

Table 35-2 lists the platform related source files in the directory,
`<ltib_dir>/rpm/BUILD/linux-2.6.26/include/asm-arm/arch-mxc`.

Table 35-2. USB Platform Source File List

File	Description
arc_otg.h	USB register define.

Table 35-3 lists the platform-related source files in the directories:

`<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3/`

Table 35-3. USB Platform Header File List

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

Table 35-4 lists the common platform source files in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/plat-mxc`.

Table 35-4. USB Common Platform File List

File	Description
isp1301xc.c	ISP1301 USB driver
isp1504xc.c	ISP1504 USB driver
mc13783_xc.c	mc13783 USB driver
utmixc.c	internal utmi transceiver driver
serialxc.c	internal serial transceiver driver
usb_common.c	common platform related part of USB driver

35.6 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- 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 under Device 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 under Device drivers > USB support > Support for Host1 port on Freescale controller. By default, this option is N.
- CONFIG_USB_EHCI_ARC_H2—Build support for selecting the USB Host2. In menuconfig, this option is available under Device drivers > USB support > Support for Host2 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 Y.
- 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 M.
- CONFIG_USB_HIDINPUT—Build support for USB HID input devices. In menuconfig, this option is available under Device drivers > HID devices. 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 > Support for USB Gadgets. By default, this option is *.
- CONFIG_USB_GADGET_ARC—Build support for ARC USB gadget. In menuconfig, this option is available under Device drivers > USB support > USB Gadget Support > Support for USB Gadgets > USB Peripheral Controller > Freescale USB Device Controller. By default, this option is Y.
- CONFIG_USB_GADGET_ARC_OTG—Build support for the USB OTG port in HS/FS peripheral mode. In menuconfig, this option is available under Device Drivers > USB support > USB Gadget Support > OTG support. By default, this option is Y.
- CONFIG_USB_ETH—Build support for Ethernet gadget. In menuconfig, this option is available under Device drivers > USB support > USB Gadget Support > Support for USB Gadgets > Ethernet Gadget. By default, this option is *.
- CONFIG_USB_ETH_RNDIS—Build support for Ethernet RNDIS protocol. In menuconfig, this option is available under Device drivers > USB support > USB Gadget Support > Support for USB Gadgets > Ethernet Gadget > RNDIS support (EXPERIMENTAL). 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 > Support for USB Gadgets > File-backed Storage Gadget. By default, this option is *.
- CONFIG_USB_G_SERIAL—Build support for ACM gadget. In menuconfig, this option is available under Device drivers > USB support > USB Gadget Support > Support for USB Gadgets > Serial Gadget. By default, this option is *.
- CONFIG_USB_EHCI_FSL_1504—Build support for selecting ISP1504 transceiver for OTG port host mode. In menuconfig, this option is available under Device drivers > USB support > Select transceiver for DR port > Philips ISP1504.
- CONFIG_USB_GADGET_FSL_1504—Build support for selecting ISP1504 transceiver for OTG port gadget mode. This option is available under Device drivers > USB support > USB Gadget Support > Select transceiver for DR port -> Philips ISP1504.
- CONFIG_USB_EHCI_FSL_1301—Build support for selecting ISP1301 transceiver. In menuconfig, this option is available under Device drivers > USB support > Select transceiver for DR port > Philips ISP1301.

- CONFIG_USB_GADGET_FSL_1301—Build support for selecting ISP1301 transceiver for OTG port gadget mode. In menuconfig, this option is available under Device drivers > USB support > USB Gadget Support > Select transceiver for DR port > Philips ISP1301.
- CONFIG_USB_EHCI_FSL_MC13783—Build support for selecting MC13783 transceiver for Host. In menuconfig, this option is availabe under Device drivers > USB support > Select transceiver for DR port > Freescale MC13783.
- CONFIG_USB_GADGET_FSL_MC13783—Build support for selecting MC13783 transceiver for OTG gadget mode. In menuconfig, this option is available under Device drivers > USB support > USB Gadget Support > Select transceiver for DR port > Freescale MC13783.

35.7 Programming Interface

This driver implements all the functions that are required by the USB bus protocol to interface with the i.MX USB ports. For more information, see the BSP API document.

35.7.1 Notes

Table 35-5. Default USB Settings

Default value	OTG HS	OTG FS	Host1	Host2(HS)	Host2(FS)
i.MX31 3-Stack	enabled	N/A	N/A	enabled	N/A

IC type is `IDT74CBTLV3257Q`. All these resistor packs are 0 Ohm.

The USB Host2 port: pin conflicts with NAND Flash. If Host2 is to be used, remove the NAND Flash card from board.

For the i.MX31 3-Stack board, only OTG HS and Host2(HS) are available. The transceiver of OTG HS is ISP1504; the transceiver of Host2(HS) is USB3317.

Chapter 36
Bluetooth Driver

The Bluetooth driver provides synchronous and asynchronous wireless connection among multiple devices. The synchronous oriented channel provides voice transmission. The asynchronous channel allows more time delay in data transmission. The synchronous and asynchronous data transfer between the host and Bluetooth chip is performed by different hardware interfaces. The SSI interface is used to transfer voice from the host to the Bluetooth chip. UART or USB is used for asynchronous data communication.

Based on the wireless connection, many services can be supported by profiles defined by the Bluetooth Group. On the i.MX platform, the A2DP and AVRCP profile is used to play music (mp3, wav, and so forth). The FTP profile provides access to the file system on another device. The SPP profile emulates a serial cable to provide a simply implemented wireless replacement for the existing RS-232 based serial communications applications. The handset profile is reserved for future support, so the SSI interface is reserved. The UART interface is used for communication between the host and the Bluetooth chip.

36.1 Hardware Operation

The platform uses the APM6628, which is a Bluetooth and Wi-Fi combination module that integrates CSR Bluetooth and the Wi-Fi chip. The Bluetooth/Wi-Fi chip in the APM6628 module works independently.

Figure 36-1 illustrates the hardware interface between i.MX31 3-Stack and the APM6628 module. Bluetooth and Wi-Fi share one reset signal. UART2 is used for data communication.

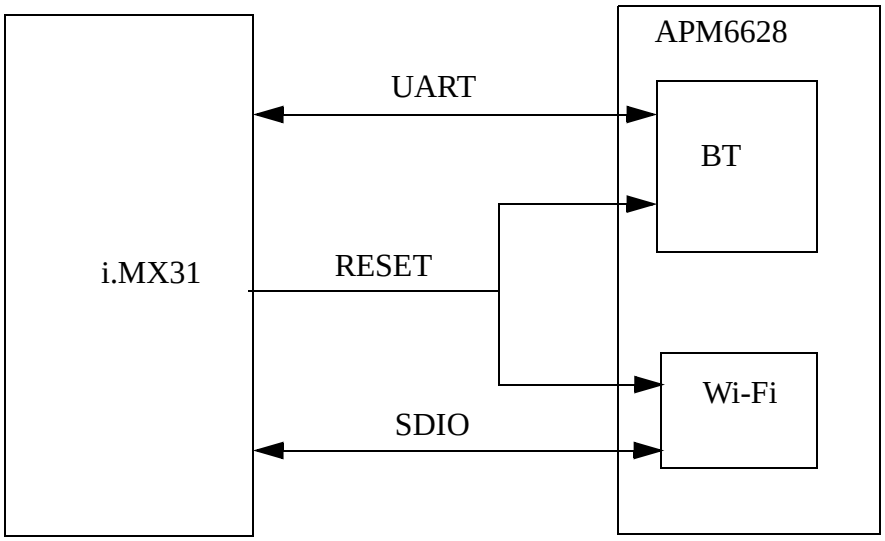


Figure 36-1. Bluetooth Hardware Interface for i.MX31 Platform

36.2 Software Operation

BlueCore™ Host Software (BCHS) is a Bluetooth protocol provided by a third-party company, Cambridge Silicon Radio (CSR). The porting of BCHS to Linux is divided into:

- A **user space port**, in which the BCHS protocol stack runs in user space together with the application.
- A **kernel space port**, in which the BCHS protocol stack runs in kernel space and the application runs in user space.

There are two ways to set up the user space port:

- The application and the BCHS protocol stack are running within the same process.
- The application and the BCHS protocol stack are running in two different processes.

In i.MX platform, the BCHS protocol stack runs in user space. And the application runs in the same process, as shown in Figure 36-2.

Encoding is used to minimize the bandwidth required for transferring the audio data. Thus, the encoding compresses the audio before transmission over the air. The A2DP profile mandates support for SBC encoding, and other codecs, such as MP3 and WMA, are optional. The A2DP source checks the capabilities of sink and then configures sink to select the dedicated codec.

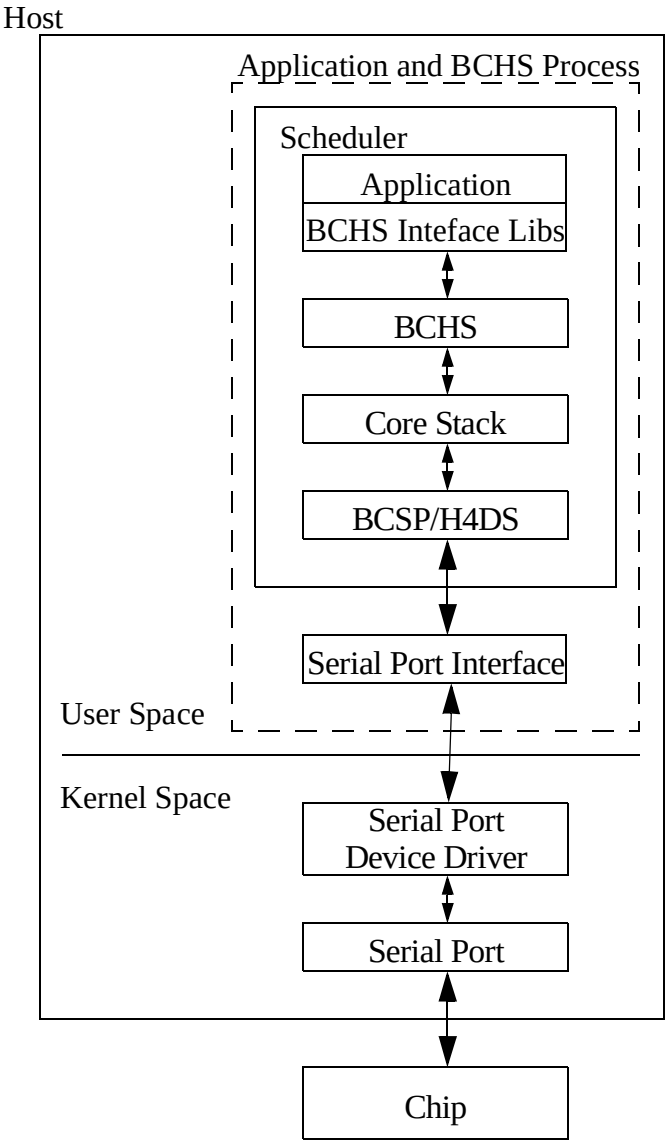


Figure 36-2. BCHS Protocol Stack

36.2.1 UART Control

For user space porting, first configure the universal asynchronous receiver transmitter (UART). On the i.MX platforms, UART2 is used for communication between the CPU chip and the Bluetooth chip. The BCHS protocol opens `/dev/ttymxc1` and configures the device according to profile requirements.

The minimum baud rate for the AD2P profile is 460.8 kbps; 921.6 kbps baud is recommended. Table 36-1 maps the relationship between the UART baud rate and maximum SBC bit rate.

Table 36-1. UART Mapping

Baud Rate (kbps)	Max SBC bit rate (kbps)
115.2	75
230.4	150
460.8	300
600.0	400

The following table describes the UART configuration files.

Table 36-2. Bluetooth UART configuration files

File	Description	Platform
`<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3/board-mx3_3stack.h`	Disable IRDA, Enable UART2	i.MX31
`<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/mach-mx3/mx3_3stack_gpio.c`	Set UART operation mode	i.MX31

36.2.2 Reset and Power control

Besides BCHS and UART, the power control and reset for BT chip is also required. Table 36-3 lists the file for the driver.

Table 36-3. Bluetooth Driver File

File	Description
`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/bt/mxc_bt.c`	bluetooth kernel driver

36.2.3 Configuration

The CONFIG_MXC_BLUETOOTH Linux kernel configuration is provided. This is the configuration option for the bluetooth driver for the MXC processors. In the menuconfig this option is available under Device Drivers > MXC support drivers > MXC Bluetooth support > MXC Bluetooth support. By default, this option is M for all architectures.

Chapter 37
ATA Driver

The ATA module is an AT attachment host interface mainly used to interface with hard disk devices. The ATA driver is compliant to the ATA-6 standard, and supports the following protocols:

- PIO mode 0, 1, 2, 3, and 4
- multi-word DMA mode 0, 1, and 2
- Ultra DMA mode 0, 1, 2, 3, and 4 with bus clocks of 50MHz or higher
- Ultra DMA mode 5 with bus clock of 80MHz or higher
- LibATA interfaces

37.1 Hardware Operation

The detailed hardware operation of ATA is described in the hardware documentation.

37.2 Software Operation

37.2.1 ATA Driver Architecture

Figure 37-1 shows ATA driver architecture. File systems are built upon the block device. The integrated external DMA engine, which assists the ATA controller hardware in the DMA transfer modes, is accessed through the Linux SDMA Driver. The DMA engine used depends on chip capability. See Table 37-1 for detailed information.

Table 37-1. DMA engine

DMA Engine Type	Available Platform
external DMA	MX31

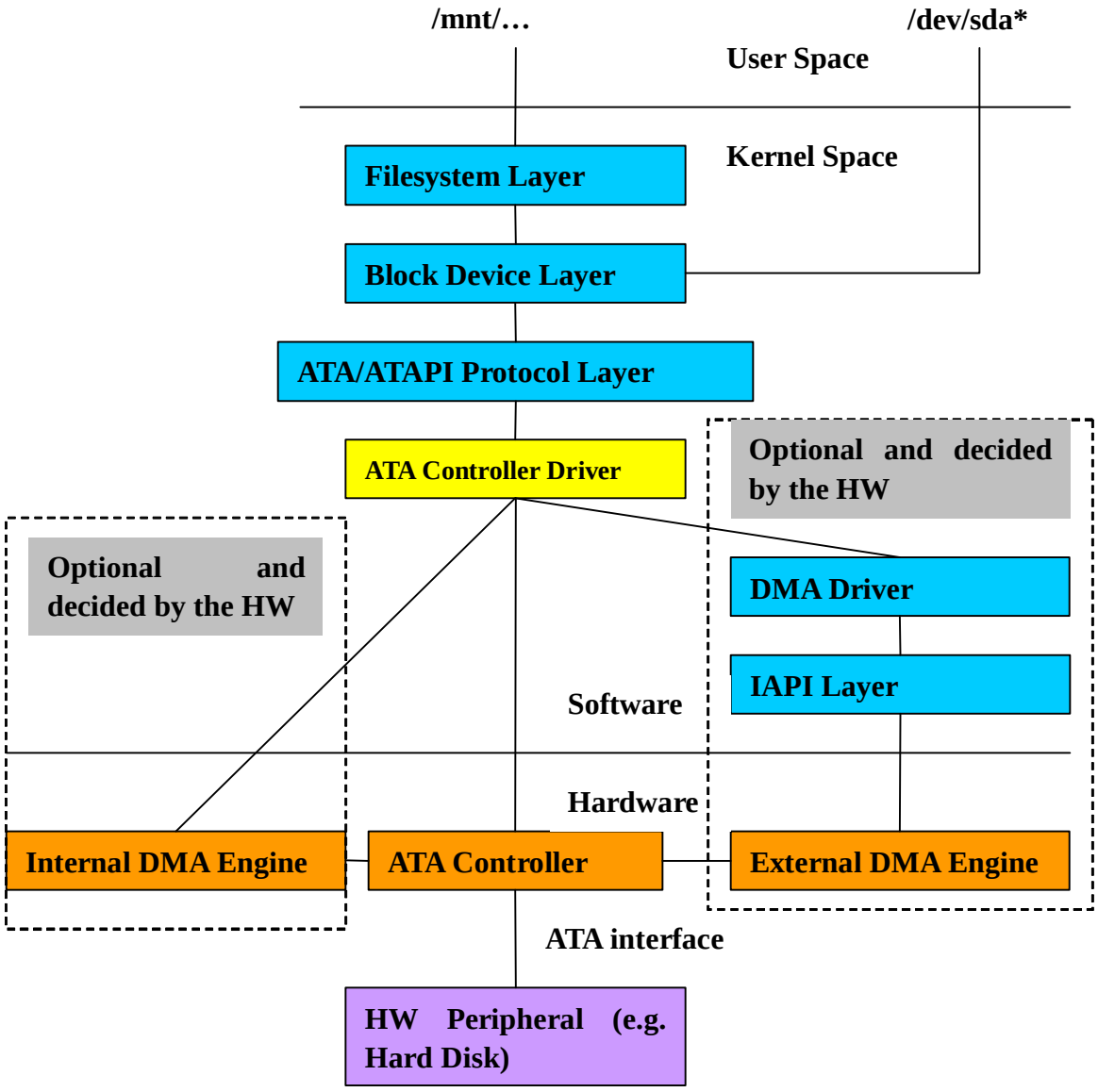


Figure 37-1. ATA Driver Layers

37.2.2 LibATA Driver

LibATA is a library used inside the Linux kernel to support ATA host controllers and devices. libATA provides an ATA driver API, class transports for ATA and ATAPI devices, and SCSI <-> ATA translation for ATA devices according to the T10 SAT specification driver. Hard disk is exposed to the application in user space by the /dev/sda interface.

37.3 Source Code Structure Configuration

37.3.1 LibATADriver

Table 37-2 lists the source file available in the directory,
`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/ata`

Table 37-2. LibATA Driver File List

File	Description
pata_fsl.c	ATA Driver Implementation file

37.4 Linux Menu Configuration Option

Enable these kernel configuration options as either modules (M) or built-in to the kernel (Y). These options are all under “Device Drivers > Serial ATA (prod) and Parallel ATA (experimental) drivers > Freescale on-chip PATA support”:

For ATA device support, enable these options: Device Drivers > SCSI device support > SCSI disk support.

37.5 Board Configuration Options

Table 37-3 lists the hardware configurations for 3-Stack boards:

Table 37-3. Hardware configuration for 3-Stack boards

Platform	Hardware configuration
MX31	• Ensure R189 is removed and R190 is populated • The ATA connector is at the back of the Personality card.

Chapter 38
Real Time Clock (RTC) Driver

Each i.MX processor has an integrated real time clock (RTC) module. The RTC is used to keep the time and date while the system is turned off. The driver can also do the following:

- Provide periodic interrupt at certain frequency (PIE)
- Wake up the system by providing the alarm feature (AIE)

38.1 Hardware Operation

The RTC prescaler converts the incoming crystal reference clock to a 1 Hz signal, which is used to increment seconds, minutes, hours, and days Time-Of-Day (TOD) counters. The alarm functions, when enabled, generate RTC interrupts when the TOD settings reach programmed values. The sampling timer generates fixed-frequency interrupts, and the minutes stopwatch allows efficient interrupts on minute boundaries.

38.2 Software Operation

The RTC module's software implementation is through a RTC driver. Besides the initialization function, it provides `ioctl` functions to set up the RTC timer, interrupt, and so on. The periodic interrupt is supported at fixed frequencies of 2 Hz, 4 Hz, 8 Hz, 16 Hz, 32 Hz, 64 Hz, 128 Hz, 256 Hz, and 512 Hz given the clock input of 32.768 kHz (Other clock input frequencies are not supported by the driver.) The 1 Hz periodic interrupt is also called update interrupt (UIE).

NOTE

The i.MX RTC driver implementation follows what is stated in the <ltib_dir>/rpm/BUILD/linux-2.6.26/Documentation/`rtc.txt` file under Linux kernel `Documentation` directory that "Programming and/or enabling interrupt frequencies greater than 64 Hz is only allowed by root."

38.3 Requirements

This RTC implementation meets the following requirements:

- The RTC module implements all the functions required by Linux to provide the real time clock, alarm interrupt and periodic interrupt.
- The RTC module conforms to the Linux coding standard as documented in the *Coding Conventions* chapter.

38.4 Source Code Structure

Table 38-1 shows the RTC module files.

Table 38-1. RTC Driver File List

File	Description
rtc-mxc.c	rtc driver implementation file

The source file, `mxc_rtc.c`, for the RTC specifies the RTC function implementations.

38.5 Programming Interface

All the Linux RTC functions are implemented in the `time.c` file.

NOTE

The `include/linux/rtc.h` file specifies all the ioctls for RTC.

The following RTC ioctls are supported on i.MX platforms.

```
column (1) = IOCTLs listed in include/linux/rtc.h
column (2) = Supported by MXC platform RTC driver. “Y” means supported.

              (1)   (2)
RTC_UIE_ON       Y     Y
RTC_UIE_OFF      Y     Y
RTC_RD_TIME      Y     Y
RTC_SET_TIME     Y     Y
RTC_ALM_READ     Y     Y
RTC_ALM_SET      Y     Y
RTC_WKALM_RD     Y     Y
RTC_WKALM_SET    Y     Y
RTC_AIE_ON       Y     Y
RTC_AIE_OFF      Y     Y
RTC_WIE_ON       Y     -
RTC_WIE_OFF      Y     -
RTC_IRQP_READ    Y     Y
RTC_IRQP_SET     Y     Y
RTC_PIE_ON       Y     Y
RTC_PIE_OFF      Y     Y
RTC_EPOCH_READ   Y     Y
RTC_EPOCH_SET    Y     -
RTC_PLL_GET      Y     -
RTC_PLL_SET      Y     -
```

See the API documentation for the detailed programming interface.

Chapter 39
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.

39.1 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, WDOG times out. Upon a time-out, WDOG either asserts the `wdog_b` signal or a `wdog_rst_b` system reset signal, depending on software configuration. The watchdog module can not be deactivated once it is activated.

39.2 Software Operation

The Linux OS has a standard WDOG interface that allows a WDOG driver for a specific platform to be supported. For the platforms that have two WDOG hardware modules, another implementation is done in the machine-specific layer as part of the `time.c` file. The following sections describe both implementations.

WDOG can be suspended/resumed in STOP/DOZE and WAIT modes independently.

39.2.1 Generic WDOG driver

This is implemented in the `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/watchdog/mxc_wdt.c` file. It essentially provides functions for various IOCTLs and read/write calls from the user level program to control the WDOG.

39.2.1.1 Requirements

This WDOG implementation meets the following requirements:

- Generates the reset signal if it is enabled but not serviced within a predefined timeout value.
- 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.

39.2.1.2 Source Code Structure

The WDOG source code is in `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/watchdog/mxc_wdt.c` and `mxc_wdt.h`.

Table 39-1 lists the source files for WDOG.

Table 39-1. WDOG File List

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

39.2.1.3 Programming Interface

For more information, see the API documentation for the detailed programming interface.

39.2.2 WDOG under Machine Specific Layer

The WDOG software implementation provides routines to service WDOG so that the timeout never occurs. If the WDOG timer is enabled before the Linux kernel boots (enabled by boot loader or ROM) it is automatically serviced, with the service interval being configurable. In addition, compile-time options specify if the Linux kernel should enable the watchdog, and if so the parameters to be used. If the second WDOG presents (it is used to generate an interrupt after the timeout occurs), the highest interrupt priority (16) is assigned to this interrupt

Figure 39-1 shows the flow chart for the operation. It applies to all platforms with two WDOGs.

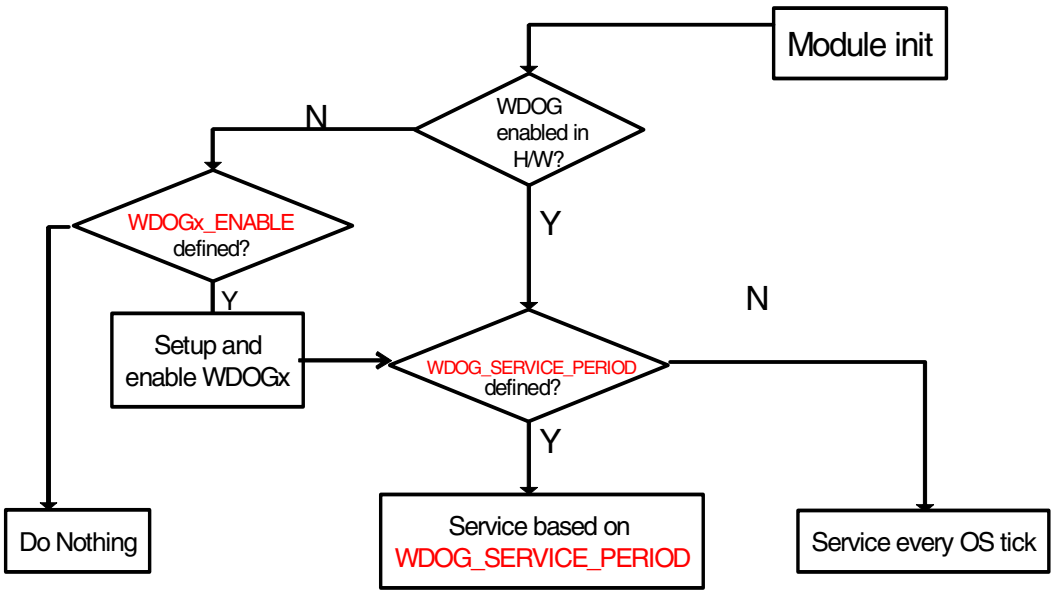


Figure 39-1. WDOG Software Operation Flow Chart

39.2.2.1 Requirements

This WDOG implementation meets the following requirements:

- Generates the reset signal if it is enabled but not serviced within a predefined timeout value.
- Does not generate the reset signal if it is serviced within a predefined timeout value.

- The second WDOG (when present) generates an interrupt if it is enabled but not serviced within a predefined timeout value.

39.2.2.2 Source Code Structure

The WDOG module implementation is embedded inside the timer module as described above. The source code is available in the `time.c` file under the MSL directory
`<ltib_dir>/rpm/BUILD/linux-2.6.26/arch/arm/plat-mxc`.

The source files for WDOG is `time.c`, which specifies WDOG function implementations.

39.2.2.3 Programming Interface

The following DEFINES are provided:

```
WDOG1_ENABLE            /* not defined by default */
WDOG2_ENABLE            /* not defined by default */
WDOG1_TIMEOUT           /* WDOG1 timeout in ms */
WDOG2_TIMEOUT           /* WDOG2 timeout in ms */
WDOG_SERVICE_PERIOD     /* time interval in ms to service WDOG */
```

Chapter 40
FM Driver

Si4702 is used as the FM chip on the board. The Si4702 extends Silicon Laboratories Si4700 FM tuner family and further increases the ease and attractiveness of adding FM radio reception to mobile devices through small size and board area, minimum component count, flexible programmability. Headset cable is used for antenna on the board.

40.1 FM Overview

The device offers significant programmability and caters to the subjective nature of FM listeners and variable FM broadcast environments world-wide through a simplified programming interface and mature functionality.

Power management is also simplified with an integrated regulator allowing direct connection to a 2.7 to 5.5 V battery. The features of the FM module are as follows:

- Worldwide FM band support (76-108 MHz)
- Digital low-IF receiver
- Seek tuning
- Automatic frequency control (AFC)
- Automatic gain control (AGC)
- Signal strength measurement
- Adaptive noise suppression
- Volume control
- 32.768 kHz reference clock
- 2-wire and 3-wire control interface
- 2.7 to 5.5 V supply voltage
- Integrated LDO regulator allows direct connection to battery
- Integrated crystal oscillator

40.1.1 Hardware Operation

Si4702 supports both three-wire control and two-wire control. Two-wire control is chosen by driving SEN pin high during boot up.

For two-wire operation, a transfer begins with the START condition. The control word is latched internally on rising SCLK edges and is eight bits in length: a seven bit device address equal to 0010000b and a read/write bit (write = 0 and read = 1). The device acknowledges the address by setting SDIO low on the next falling SCLK edge.

For write operations, the device acknowledge is followed by an eight bit data word latched internally on rising edges of SCLK. The device always acknowledges the data by setting SDIO low on the next falling SCLK edge. An internal address counter automatically increments to allow continuous data byte writes, starting with the upper byte of register 02h, followed by the lower byte of register 02h, and onward until the lower byte of the last register is reached. The internal address counter then automatically wraps around to the upper byte of register 00h and proceeds from there until continuous writes cease. Data transfer ceases with the STOP command. After every STOP command, the internal address counter is reset.

For read operations, the device acknowledge is followed by an eight bit data word shifted out on falling SCLK edges. An internal address counter automatically increments to allow continuous data byte reads, starting with the upper byte of register 0Ah, followed by the lower byte of register 0Ah, and onward until the lower byte of the last register is reached. The internal address counter then automatically wraps around to the upper byte of register 00h and proceeds from there until continuous reads cease. After each byte of data is read, the controller IC should return an acknowledge if an additional byte of data is requested. Data transfer ceases with the STOP command. After every STOP command, the internal address counter is reset.

FM analog signals connect directly to the audio chip which routes them out to headset.

40.1.2 Software Operation

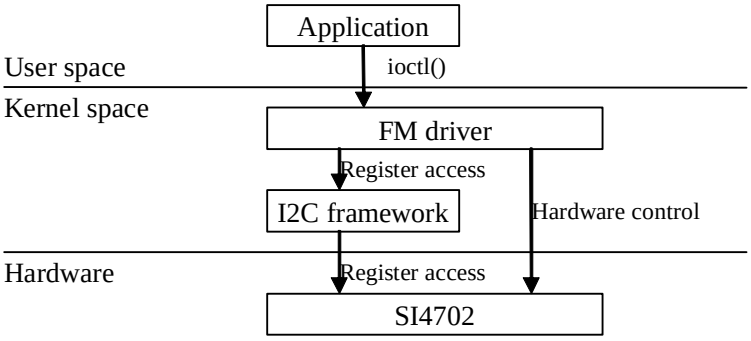


Figure 40-1. Software Operation

The FM driver serves as an interface between kernel and user space. The driver can control hardware directly (for example, a reset operation) but most of the functional operation comes from the I^2C framework, which is especially convenient in the 2-wire control case.

The main software operation is as follows:

1. In initialization stage, register device in character sub-system, and then register it to I^2C framework.
2. In open operation, reset the chip, and initialize the register on the chip.
3. In release operation, shutdown the chip.
4. In ioctl operation, handle all the commands from user space, execute them and then feed back information if there is any.

40.2 Source Code Structure Configuration

Table 40-1 lists the source files associated with the FM driver that are available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/char/`.

Table 40-1. FM Driver Source and Header File List

File	Description
`mxc_si4702.c`	Source file for SI4702 FM driver
`<ltib_dir>/rpm/BUILD/linux-2.6.26/include/linux/mxc_si4702.h`	Header file for SI4702 FM driver

40.3 Linux Menu Configuration Options

The Linux kernel configurations are provided for this module. CONFIG_FM_SI4702 is the configuration option for the FM driver. By default, this option is M.

To load the FM drivers use the command:

```
insmod mxc_si4702.ko
```

It is located in `/lib/modules/2.6.26-*/kernel/drivers/char`.

Chapter 41
MMA7450L Accelerometer Driver

The MMA7450L is a feature rich accelerometer device with a flexible programming interface exposed to the software. It can be used on many applications, such as image stability, freefall detection, motion dialing, e-compass, and so forth.

41.1 MMA7450L Features

- Digital Output (I2C/SPI) - 10-Bit at 8g Mode
- 3mm x 5mm x 1mm LGA-14 Package
- Low Current Consumption: 400 µA
- Self Test for Z-Axis
- Low Voltage Operation: 2.4 V - 3.6 V
- Customer Assigned Registers for Offset Calibration
- Programmable Threshold Interrupt Output
- Level/Pulse Detection for Motion Recognition (shock, vibration, freefall)
- Click Detection for Single or Double Click Recognition
- High Sensitivity (64 LSB/g at 2g and at 8g in 10-Bit Mode)
- Selectable Sensitivity (±2g, ±4g, ±8g)
- Self Test for Z-Axis
- Robust Design, High Shocks Survivability (10,000 g)
- RoHS Compliant
- Environmentally Preferred Product
- Low Cost

41.2 Driver Requirements

MMA7450L driver is based on a I^2C driver and makes use of hardware monitor system and input poll device system; therefore, the user must enable this support in the Linux kernel.

41.3 Driver Architecture

Figure 41-1 shows the software architecture. The MMA7450L provides two methods of register access, I^2C and SPI. This driver uses the I^2C. At driver initial phase, a I^2C client is registered to the I^2C system and is used during the process of MMA7450L operations. The MMA7450L registers itself to the hardware monitor system and the input poll device system that provide user access facilities.

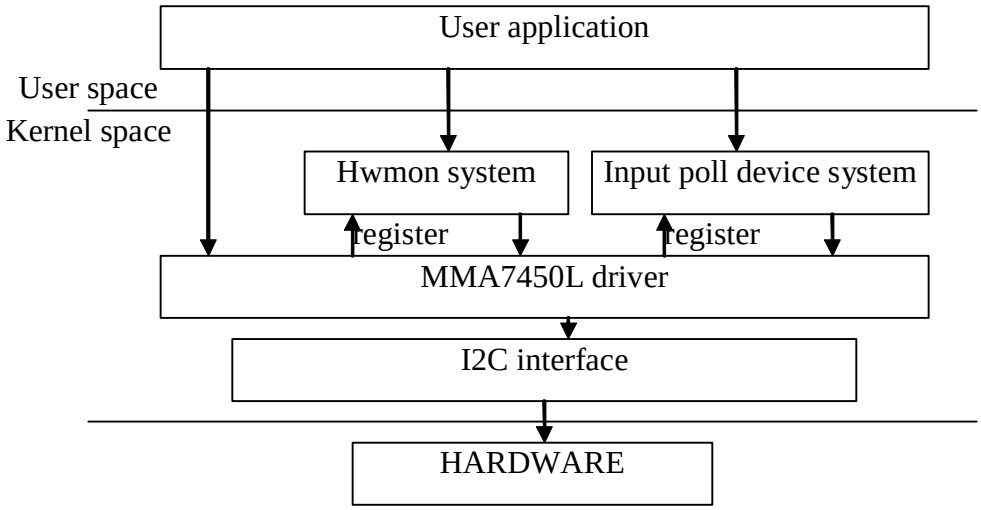


Figure 41-1. Driver Architecture

41.4 Driver Source Code Structure

The driver source code structure contains only one file located in the directory:

`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/hwmon/`

Table 41-1. Driver Source Code Structure File

File	Description
mxc_mma7450.c	Implementation of the mma7450 accelerometer driver.

41.5 Driver Configuration

To get to the MMA7450 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 MMA7450 driver:

- Device Drivers > Hardware Monitoring support > MMA7450 device driver

Chapter 42
Global Positioning System (GPS) Driver

42.1 GPS Driver Overview

An external global positioning system (GPS) module can be supported through the serial port and necessary GPIO resources. Currently, Broadcom's Barracuda Single Chip A-GPS Solution is supported. Since this chip set features a host-based architecture, several software components need to be loaded on the platform to enable full operation. Figure 42-1 shows a coarse block diagram of the complete GPS system architecture consisting of the BCM4750 Barracuda GPS IC and the host CPU.

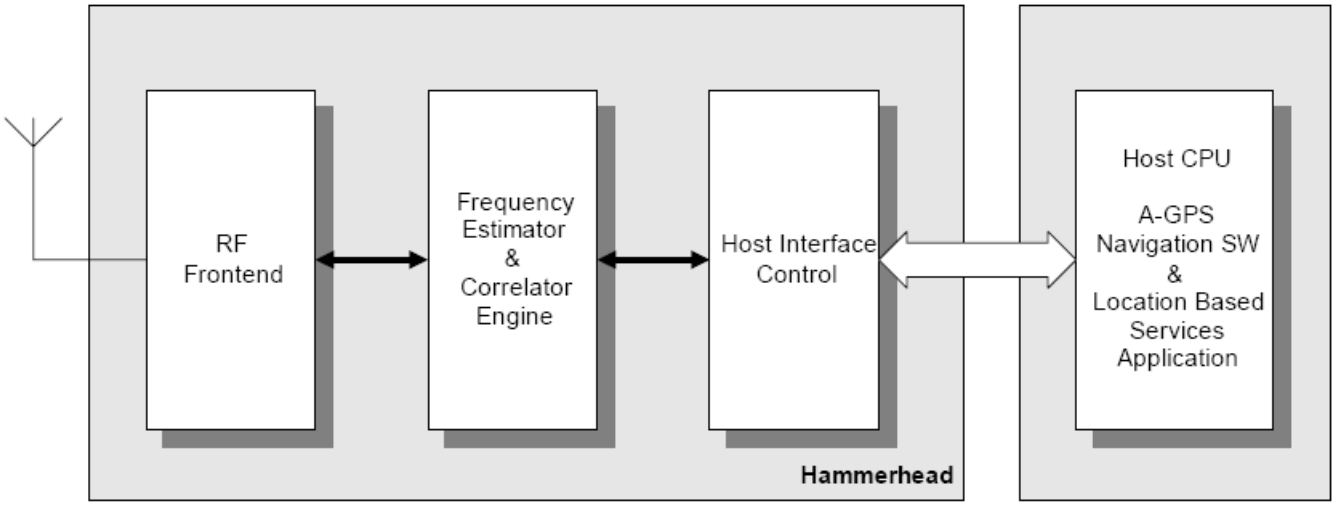


Figure 42-1. Barracuda GPS Coarse System Architecture including Host CPU

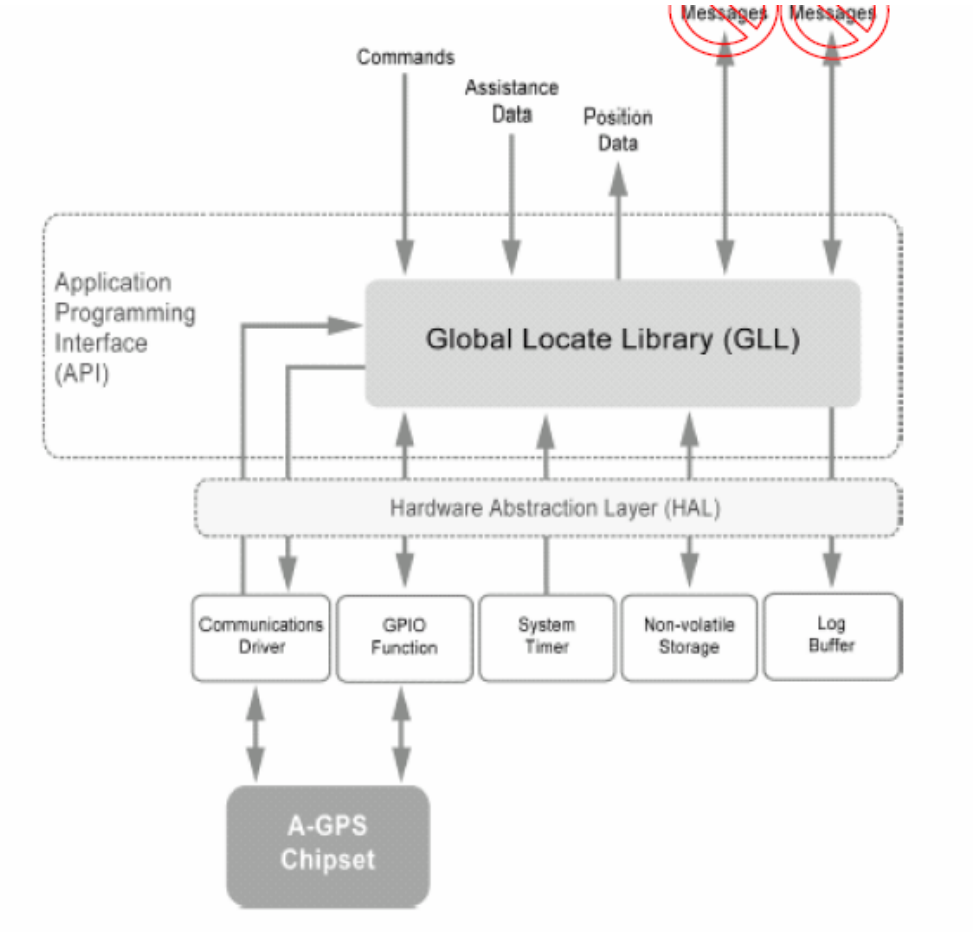


Figure 42-2. GL GPS SW Architecture

Figure 42-2 shows the GPS software architecture on the host. The communications driver provides a serial interface to the core driver and communicates with the external GPS chip set. In the current platform, the UART acts as this serial interface. The GPIO function controls the power and reset functions of the GPS chip set. The system timer, that is the GPT timer, provides an accurate timer to the GPS core driver. Non-volatile storage is used to store assistance data and useful information for the last GPS position information received, which can accelerate the next position fix. A log buffer is generated by the GLL lib and tracks problems when integrating the GL GPS solution. The hardware abstraction layer (HAL) is used to abstract specific hardware platforms, which makes the GPS core driver hardware-independent. The core driver processes GPS data and controls the work flow with the GPS device. It accepts application requests and sends the GPS position information to the upper layer.

Most of the GPS software modules are provided in binary format only. BSP source code is available for the driver that handles the concrete hardware access, additional source code controls the GPS data flow when GPS is running.

The GPS module uses UART3 on the i.MX31 PDK platform. Reset and power on/power off to the GPS module are controlled with the GPIO pins of the i.MX31.

Generally, the functionality of the GPS module is segmented into two parts: control driver and core driver. The control driver is mainly for hardware-specific IO settings. The core driver (glgps_freescaleLinux) manages the GPS system, gets and parses data from the Barracuda chip, and generates position data and sends it to a pipe which connects to the application.

42.2 Hardware Operation

The GPS daughter board connects to the UARTx slot of the platform hardware. Since GPS is very sensitive to timing accuracy, the platform provides a high-accuracy, non-drifting millisecond timer function to the GPS core drivers.

The GPS Control driver manages hardware-specific IOs to regulate power usage on the platform, power on/off and reset GPIO pins setting when the GPS is supported. Before GPS power up, the reset/init sequence is done. The reset pin is asserted for a few milliseconds, then de-asserted. The reset sequence is complete when the gps_gpiodrv.ko is loaded. When GPS is launched, the power pin is asserted.

42.2.1 UART Port

The GPS module uses UARTx to communicate with the host, the device name in linux system is shown in Table 42-1.

Table 42-1. UART Port

Platform	UART	Device Name
i.MX31	UART3	/dev/ttymxc2

42.2.2 GPIO Control

GPIO pins are used to control the GPS module as shown in Table 42-2.

Table 42-2. GPIO Control Signals

GPIO Name PIN	Value	Description
MX31_PIN_DCD_DTE1	MCU2_15	0: Reset of GPS module is asserted 1: Reset of GPS module is de-asserted
MX31_PIN_SCLK0	MCU3_2	1: GPS module is power on 0: GPS module is power off

42.2.3 Hardware Dependent Parameters

The TCXO clock is a hardware parameter that must be set in the XML file in order for the GPS to work properly.

Table 42-3. Hardware Dependent Parameters

Parameter	Value Description
Frequency Plan:	The TCXO has to be accurate +/- 2.0 ppm. The number after "FRQ_PLAN_" describes the type of TCXO used, for example, FRQ_PLAN_13MHZ_2PPM is a 13MHz reference clock. FRQ_PLAN_13MHZ_2PPM FRQ_PLAN_16_8MHZ_2PPM FRQ_PLAN_26MHZ_2PPM FRQ_PLAN_10MHZ_2PPM_10MHZ_50PPB FRQ_PLAN_20000_2PPM_13MHZ_50PPB FRQ_PLAN_27456_2PPM_26MHZ_50PPB FRQ_PLAN_33600_2PPM_26MHZ_50PPB FRQ_PLAN_19200_2PPM_26MHZ_100PPB

42.3 Software Operation

Broadcom provides several software components to drive the GPS hardware. Broadcom's software architecture allows the LTO feature to be enabled or disabled as needed, which is a way to get the assistance data shown in Table 42-2.

Software applications communicate with the GPS module through a NMEA pipe where the incoming NMEA data is read. The GPS application creates a NMEA pipe then receives the NMEA sentences from this pipe.

42.3.1 GLGPS Configuration

The GLGPS reads configuration information from an XML file. This configuration file defines the hardware dependent settings, paths, and different tasks.

Example 42-1 shows the XML file.

Example 42-1. GLGPS Configuration

```
<?xml version=”1.0” encoding=”utf-8”?>
<glgps xmlns=”http://www.glpals.com/” xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xsi:schemaLocation=”http://www.glpals.com/ glconfig.xsd” >

  <!--HAL Confguration  -->
  <hal acPortName=”/dev/ttymxc2” lBaudRate=”115200” cLogEnabled=”true” acLogDirectory=”./log”
ltoFileName=”lto.dat”
       bPrintToConsole=”false” />

  <!-- Parameters passed to GlEngine   -->
  <gll FrqPlan=”FRQ_PLAN_26MHZ_2PPM” RfType=”GL_RF_BARRACUDA”
       LogPriMask=”LOG_DEBUG”
      LogFacMask=”LOG_GLLAPI | LOG_DEVIA | LOG_NMEA | LOG_RAWDATA “
  />
```

```
  <!-- List of jobs can be performed by the GPS controller    -->
      <!-- The default job  all parameters are set to default values  -->
    <job id=”normal”>
      <task>
          <req_pos />
      </task>
    </job>

    <job id=”cold”>
      <task>
            <!-- Instructs GLL to ignore all elements stored in NVRAM listed below -->
            <startup ignore_time=”true” ignore_osc=”true” ignore_pos=”true” ignore_nav=”true”
ignore_ram_alm=”true”  />
          <req_pos />
      </task>
    </job>
</glgps>
```

The <hal> tag defines the glhal settings (serial COM settings, enable hal log, some other paths). The parameters are explained in Table 42-4.

Table 42-4. hal Attributes

hal Attributes	Description	Comment
acPortName	serial COM port	in mx31 it is /dev/ttymxc2
lBaudRate	baud rate	9600,19200,38400,57600,115200
acLogDirectory	directory where the logs are placed	

The <gll> tag defines the gll specific parameters, as explained in Table 42-5.

Table 42-5. gll Attributes

gll attributes	Description	Comment
FrqPlan	type of TCXO	For GPS/B it is FRQ_PLAN_26MHZ_2PPM
RfType	Type of rf chip	it should be GL_RF_BARRACUDA

Different tasks are configured in Example 42-1. The task named “*normal*” makes an autonomous fix every second and the task named “cold” starts autonomous fixes with no assisted data.

42.3.2 Driver Configuration

The GPS GPIO control driver is configured using the same mechanisms that are provided to configure the Linux kernel image. That is, a Kconfig file within the source files directory is used to select whether or not the device driver is to be included in the kernel build process and whether it is to be built as a loadable kernel module or not. Any of the standard kernel configuration tools, such as menuconfig, can be used to select and configure the GPS control driver.

42.3.2.1 Linux Menu Configuration Options

To get to the GPIO device 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 GPS device driver:

- Device Drivers > MXC Support Drivers > Broadcom GPS ioctrl support > GPS ioctrl support

The CONFIG_GPS_IOCTRL Linux kernel configuration option is provided for the GPS GPIO control driver. It is the build option for GPS GPIO control driver support. By default, this option is M for all architectures.

42.3.3 Source Code

Table 42-6 lists the source files available in the source directory

`<ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/gps_ioctrl`

Table 42-6. GPS Driver Source Code

File	Description
agpsgpiodev.c	Main file for GPIO kernel module
agpsgpiodev.h	head file of Simple character device interface for AGPS kernel module

42.3.4 LTO Feature (Optional)

The Long Term Orbit (LTO) feature is a way to get assistance data for the device. The assistance data is valid for several days, giving users the advantages of assisted GPS performance while preserving the freedom of autonomous GPS operation.

The reference board should be connected through a NFS server. Put the LTO file in the folder pointed to by the GPS driver. It is the responsibility of the user to implement the method used to get a TCP/IP connection on the platform (Bluetooth, IP over USB).

42.3.4.1 Enabling The LTO Feature on the Platform

Contact Broadcom in order to obtain the necessary license for enabling this feature on the Freescale reference platform. Submit the following information to obtain the license: platform name and platform type. Also, ensure that the LTO feature is enabled through the different input parameters of the GPS driver.

42.3.5 Power Management

The GPS driver automatically manages the power management of the Broadcom chip set by toggling the different IOs of the chip set. The GPS chip set enters low power standby mode when the GPS driver is stopped.

42.3.6 irm Commands

While the GLGPS is running, commands can be sent to the process. There is a special FIFO file in `/var/run/glgpsctrl` where the commands are sent. Commands are ASCII strings with the '`$pglirm, prefix + command name`' format.

- `$pglirm,req_pos,name,period,N1,fixcount,N2,validfix,N3,duration_sec,N4`

  Creates a periodic position request.
  - Name – Unique request identifier, it also used by GLHAL to output NMEA data
  - Period – Update period in milliseconds
  - Fixcount – Stop after that number of fixes (valid or invalid) reported
  - Validfix – Stop after that number of valid fixes reported
  - Duration_sec – Stop after that many seconds
- `$pglirm,req_pos_single,name,acc,timeout,N1`

  Creates a single shot request.
  - Name – Unique request identifier, it also used by GLHAL to output NMEA data
  - Acc – Accuracy QoS parameter
  - Timeout – Time-out QoS parameter
- `$pglirm,req_aid,name`

  Queries for what assistance data is missing.
  - Name – Unique request identifier, it also used by GLHAL to output NMEA data
- `$pglirm,factory,test,prn,17,timeout,16,GL_FACT_TEST_MODE,GL_FACT_TEST_CONT,GL_FACT_TEST_ITEMS,GL_FACT_TEST_WER`

  Creates a factory request.
  - Test – Unique request identifier, it also used by GLHAL to output NMEA data
  - Prn – PRN number
  - Timeout – How long to run test for
  - GL_FACT_TEST_MODE – Test mode [GL_FACT_TEST_ONCE|GL_FACT_TEST_CONT]
  - GL_FACT_TEST_ITEMS – What to test GL_FACT_TEST_CN0|GL_FACT_TEST_FRQ|GL_FACT_TEST_WER
- `$pglirm,startup,{[ignore_osc|ignore_rom_alm|ignore_rom_alm|ignore_pos|ignore_ram_alm|ignore_time],[true|false]}`

  Tells the GLCT to ignore specified elements of the data previously stored in nonvolatile storage.
- `$pglirm,stop,name`

  Stops an ongoing request with a name "name"; The name "all" is reserved to stop all ongoing requests.
- `$pglirm,quit`

  Causes GLCT to exit.
- `$pglirm,pwm,[on|off]`

Turns auto power management on or off

As an example, to send quit command in a shell, do the following:

```
echo '$pglirm,quit' >/var/run/glgpsctrl
```

Chapter 43
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 and a daemon for collecting sample data, and several post-profiling tools for turning data into information.

43.1 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.

43.2 Features

- Unobtrusive: No special recompilations, wrapper libraries or the like 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 with 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%, dependent on 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: OProfile works against almost any 2.2, 2.4 and 2.6 kernels, and works on ARM11 based platform.

43.3 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 of binary-image, offset. This is something that the

userspace tools can use directly to reconstruct where the code came from, including the particular assembly instructions, symbol, and source line (through the binary's debug information if present).

Regularly sampling the PC value like this approximates what actually was executed and how often - 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.

ARM11 CPU provides hardware performance counters capable of measuring these events on 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 of the given event.

The ARM1136 Platform Event Monitor (EVTMON) is a 32-bit IP-bus peripheral used for monitoring the Level 2 Cache Controller (L2CC) events through the L2CC event bus interface. The EVTMON contains six event counters (EMC0-EMC5) which may be used to count six different events selected from a list of ten possible external events (through the L2CC event bus) or seven internal events (overflows or clock edges).

43.4 Software Operation

43.4.1 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 `arch/<arch>/oprofile/` directory. The architecture-specific implementation works 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.

43.4.2 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.

43.4.3 Generic Kernel Driver

The generic kernel driver resides in `drivers/oprofile/`, and forms the core of how OProfile works 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 form as described later) until releasing the data to the userspace daemon through the `/dev/oprofile/buffer` character device.

43.4.4 The 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). So the post-profiling tools can run on this data at any time (assuming the original binary files are still available and unchanged, naturally).

43.4.5 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.

43.5 Requirements

- User must add Oprofile support with ARM11 PMU events for i.MX platforms:
  Enter to the LTIB configuration screen
  <ltib dir>/.ltib -c
  Package List > Oprofile
- User must add ARM11 L2 Cache EVTMON support to oprofile for i.MX platforms:
  Go to the Kernel menuconfig
  <ltib dir>/.ltib -c
  Configure the kernel
  General Setup > Oprofile
- User must add Oprofile timer interrupt mode support

43.6 Source Code Structure

Table 43-1. Source Code Structure

File	Description
op_model_arm11.c	Source file with the implementations required to support HW performance counters on both PMU and EVTMON.
common.c	Source file with the implementation required for all platforms.
op_arm_model.h	Header File with all the register and bit definitions.

43.7 Configuration

This module has the following Linux menu configuration options.

43.7.1 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module. In order to get to the Oprofile configuration, use the command `./ltib -c` from the <`ltib dir`>. On the screen, select **Configure Kernel**, then exit, and a new screen appears.

CONFIG_OPROFILE is the configuration option for the oprofile driver. In the `menuconfig` this option is available under General Setup > OProfile system profiling. By default it is Y for all i.MX platforms.

43.8 Programming Interface

This driver implements all the methods required to configure and control PMU and L2 cache EVTMON counters.

43.9 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.

43.10 Device Specific Information

For ARM11 PMU support:

For ARM11 cores with rev r0p2 to configure PMU event numbers 0x7 (Instruction count) or 0x22 (event of ETMEXTOUT[0] & ETMEXTOUT[1]), configure the cycle count event with the same event count. Also, the minimum event count for cycle count and increment cycle events must be 5000.

Chapter 44
Frequently Asked Questions

44.1 Downloading a File

There are various ways to download files on to a Linux system. The following procedure gives instructions how to do this through 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 hyperterminal. For more information about how to set this up, see the tools section.
2. Make sure Linux boots to the serial prompt and log in using “root”
3. Type “`rz`” under serial prompt at `/mnt/ramfs/root`
4. Under hyperterminal, click on “`Transfer > Send File > Browse.. ->`”, then go to the right directory with the file you want to download. 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, lrzsz package is required.

Another way for file transfer 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.

44.2 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. Note that 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 that you want to have inside the JFFS2 file system into the `jffs2` directory. Now 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 4MB. The output file is `fs.jffs2`.

   NOTE

   - Make sure the `fs.jffs2` file is within this size limit of 4MB.
   - 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 un-used 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"
```

   So the jffs2 is the fourth MTD partition under Linux in this case. To mount this MTD partition after booting Linux, you can do:

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

This mounts `/dev/mtdblock/3` to `/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 has been described above.

44.3 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
```

   Install NFS RPM if not already installed.

3. To boot with NFS mounted fs, under RedBoot, do the following:

```
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's IP address is `1.1.1.1`. This needs to be modified in the command line you actually use.

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 r/w disk space. However, this causes problems for people 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.

44.4 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.

44.5 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 it may happen that 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_ND_LOW_LEVEL_ERASE definition in the mxc_nd.c file. This produces an MXC NAND driver, which upon loading, erases the whole 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.

44.6 How to Use 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 argument 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.

44.7 How to Make Software Workable when JTAG is Attached

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

44.8 How to Use the Hardware Event Kernel Module

The hardware event kernel module is designed to provide a uniform interface to enable the kernel to notify the user space of hardware events, such as headset and AV line insert/remove events. This module takes advantage of netlink feature in the kernel network subsystem to multicast the event to the user applications who are interested in the event, creating netlink socket and adding itself to the multicast group. In the i.MX31 3-Stack kernel, the phone jack detect (headset/CVBS) module, battery event module and power key detect module are using the HW event feature and calling the hw_event_send() interface to send out the event.

44.8.1 Source File

Table 44-1 lists the source files associated with the Hardware Event kernel module. They are available in the directory <ltib_dir>/rpm/BUILD/linux-2.6.26/drivers/mxc/hw_event/.

Table 44-1. HW Event Module Source File List

File	Description
mxc_hw_event.c	Source file for Hardware event kernel module. Provide an event sending interface to kernel and drivers.
include/asm-arm/arch-mxc/hw_events.h	Header file for Hardware event kernel module.

44.8.2 API

The HW event module exports one API to the kernel and drivers:

```
int hw_event_send(int priority, struct mxc_hw_event *new_event);
```

It sends a new event according to the priority. Current implementation only has two priorities: HWE_HIGH_PRIORITY and HWE_DEF_PRIORITY. With the former priority, the event can be sent immediately; but with the latter, it is queued and sent later.

The mxc_hw_event structure is defined as below:

```
struct mxc_hw_event
{
        unsigned int event;
        int args;
};
```

In the structure *event* is an ID to identify the Hardware Event, and *args* is an extra information for this event. Events supported by i.MX31 3-Stack BSP are listed as follows:

Table 44-2. List of Event IDs

Event	Event id	Event arg
Low Battery	HWE_BAT_BATTERY_LOW	N/A
Charger Full	HWE_BAT_CHARGER_FULL	N/A
Charger Overvoltage	HWE_BAT_CHARGER_OVERVOLTAGE	N/A
Charger Plug	HWE_BAT_CHARGER_PLUG	UNPLUG/PLUGGED
Power Failed	HWE_BAT_POWER_FAILED	N/A
Phonejack	HWE_PHONEJACK_PLUG	PJT_NONE/PJT_CVBS/PJT_HEADSET
Power Key	HWE_POWER_KEY	PWRK_PRESS/PWRK_UNPRESS

44.8.3 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

CONFIG_NET: Netlink is enabled by this option.

CONFIG_MXC_HWEVENT: In the `menuconfig` this option is available under “Device Drivers->MXC support drivers->MXC HARDWARE EVENT->MXC Hardware Event Handler”. By default, this option is “Y” for i.MX31.

44.8.4 User Application

The user application that is interested in hardware event creates a socket with PF_NETLINK protocol family (domain) and NETLINK_USERSOCK protocol type. Then binds on this socket to receive mxc_hw_event structure message.