i.MX27 PDK 1.0 Linux

Reference Manual

Document Number: 926-78094
Rev. 1.0
11/2008

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

Contents

Chapter 4
Machine Specific Layer (MSL)

Chapter 5
Direct Memory Access Controller (DMAC) API

Chapter 6
PMIC Protocol Driver

Chapter 7
PMIC Audio Driver

Chapter 8
PMIC Digitizer Driver

Chapter 9
PMIC Power Management Driver

Chapter 10
PMIC Connectivity Driver

Chapter 11
PMIC Battery Driver

Chapter 12
PMIC Light Driver

Chapter 13
PMIC Real Time Clock (RTC)

Chapter 14
i.MX27 Low-Level Power Management Driver

Chapter 15
Dynamic Frequency Scaling Driver (CPUFreq)

Chapter 16
CH7024 TV Encoder (TV-Out) Driver

Chapter 17
Video Processing Unit (VPU) Driver for MX27

Chapter 18
OmniVision Camera Driver (OV2640)

Chapter 19
Advanced Linux Sound Architecture (ALSA) Sound Driver with PMIC Hardware Support

Chapter 20
Digital Audio Multiplexer (AUDMUX) Driver

Chapter 21
Synchronous Serial Interface (SSI) Driver

Chapter 22
NAND Flash Memory Technology Device (MTD) Driver

Chapter 23
Low-Level Keypad Driver

Chapter 24
SMSC LAN9217 Ethernet Driver

Chapter 25
Fast Ethernet Controller (FEC) Driver

Chapter 26
Symmetric/Asymmetric Hashing and Random Accelerator (SAHARA) Drivers

Chapter 27
Inter-IC (I2C) Driver

Chapter 28
I2C Slave Driver

Chapter 29
Configurable Serial Peripheral Interface (CSPI) Driver

Chapter 30
MMC/SD/SDIO Host Driver

Chapter 31
Universal Asynchronous Receiver/Transmitter (UART) Driver

Chapter 32
16C652 Universal Asynchronous Receiver/Transmitter (UART) Driver

Chapter 33
ARC USB driver

Chapter 34
ATA Driver

Chapter 35
Real Time Clock (RTC) Driver

Chapter 36
Watchdog (WDOG) Driver

Chapter 37
FM Driver

Chapter 38
MMA7450L Accelerometer Driver

Chapter 39
Global Positioning System (GPS) Driver

Chapter 40
Unit Tests

Figures

Tables

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 3-Stack 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.
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.

Definitions and Acronyms (Continued)

Term	Definition
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 of 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.
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.

Definitions and Acronyms (Continued)

Term	Definition
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 offer 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.
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.

Definitions and Acronyms (Continued)

Term	Definition
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.MX27 PDK Linux User's Guide*
- *i.MX27 PDK Hardware User's Guide*
- *MCIMX27 Multimedia Applications Processors Reference Manual*, (MCIMX27RM)

- [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.MX27 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 platform (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 2.6.22 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. • 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 and ARM9 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 1, “Machine Specific Layer (MSL)”	all
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 1, “PMIC Protocol Driver”	i.MX27

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
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 1, “PMIC Audio Driver”	i.MX27
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 1, “PMIC Digitizer Driver”	i.MX27
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 1, “PMIC Real Time Clock (RTC)”	i.MX27
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 1, “PMIC Power Management Driver”	i.MX27
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. On imx27pdk platform, this driver is not used.	Chapter 1, “PMIC Connectivity Driver”	i.MX27 ADS

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
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 1, “PMIC Battery Driver”	i.MX27
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). On the i.mx27pdk platform, this driver is not used.	Chapter 1, “PMIC Light Driver”	i.MX27 ADS
Power Management Drivers			
CPUFreq	Frequency scaling is an important part of increasing the battery life of portable devices, but it also has a place in reducing power consumption. This driver allows the CPU frequency to dynamically change according to the CPU load or to user demands. Optionally, it is also possible to change the CPU core voltage depending on the selected frequency to reduce power consumption even more.	Chapter 1, “Dynamic Frequency Scaling Driver (CPUFreq)”	I.MX27PDK
Multimedia Drivers			
LCDC	The i.MX liquid crystal display controller (LCDC) provides display data for external gray-scale or color LCD panels. The LCDC is capable of supporting black-and-white, gray-scale, passive-matrix color (passive color or CSTN), and active-matrix color (active color or TFT) LCD panels.	Chapter 23, “Liquid Crystal Display Controller (LCDC) Driver”	i.MX27
TV-OUT	TV-OUT is a television encoder device that encodes video signals and generates synchronization signals for a given television standard.	Chapter 1, “CH7024 TV Encoder (TV-Out) Driver”	i.MX27
eMMA	The enhanced Multimedia Accelerator (eMMA) consists of the video Pre-processor (PrP) and Post-processor (PP) blocks. These blocks provide video acceleration and off-load the CPU from computation-intensive tasks. The PrP and PP can be used for generic video pre- and post-processing, such as scaling, resizing, and color space conversions.	Chapter 26, “The Enhanced Multimedia Accelerator (eMMA) Driver”	i.MX27
V4L2 Output	The driver implements the standard V4L2 API for output devices.	Section 16.2.3.3, “V4L2 Output Device”	all
V4L2 Capture	The Video for Linux 2 (V4L2) capture device includes two interfaces: the capture interface and the overlay interface.	Section 16.2.3.1, “V4L2 Capture Device”	i.MX27
VPU	The Video Processing Unit (VPU) is a multi-standard video decoder and encoder that can perform decoding and encoding of various video formats.	Chapter 17, “Video Processing Unit (VPU) Driver for MX27” ”	i.MX27

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
Camera (OV2640)	The OV2640 Camera driver is designed under Linux V4L2 architecture. It implements the V4L2 capture interface.	Chapter 18, “OmniVision Camera Driver (OV2640)”	i.MX27
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 1, “Advanced Linux Sound Architecture (ALSA) Sound Driver with PMIC Hardware Support”	i.MX27
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 1, “Digital Audio Multiplexer (AUDMUX) Driver”	i.MX27
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 1, “Synchronous Serial Interface (SSI) Driver”	i.MX27
Memory Drivers			
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 1, “NAND Flash Memory Technology Device (MTD) Driver”	i.MX27
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 1, “Low-Level Keypad Driver”	i.MX27

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
Networking Drivers			
FEC	The FEC Driver performs the full set of IEEE 802.3/Ethernet CSMA/CD media access control and channel interface functions. The FEC requires an external interface adaptor and transceiver function to complete the interface to the Ethernet media. It supports half- or full-duplex operation on 10 Mbps or 100 Mbps related Ethernet networks.	Chapter 1, “Fast Ethernet Controller (FEC) Driver”	.MX27
SMSC LAN9217	he 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.	Chapter 1, “SMSC LAN9217 Ethernet Driver	MX27PDK
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 25, “Security Drivers”	i.MX27
SAHARA	The Symmetric / Asymmetric Hashing and Random Accelerator (SAHARA) driver module drives the hardware SAHARA2 present on the i.MX platforms. SAHARA2 accelerates the following security functions: • AES encryption/decryption • DES/3DES • ARC4 (RC4-compatible cipher) • MD5, SHA-1, SHA-224 and SHA-256 hashing algorithms • HMAC (support for IPAD and OPAD through descriptors) • Random number generator	Section Chapter 1, “Symmetric/Asymmetric Hashing and Random Accelerator (SAHARA) Drivers”	i.MX27

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
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 I2C 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 1, “Inter-IC (I2C) Driver”	i.MX27
1-Wire	This is an integrated One-Wire interface. The driver is implemented as a character driver and provides a custom user space API. This driver supports: • a single OWire memory device connected to the OWire peripheral for read/write bit and read/write byte operations. • the OWire peripheral in the product for single device detection and selection. • the interface to the OWire peripheral at the read/write block and read/write page level.	Chapter 32, “One-Wire Driver”	i.MX27 ADS
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 1, “Configurable Serial Peripheral Interface (CSPI) Driver”	i.MX27
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 1, “MMC/SD/SDIO Host Driver”	i.MX27
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 1, “Universal Asynchronous Receiver/Transmitter (UART) Driver”	i.MX27
16552 UART	The Universal Asynchronous Receiver/Transmitter (UART) Driver provides the standard Linux serial driver API for all of the external UART ports.	Chapter 1, “16C652 Universal Asynchronous Receiver/Transmitter (UART) Driver”	i.MX27 ADS

Table 1-1. Supported Features (Continued)

Features	Description	Chapter Source	Applicable Platform
General Drivers			
USB	The USB driver implements a standard Linux driver interfaces to ARC USB-HSOTG controller.	Chapter 1, “ARC USB driver”	i.MX27
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 1, “Real Time Clock (RTC) Driver”	i.MX27
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 1, “Watchdog (WDOG) Driver”	i.MX27
FM (Si4702)	The FM (Si4702) driver provides the interfaces to control Si4702 chips.	Chapter 1, “FM Driver”	i.MX273-Stack
MMA7450L Accelerometer	The MMA7450L is a feature-rich accelerometer device with a flexible programming interface exposed to the software.	Chapter 1, “MMA7450L Accelerometer Driver”	i.MX273-Stack
GPS	The communications driver provides a serial interface to the core driver and communicates with the external GPS chip set.	Chapter 1, “Global Positioning System (GPS) Driver”	i.MX27
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.MX27
GUI			
Qt/E	Qt/E is a Graphical User Interface supported by the Linux BSP.		i.MX27
Tools			

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 PDK 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 PDK board

— Using RedBoot to boot the Linux OS on a PDK 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 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. The optional accessory kit applies only to the PDK kits.

Table 2-1. Hardware Resources Needed

Resource	Description
i.MX27 PDK (3-stack)	i.MX27 PDK containing a board, serial cable, and other accessories

The kit may include an external memory card. It is important that this memory card be properly installed into the CPU card. This is done by tilting the memory card at a slight angle above horizontal and pushing the contacts into the connector until it clicks. Then press the card down into a flat, horizontal position until the two spring clips engage the side of the card. To remove the memory card later, pull the spring clips away from the sides of the memory card until it pops up. Then slide it out of the connector.

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.

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_200840.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
- ARM RealView tools setup for the PDK 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 RedBoot prompt does not show up the first time after turning power on or pressing the `Reset` button, press "`Ctrl` + `c`" multiple times.

NOTES

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 the `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 the 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 the 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 printing out 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 may need to be modified with proper path.
3. Resume RedBoot from RVD by typing the command "`go`" on 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.MX27 PDK Linux User's Guide*.

2.1.5 Booting Linux

To boot Linux, issue the following RedBoot commands:

```
fis load kernel
exec
```

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 such 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/mtdblock4 rw"
```

A login prompt is displayed on the terminal console:

```
localhost login:
```

Use "`root`" as user name to login (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 *Qtopia* desktop or the calibration screen, if calibration has never been performed.

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 box.

   These steps set up the default receive directory. If these steps are skipped, then files are stored in your 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 PDK board has 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 and ftpput from your shell prompt 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.

In addition to the external Ethernet, the i.MX27 PDK, also have the internal FEC module. See Chapter 25, “Fast Ethernet Controller (FEC) Driver.”

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 have been installed previously on your host. The exact package names may vary, depending on your Linux distribution.

- 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 or
http://doc.trolltech.com/qtopia4.3/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-2.6.22.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-2.6.22.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-2.6.22.tar.bz2
```

You should see the following output:

```
2e230d005c002fb3d38a3ca07c0200d0
```

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.

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.

To build using the default configuration:

`./ltib --preconfig config/platform/imx27pdk/defconfig` for PDK

When LTIB builds, it packages the results of building the kernel and each root filesystem package as an `rpm` file. 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. However, 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`

Note: 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
```

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. Save your changes and exit.
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. Save your changes and exit.

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 2.6.22-imx)**. Exit saving changes. Then type:

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

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

2.3.2.3.4 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 on the root filesystem in the `/lib/modules/2.6.22-pdk27_rel1` 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 Freescale for the i.MX family of processors.

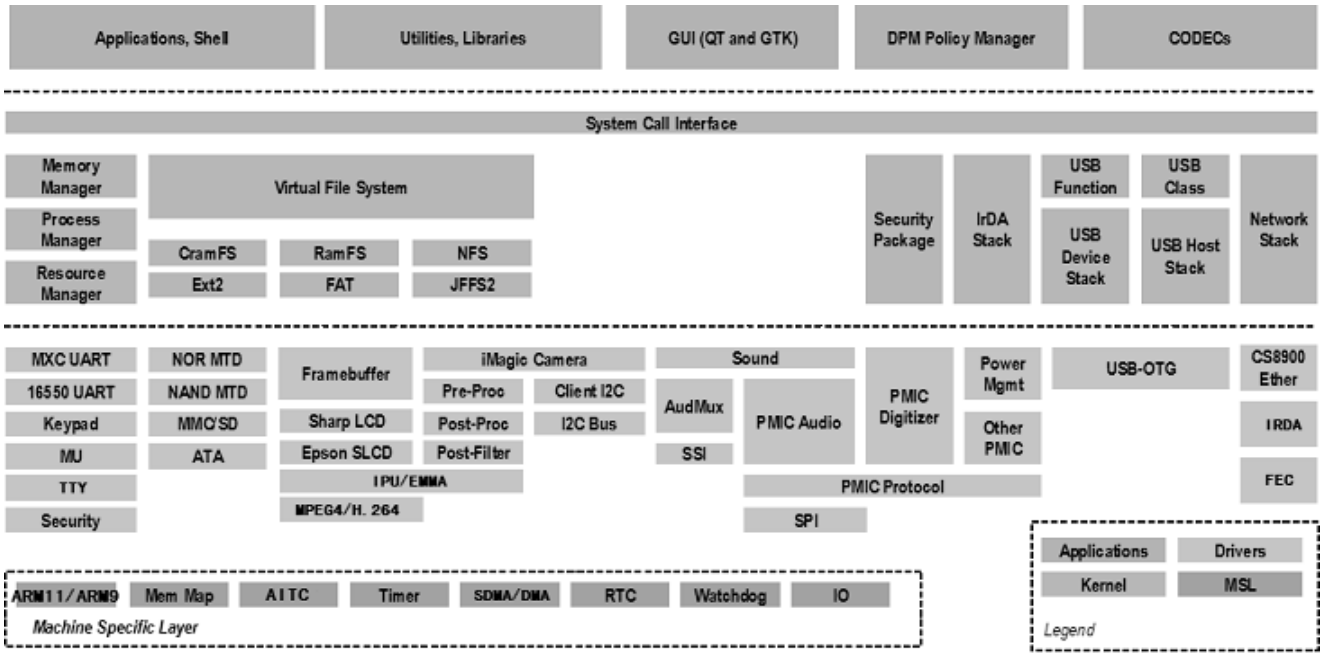


Figure 3-1. Linux BSP Block Diagram

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 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 the some of the configuration settings apart from the standard features:

The serial port is labeled as UART-DCE on the i.MX27 PDK.

- Embedded mode
- Module loading/unloading
- ARM9 and 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.MX27 3-Stack	<ltib_dir>/rpm/BUILD/linux-/arch/arm/mach-mx27

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) or the ARM9™ Interrupt Controller (AITC), depending on the platform. 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 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.

3.2.2.5 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 contains 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 16C652 UART Driver and the UART Driver.

3.3.1.1 16C652 Universal Asynchronous Receiver/Transmitter (UART) Driver

The 16C652 Universal Asynchronous Receiver/Transmitter (UART) driver provides the interface to the external SC16C652 UART on all the i.MX PDK boards. It provides the standard Linux serial driver API for both of the external UART ports. It supports the following features:

- Interrupt-driven transmit/receive of characters
- Standard Linux baud rates from 460.8K baud down to 50 baud
- Two UART ports on 16C652

- Each 16C652 UART port can be programmed for 1, 4, 8, or 14 byte threshold levels to interrupt
- Supports transmitting and receiving characters with 7-bit and 8-bit character lengths
- Supports transmitting 1, 1.5, or 2 stop bits
- Supports odd and even parity
- Supports XON/XOFF software flow control
- Supports CTS/RTS hardware flow control
- 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
- Supports the standard TTY layer ioctl calls
- Includes console support that is needed to bring up the command prompt through one of the UART ports

Currently, the 16C652 UART driver is used by default to bring up the console. Power management, autobaud detection, and DMA are not supported by this driver.

3.3.1.2 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 DMA-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 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.

3.3.1.3 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.

3.3.1.4 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 time-out 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.

3.3.2 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 and ARM9 processors. 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 codec setup

and control, DMA setup and control, and control of audio accessories, such as headphones and microphone detection. Stream mixing may also be supported, depending on the codec.

3.3.3 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.

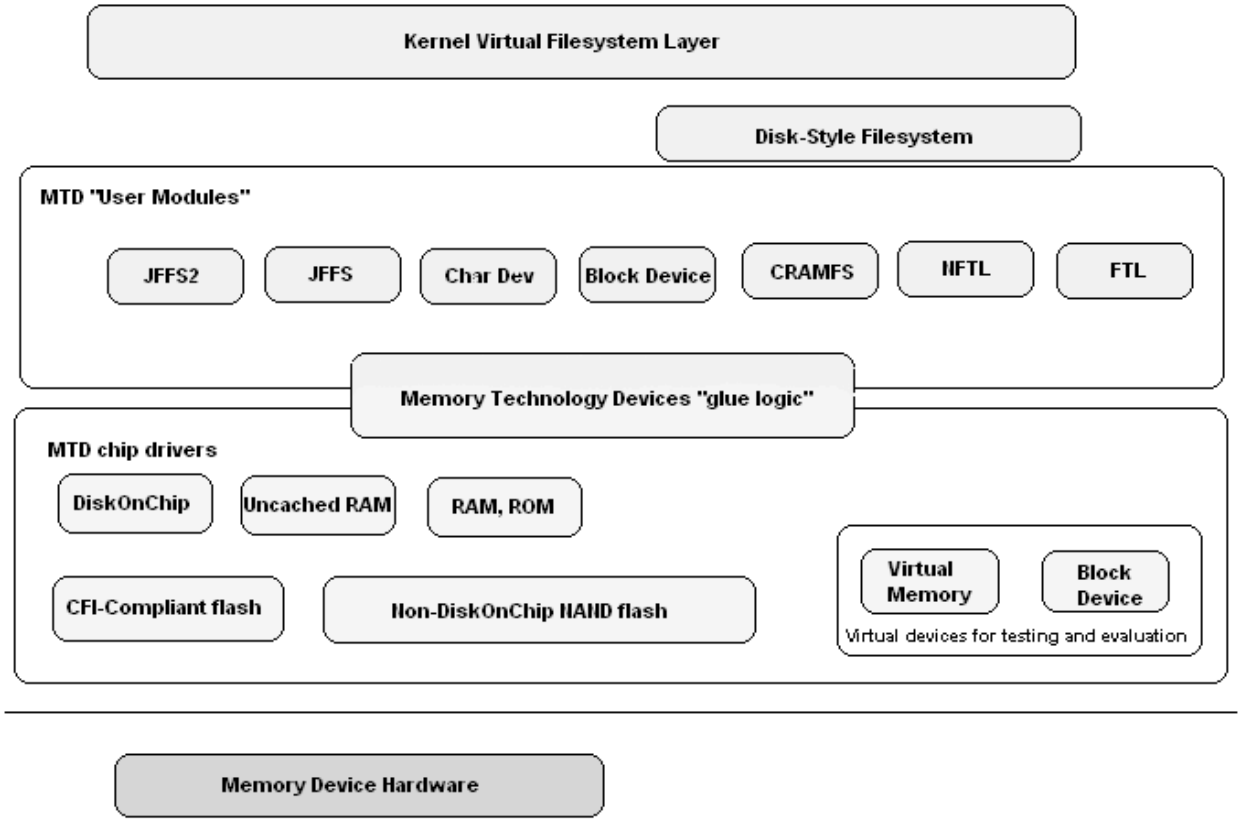


Figure 3-2. MTD Architecture

Figure 3-2 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.

3.3.3.1 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.

3.3.4 Networking Drivers

The networking drivers are described in the next sections.

3.3.4.1 CS8900A Ethernet Driver

NOTE

This driver is not supported in the i.MX27 PDK (3-Stack), i.MX31 PDK (3-Stack) board.

The CS8900 Ethernet adapter from Cirrus Logic follows IEEE 802.3 standards and supports half- or full-duplex operation at 10 Mbps. The CS8900A driver interfaces the CS8900A-specific functions with the standard Linux kernel network module. It supports the following features:

- The efficient PacketPage Architecture operates in I/O and memory space and as a DMA slave
- Full duplex operation
- On-chip RAM buffers for transmit and receive of frames
- Supports programmable transmit features like automatic retransmission on collision and automatic CRC generation
  - Supports programmable receive features
  - Support for DMA transfer
  - Early interrupts for frame preprocessing
  - Automatic rejection of erroneous frames
- EEPROM support for configuration
- Supports setting of multicast-list
- 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 initialize it during boot.

3.3.4.2 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.

3.3.4.3 Fast IrDA (FIRI) Driver

NOTE

This driver is not supported in the i.MX27 PDK (3-Stack), i.MX31 PDK (3-Stack) board and i.MX35 3-stack board.

The Fast Infra-Red Interface (FIRI) module provides support for infrared communication. The FIRI module is capable of establishing a 0.576 Mbit/sec, 1.152 Mbit/sec, or 4 Mbit sec half duplex link through an LED and IR detector.

The FIRI driver is used by the IrDA subsystem of the Linux kernel. The FIRI driver implements the standard Linux driver APIs for data transfer, `ioctls` and power management required by the upper layer.

The FIRI driver includes the following features:

- Compliant with IrDA 1.1 for MIR and FIR
- Supports 0.56 Mbps and 1.152 Mbps for MIR
- Supports 4 Mbps for FIR
- 16/32-bit CRC generation and error detection
- Interrupt-driven transmit/receive of data
- DMA capability
- Full physical layer implementation
- Device Destination Detection Hardware Support
- SIP generation for collision avoidance
- Hardware packet assembly
- Hardware packet search

- Software packet assembly
- Software packet search
- Power management

The Serial InfraRed (SIR) protocol, which supports data rate 115.2 kbps or lower, is implemented in the UART module (see Section 3.3.1.2, “Universal Asynchronous Receiver/Transmitter (UART) Driver”).

3.3.5 Disk Drivers

The disk drivers include the ATA driver.

3.3.5.1 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

3.3.6 Security Drivers

3.3.6.1 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

3.3.6.2 Hash Accelerator Controller (HACC) Driver

The Hash Accelerator Controller (HACC) is a hardware accelerator designed to assist in the hashing of the external Flash or RAM. It runs the SHA-1 algorithm on the given input to produce a 160 bit hash.

The HACC includes the following features:

- Accelerates the generation of a SHA-1 hash over selected memory contents.
- Works on 512 bits at a time, and in the end will pad a final block with a defined sequence so that the final block is also a 512 bit entity.
- Has the flexibility to hash using 16-word bursts, or with incremental bursts for memories not capable of handling 16-word bursts.
- The 160 bit resultant hash can be read out of the five 32-bit HSH registers.
- Calculates the SHA-1 hash over a number of large, potentially non-contiguous segments of memory.
- Power management

3.3.6.3 Real-Time Integrity Checker (RTIC) Driver

The Run-Time Integrity Checker (RTIC) is part of the PISA family of platform security components. Its purpose is to ensure the integrity of the peripheral memory contents and assist with boot authentication. The RTIC has the ability to verify the memory contents during system boot and during run-time execution. If the memory contents at runtime fail to match the hash signature, an error in the security monitor is triggered.

The RTIC includes the following features:

- SHA-1 message authentication
- Segmented data gathering to support non-contiguous data blocks in memory (up to 2 segments per block)
- Works with High Assurance Boot process

- Support for up to 4 independent memory blocks
- Programmable DMA bus duty cycle timer and watchdog timer

3.3.7 General Drivers

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

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

3.3.7.1 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

3.3.7.2 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.

3.3.7.3 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.

3.3.7.4 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 DMA API.

3.3.7.5 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.

3.3.7.6 Dynamic Power Management (DPM) Driver

NOTE

The i.MX27 PDK board doesn't use DPM to manage dynamic frequency scaling. It uses CPUFreq instead. So paragraphs 3.3.7.6, "Dynamic Power Management (DPM) Driver", 3.3.7.7, "Policy Architecture", 3.3.7.8, "Operating Points", 3.3.7.9, "Operating States", 3.3.7.10, "Policy Managers", and 3.3.7.11, "Low-Level Power Management Driver" do not refer to i.MX27 PDK.

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

3.3.7.7 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.

3.3.7.8 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.

3.3.7.9 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.

3.3.7.10 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 3-3 shows the high-level design for DPM.

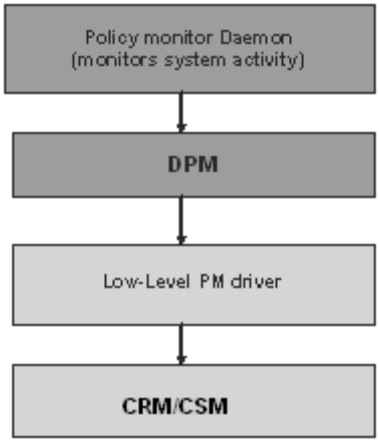


Figure 3-3. DPM High Level Design

Figure 3-4 specifies the DPM architecture block diagram.

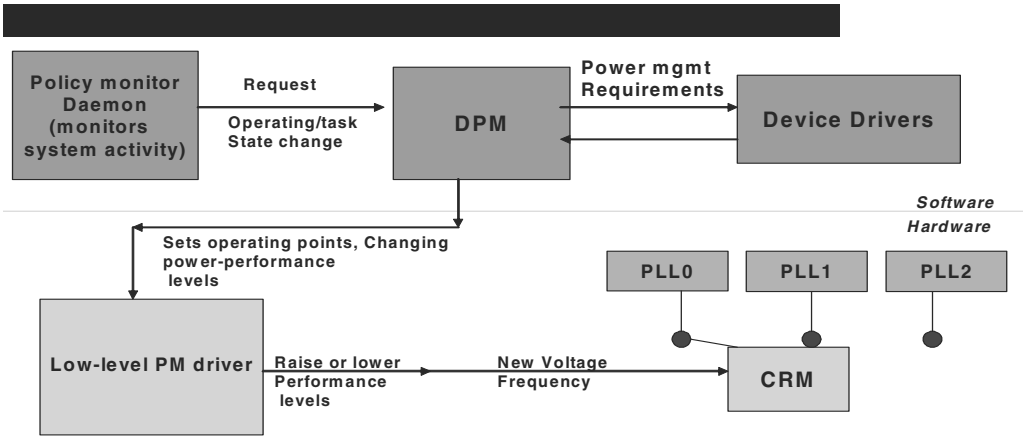


Figure 3-4. DPM Architecture Block Diagram

3.3.7.11 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.

3.3.7.12 CPUFreq

Frequency scaling is an important part of increasing the battery life of portable devices, but it also has a place in reducing power consumption.

This driver allows the CPU frequency to dynamically change according to the CPU load or to user demands. Optionally, it is also possible to change the CPU core voltage depending on the selected frequency to reduce power consumption even more.

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>/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 the kernel image during the kernel build.

The following boot loader is provided in the BSP:

- RedBoot

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

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:

- Qtopia v4.3 for PDK

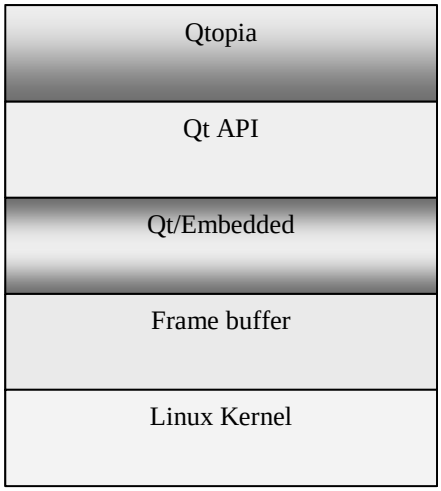


Figure 3-5. Qt/Embedded

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”.

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.

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 can be obtained from http://www.trolltech.com.

3.8 Source of Linux BSP Components

Figure 3-6 shows the source of the code for each of the Linux BSP components.

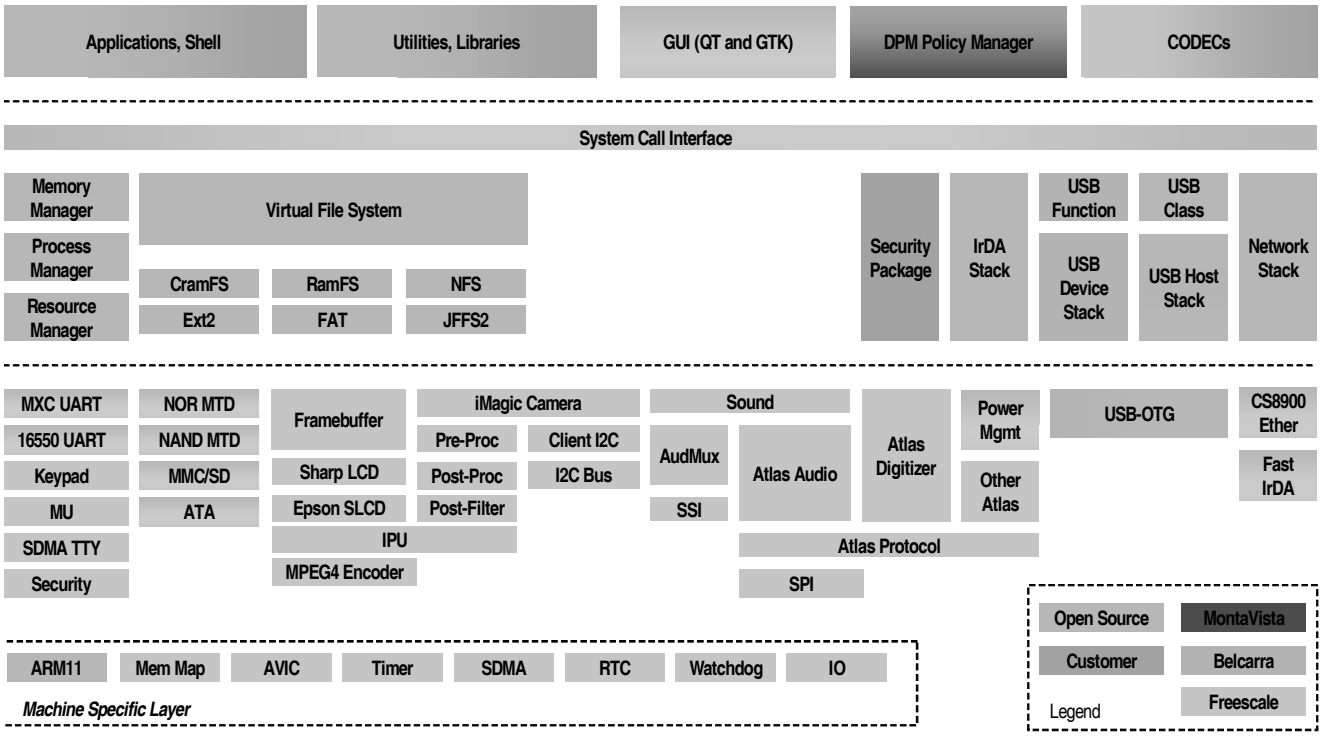


Figure 3-6. Linux BSP Source Diagram

3.9 Linux BSP APIs

Figure 3-7 shows a high level view of the Linux BSP components.

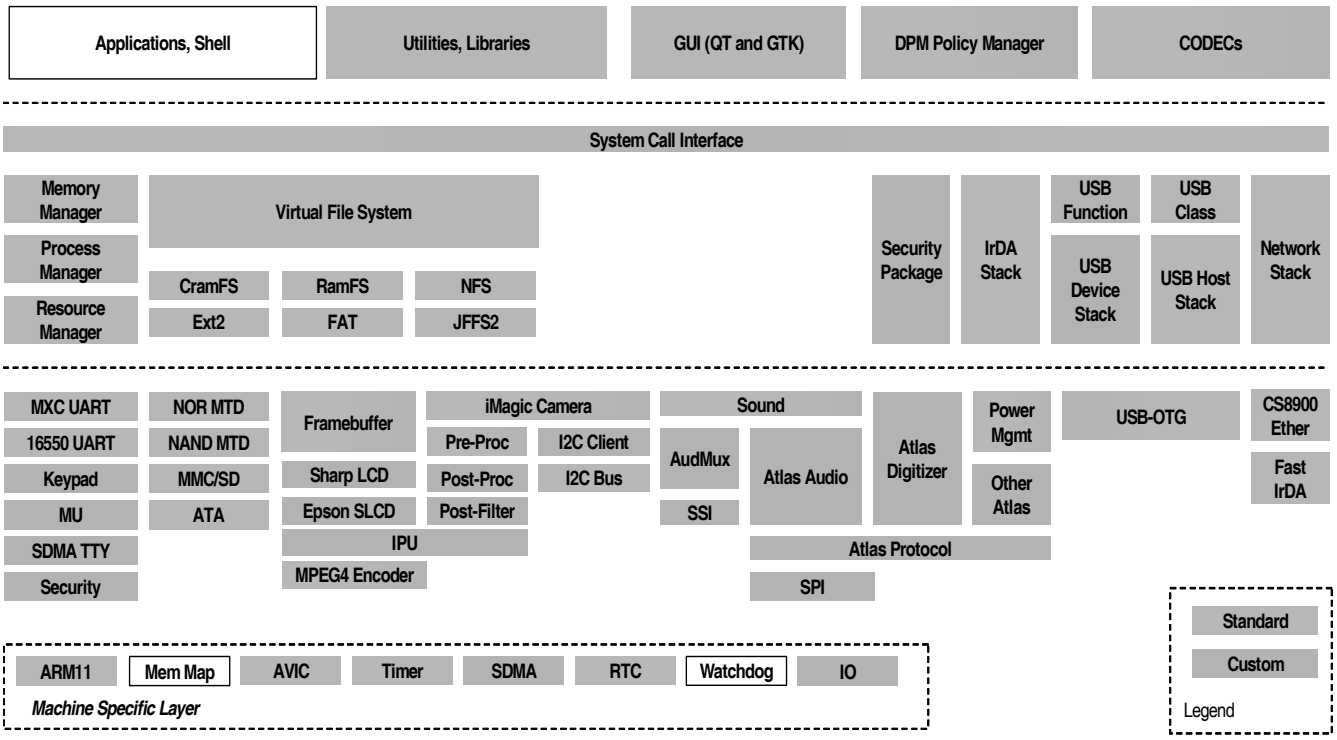


Figure 3-7. Linux BSP API Diagram

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
DMAC(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
SMSC 9217 Ethernet		X	Linux Std

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

SW Component	Kernel API	User API	Comment
Fast IrDA			Linux Std
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
Memory Stick			TBD
I2C Bus	X		Linux Std
I2C Client	X		FSL Custom
DMA 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
CPUFreq	X	X	Linux Std
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)

These modules are normally available in the following directory:

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

The header files are implemented under the following directory:

```
<ltib_dir>/rpm/BUILD/linux-/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 GPIO (including IOMUX on some platforms) are detailed.

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.

Only in supervisor mode can Interrupt Controller registers be accessed. The Interrupt Controller's interrupt requests are prioritized in the order of fast interrupts (in order of highest order) 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 exceptions. 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 some detailed description about the ARM interrupt architecture.

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

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

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-/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-/arch/arm/plat-mxc/irq.c
```

There are also two header files:

```
<ltib_dir>/rpm/BUILD/linux-/include/asm-arm/arch-mxc/hardware.h
<ltib_dir>/rpm/BUILD/linux-/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 a 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-/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-/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
`mx27.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 memory map between physical and virtual addresses. It just defines an initialization function to be called during system startup.

4.4 IOMUX

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 at the same time to meet requirements from various customers.

Note that not all platforms have the IOMUX hardware module. On those platforms, the pin muxing is done through the GPIO module.

Pins coming out of highly integrated processors can have multiple purposes due to its limited size. For a multi-purpose pin, the IOMUX module 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 some other alternate functions) 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. For example, on certain platform, the TXD1 pin can be multiplexed with:

- TXD1— internal UART1 Transmit Data pin. This is the primary function of this pin.
- TCK—hardware mode 1
- USB_OTG_DATA1—alternate mode 1
- PP4_TXCLK/SCK—alternate mode 2
- RI_DCE1—alternate mode 4
- MCU2_5—GPIO

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose which cannot be changed by software. Otherwise, the IOMUX 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 and therefore to be used as the UART data transmit signal, it should be configured as the primary function. If this 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 module is divided into MUX control and pad control sub modules. The following sections briefly describe the hardware operation. For detailed information, refer to the relevant IC specifications.

4.4.1.1 SW_MUX_CTL

The SW_MUX_CTL module contains SW_MUX_CTL registers to control each IOMUX, on a pad-to-pad basis. The SW_MUX_CTL registers are partitioned into 4×8 bit fields. Each field is mapped to a specific pad. Each field is partitioned as follows: 4 bits to control the input path, 3 bits to control the output path, and 1 reserved bit.

4.4.1.2 SW_PAD_CTL

The IOMUX controller has a pad control register (SW_PAD_CTL) made of a set of registers to control each PAD. The SW_PAD_CTL registers are used to control several parameters of the pad's configuration:

- Active devices, such as pull-up/down, keeper, and open drain.
- Pad functionality, such as software input on, max drive, hysteresis, and power down.

4.4.2 IOMUX Software Operation

The iomux software implementation provides an API to set up 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 iomux module is implemented in iomux.c file under the relevant MSL directory. The header file to define the pin names is under:

<ltib_dir>/rpm/BUILD/linux-/arch/arm/mach-xxx/xxx_pins.h (where xxx 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
xxx_pins.h	pin name definitions

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 GPIO Module

The following discussion applies to those platforms that control the muxing of a pin through a 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 cannot 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 a 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. 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 set up 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-/arch/arm/mach-xxx/xxx_pins.h` (where `xxx` stands for a specific CPU)

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

Table 4-4. IOMUX File List

File	Description
gpio_mux.c	iomux function implementation
xxx_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 dedicated general-purpose pins that can be configured as either inputs or outputs. When configured as an output, a pin's state (high/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 utility functions should be called to set up the desired functionality for a pin instead of directly access to the GPIO registers. The GPIO interrupt implementation contains functions, such as interrupt service routine (ISR) registration/un-registration and ISR dispatching once an interrupt occurs. All driver-specific GPIO setup functions are recommended to be made during device initialization time in the MSL layer to provide better portability and maintainability. This GPIO interrupt is initialized automatically during the system starts.

If a pin is configured as GPIO by the IOMUX, more function calls should be made to set up the state of a pin. The pad sub-module control within IOMUX may be required to be set up as well.

4.5.1.1 API Interface for GPIO

The GPIO implementation has the following features:

- Standardized 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 for all the possible GPIO pins that are able of generating interrupts.
- Capability to request and free the usage of a IOMUX pin. If that pin is used as GPIO, another set of request/free function calls are provided. User should check the return value of the "request" calls before modifying the pin's current state. The "free" function calls should be made when the pin is not needed. See 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 where that pin is located in the GPIO module.
- Minimal changes required for the public drivers such as Ethernet driver and 8250 UART driver as no special GPIO function call is needed for registering interrupt.

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

4.5.1.2 IOMUX/GPIO API Usage Example For MX27

To configure the `UART1_RXD` pin for functional mode used by the UART driver, `gpio_uart_active()` calls:

```
...
gpio_request_mux(MX27_PIN_UART1_RXD, GPIO_MUX_PRIMARY);
```

To set it up, 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:

```
...
gpio_free_iomux(MX27_PIN_UART1_RXD);
```

4.5.2 GPIO Requirements

This GPIO implementation should meet the following requirements where the GOPIO module:

- Implements the functions for accessing the GPIO hardware modules.
- Provides a centralized place to control GPIO signal setup and GPIO interrupts for the whole system.

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-/arch/arm/plat-mxc/gpio.c`

Includes are available in the following files:

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

Table 4-5. GPIO File List

File	Description
`xxx_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

Note that not all platforms have the EDIO hardware module. So this section may only apply to the platforms that have a EDIO module.

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 configuration for each incoming signal reaching EDIO.

4.6.2 EDIO Software Operation

EDIO interrupt has been integrated into the generic platform level interrupt implementation as in `irq.c` under `<ltib_dir>/rpm/BUILD/linux-/arch/arm/plat-mxc` directory. For the 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-/arch/arm/plat-mxc/` directory and the `hardware.h` is under the `<ltib_dir>/rpm/BUILD/linux-/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 the 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-/arch/arm/plat-mxc/spba.c
<ltib_dir>/rpm/BUILD/linux-/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
Direct Memory Access Controller (DMAC) API

5.1 Overview

The direct memory access controller (DMAC) provides 16 channels supporting linear memory, 2D memory, and FIFO transfers to provide support for a wide variety of DMA operations.

5.1.1 Hardware Operation

- Sixteen channels support linear memory, 2D Memory, and FIFO for both source and destination.
- DMA chaining for variable length buffer exchanges and high allowable interrupt latency requirement.
- Increment, decrement, and no-change support for source and destination addresses.
- Each channel is configurable to response to any of the DMA request signals.
- Supports 8, 16, or 32-bit FIFO and memory port size data transfers.
- DMA burst length configurable up to a maximum of 16 words, 32 half-words, or 64 bytes for each channel.
- Bus utilization control for the channel that is not triggered by a DMA request.
- Burst time-out errors terminate the DMA cycle when the burst cannot be completed within a programmed time count.
- Buffer overflow error terminates the DMA cycle when the internal buffer receives more than 64 bytes of data.
- Transfer error terminates the DMA cycle when a transfer error is detected during a DMA burst.
- DMA request time-out errors are generated for channels that are triggered by DMA requests to interrupt the CPU when a DMA burst does not start on that channel after a programmed time count.
- Interrupts provided to the interrupt controller (and subsequently to the core) on bulk data transfer complete or transfer error.
- Each peripheral supporting DMA transfer generates a DMA_REQ signal to the DMA controller, assuming that each FIFO has a unique system address and generates a dedicated `dma_req` signal to the DMA controller. For example, a USB device with 8 end-points has 8 DMA request signals to the DMA if they all support DMA transfer.
- The DMA controller provides an acknowledge signal to the peripheral after a DMA burst is complete. This signal is sometimes used by the peripheral to clear status bits.
- Repeat data transfer function supports automatic USB host-USB device bulk/iso data stream transfer.
- Dedicated external DMA request signal.

5.1.2 Software Operation

The module provides an API for other drivers to control DMA channels. The DMA software operations mainly contain the following:

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

5.2 Requirements

- The module shall implement functions parallel to standard DMA API.
- The module shall conform to the Linux coding standard as documented in the appendix.

5.3 Source Code Structure

The header file, `dma.h`, is available in the directory
`/<ltib_dir>/rmp/BUILD/<linux_version>/include/asm-arm/arch.`

Table 5-1 lists the source files available in the directory
`/<ltib_dir>/rmp/BUILD/<linux_version>/arch/arm/.`

Table 5-1. DMA API Files

File	Description
`mach-mx27/dma.c`	Parameters of DMA channels
`plat-mxc/dma_mx2.c`	DMA API functions

5.4 Programming Interface

The module implements standard DMA API. Standard API is supported partially. It can be used along with custom API functions only. Refer to the API document titled full_cod_doc.pdf for more information on the methods implemented in the driver.

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.

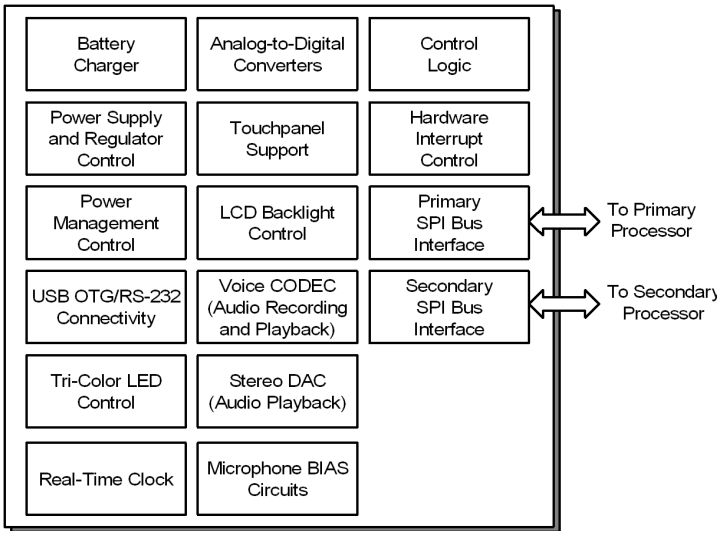


Figure 6-1. PMIC Block Diagram

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.
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).

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

PMIC Device Driver	Functions
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
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

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

Event	Description
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.

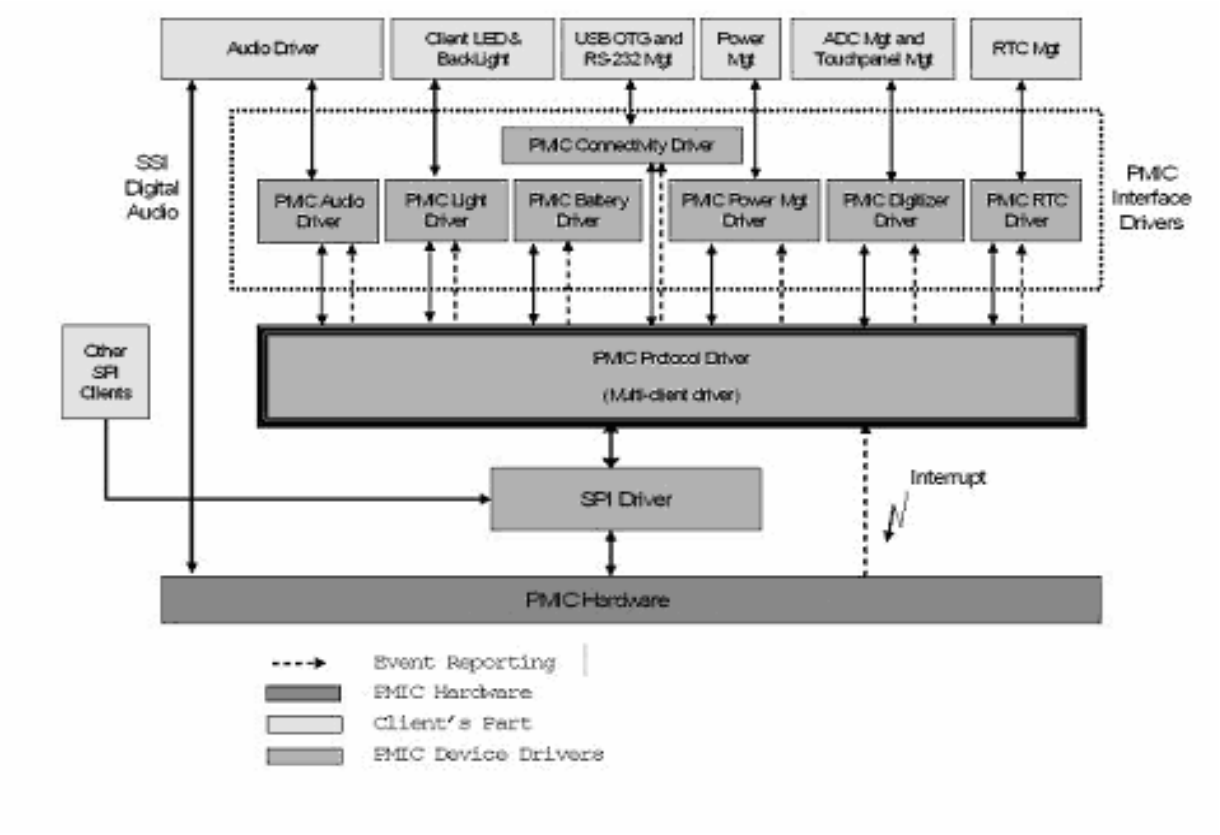


Figure 6-2. PMIC Device Driver

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.

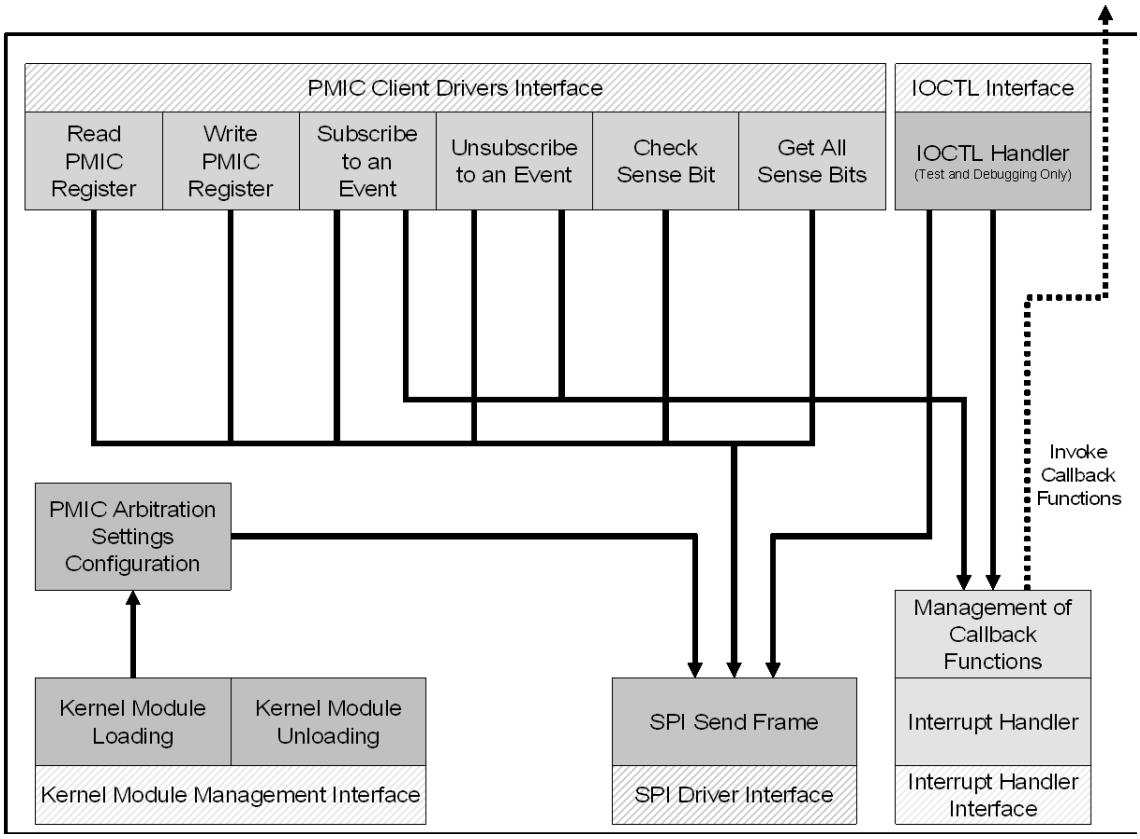


Figure 6-3. PMIC Protocol Driver Architecture and Interfaces

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:

1. 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.
2. 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. Acknowledge and clear the hardware interrupt condition.
2. Schedule 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-/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_config.h	Define global configuration definitions or macros used by client drivers.
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 -> PMIC Protocol Support (SPI Interface) - Choose this to have PMIC protocol driver support. By default, this option is Y for all architectures.

```
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
- Device Drivers -> MXC Support Drivers -> MXC PMIC Support -> MXC_PMIC_FIXARB - This option includes the software workaround for reconfiguring the PMIC's arbitration bit settings to enable secondary processor access. See also Section 6.1.1, "PMIC Register Access and Arbitration."

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.

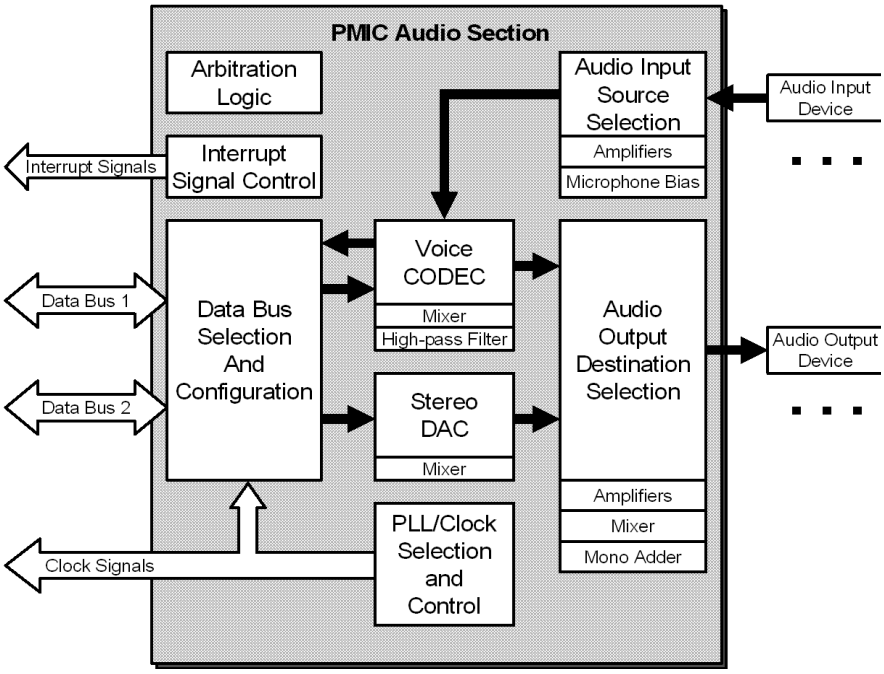


Figure 7-1. PMIC Audio Hardware Components

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 these components are actually available and, if so, 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.

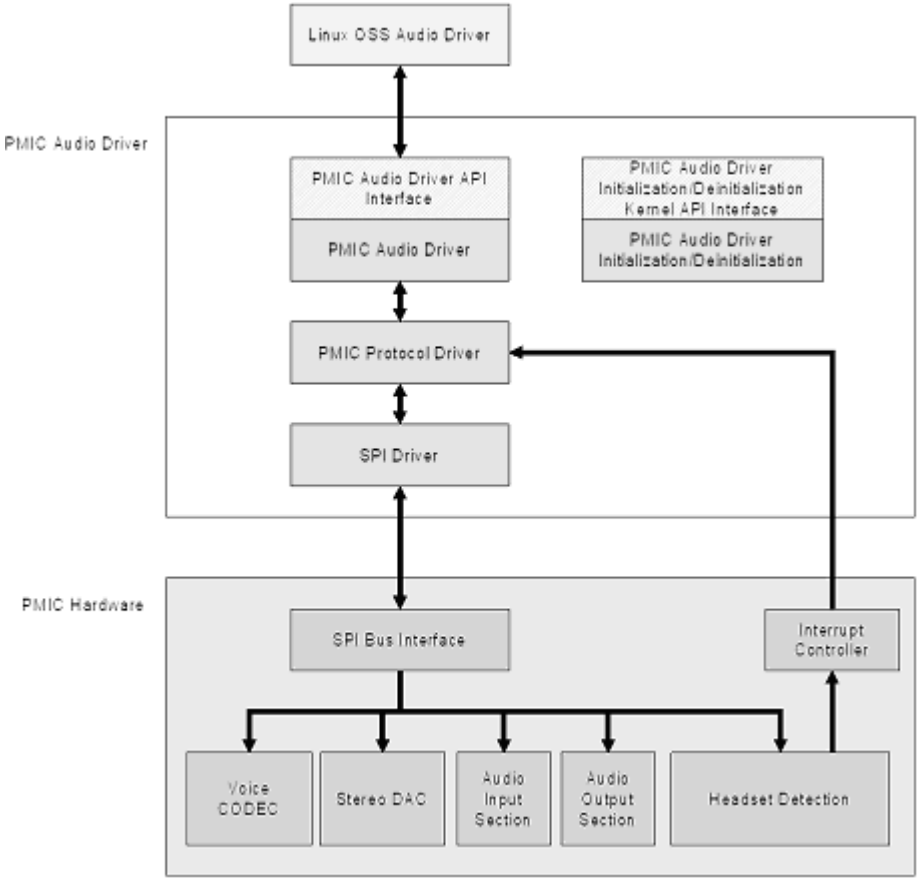


Figure 7-2. PMIC Audio Driver Architecture

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-/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-/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 audio configuration use the command `./ltib -c` when located in the `<ltib dir>`. On the screen displayed, select **Configure the kernel** and exit. When the next screen appears select the following option to enable the audio 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.

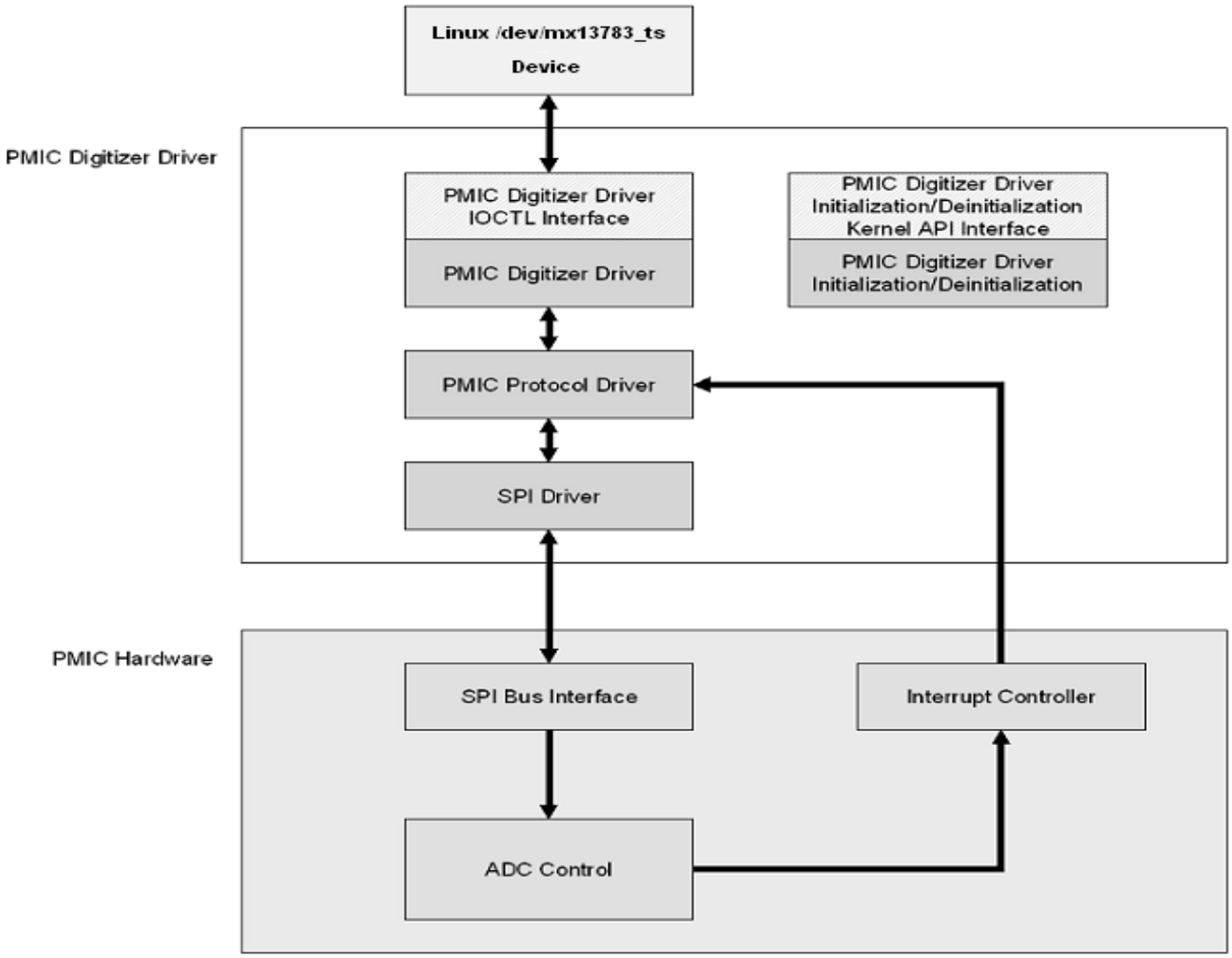


Figure 8-1. PMIC Digitizer Driver Architecture

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

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-/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-/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 6, “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.

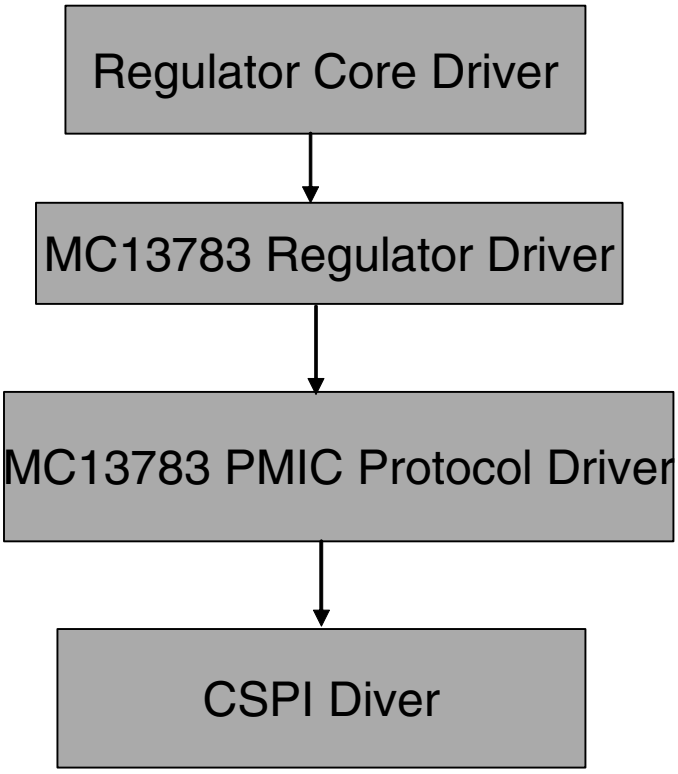


Figure 9-1. MC13783 Regulator Driver Architecture

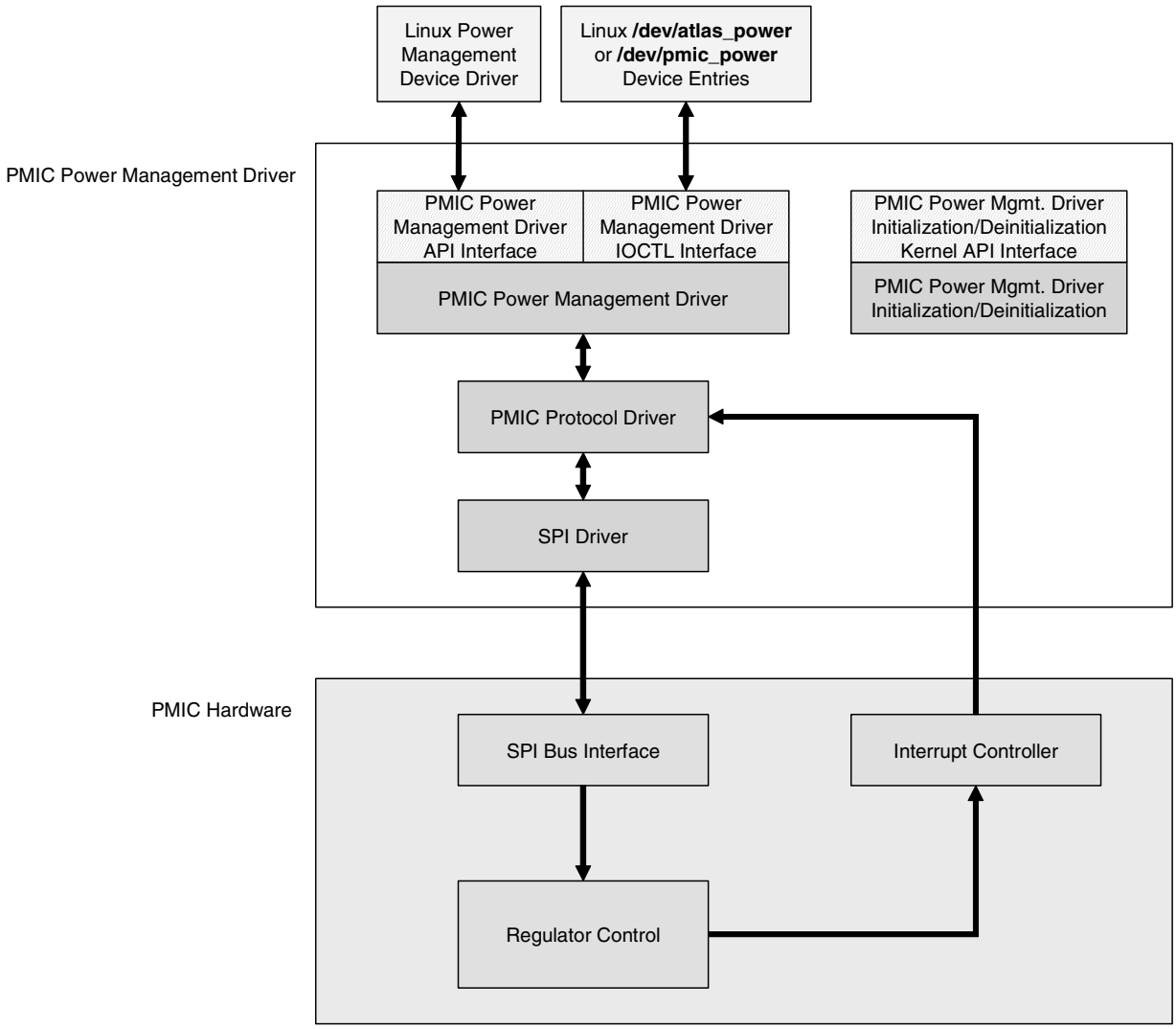


Figure 9-2. PMIC Power Management Driver Architecture

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-/drivers/mxc/pmic/mc13783` and `<ltib_dir>/rpm/BUILD/linux-/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-/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 pmic power 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.

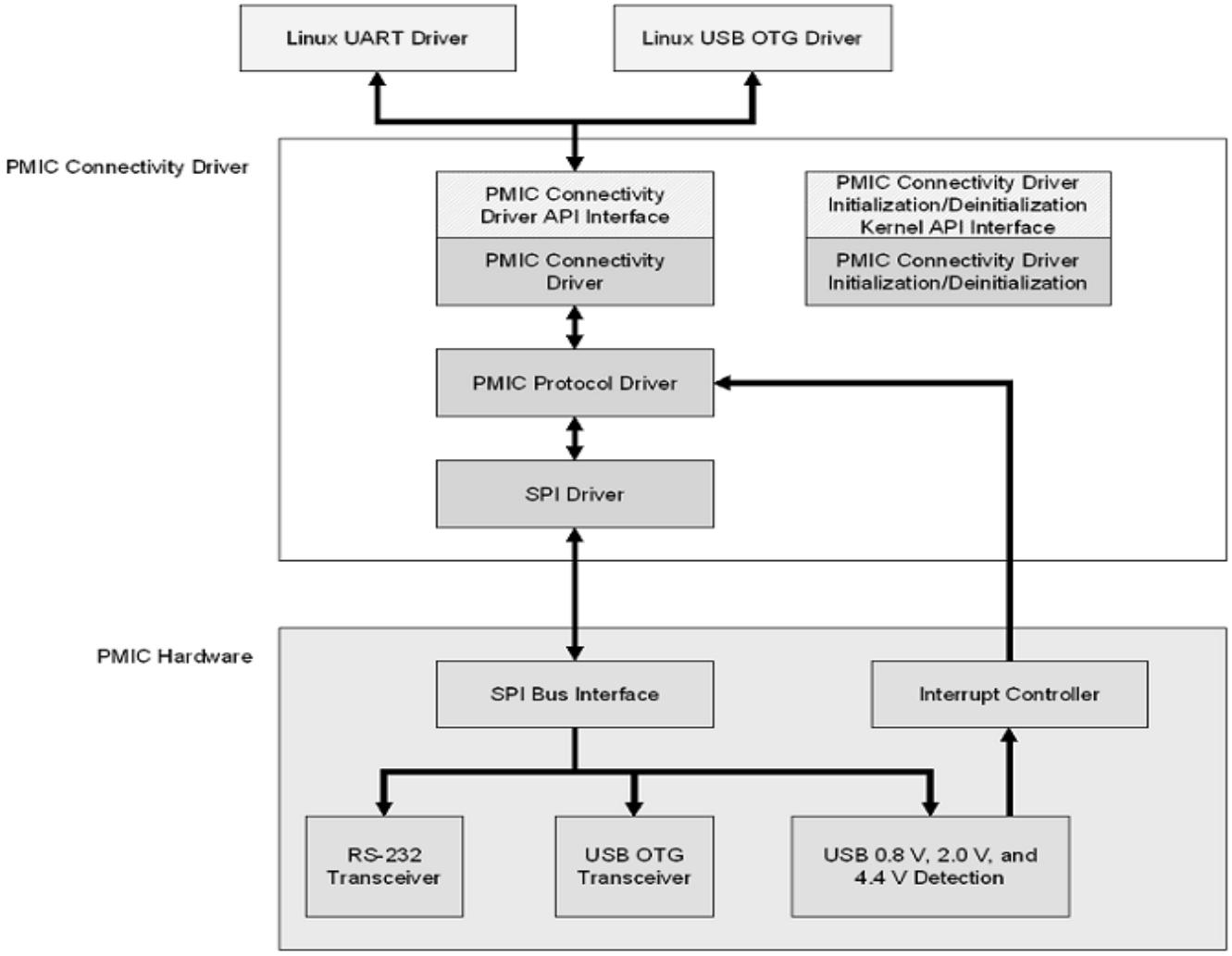


Figure 10-1. PMIC Connectivity Driver Architecture

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 tasklet 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-2.6.22/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-2.6.22/include/asm-arm/arch-mxc/pmic_convity.h`.

10.7 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module. In order to get to the pmic connectivity 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.

- Device Drivers -> MXC Support Drivers -> MXC PMIC Support -> MC13783 Client Drivers -> MC13783 Power API support

  Chooses the MC13783–specific power 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.

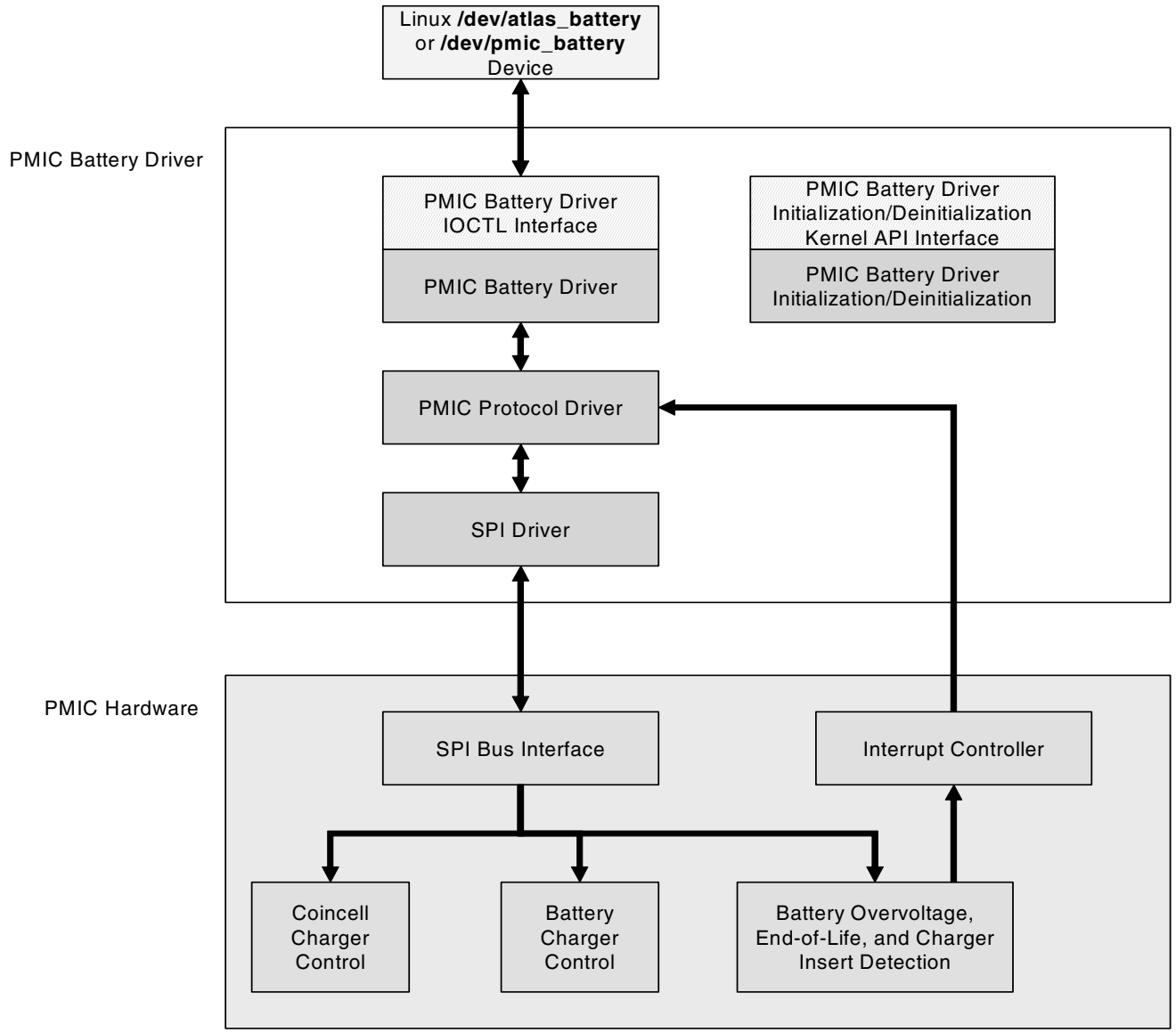


Figure 11-1. PMIC Battery Device Driver Architecture

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>/rmp/BUILD/<linux_version>/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>/rmp/BUILD/<linux_version>/include/asm-arm/arch-mxc/pmic_battery.h`.

11.7 Linux Menu Configuration Options

The following Linux kernel configurations are provided for the PMIC battery device driver. In order to get to the pmic battery 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.

- Device Drivers -> MXC Support Drivers -> MXC PMIC Support -> MC13783 Client Drivers -> MC13783 Battery API support

  Chooses 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 can be located in the documentation for each PMIC.

12.2 Driver Requirements

The PMIC light driver must provide 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 must support 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 must support 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 can be 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.

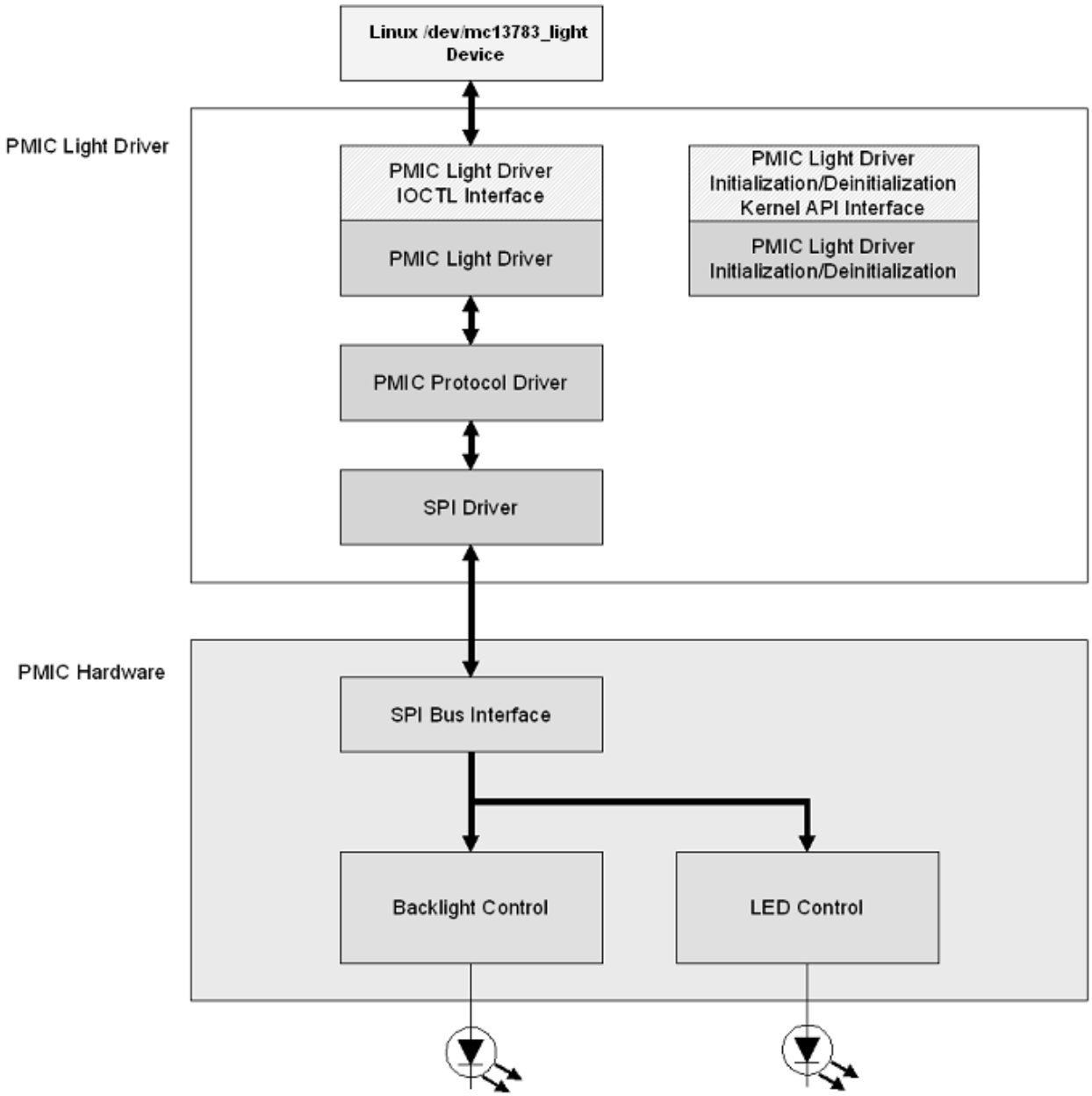


Figure 12-1. PMIC Light Driver Architecture

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>/rmp/BUILD/<linux_version>/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
`/<ltib_dir>/rmp/BUILD/<linux_version>/include/asm-arm/arch-mxc/pmic_light.h`.

12.7 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module. In order to get to the PMIC light 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.

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

Chooses 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 topics:

- 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.

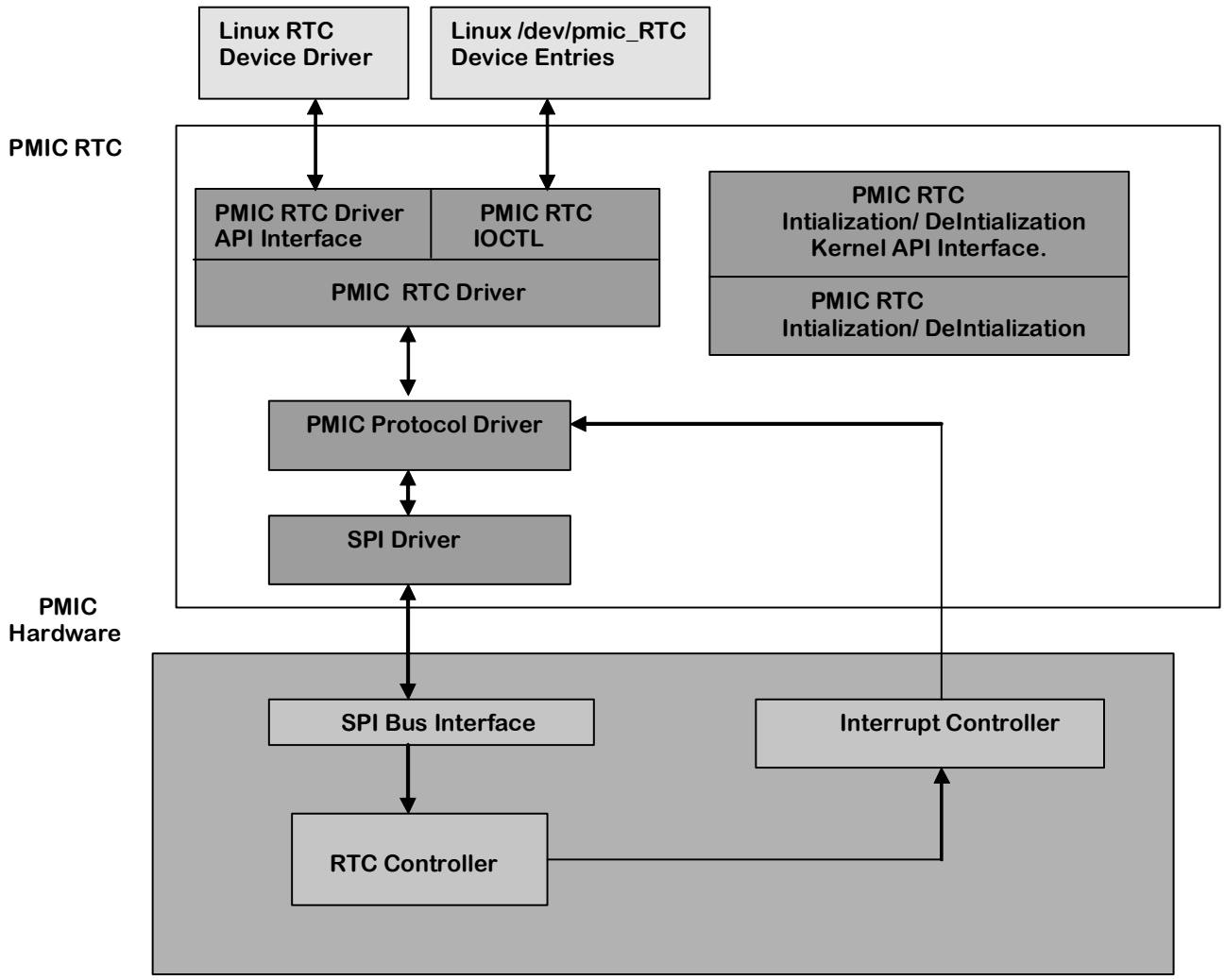


Figure 13-1. PMIC RTC Driver Architecture

13.5 Driver Implementation Details

Configuring the PMIC RTC driver includes parameters as follows:

- 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 you unload the device driver, 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>/rmp/BUILD/<linux_version>/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
`linux/include/<ltib_dir>/rmp/BUILD/<linux_version>/include/asm-arm/arch-mxc/pmic_rtc.h`.

13.7 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module. In order to get to the PMIC RTC 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.

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.MX27 Low-Level Power Management Driver

The purpose of this document is to describe the design of the low-level PM driver for the i.MX27 platform. This driver implements low-power modes. Low-power modes, such as DOZE and SLEEP are implemented to save power.

14.1 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 mode

14.2 Software Operation

For DOZE and SLEEP low-power modes, software should disable interrupts before executing a wait-for-interrupt (WFI) instruction and re-enable interrupts afterwards.

14.3 Hardware Issue

Dynamic Frequency Scaling (DFS) operation is not supported by this driver because of hardware limitation. In the current hardware, it is not possible to change the ARM frequency without a PLL relock. But a PLL relock will lead to DDR instability, because DDR requires a constant clock. As a result, the kernel may crash. This issue will be fixed in the next revision of the i.MX27 chip.

14.4 Source Code Structure

Table 14-1 lists the source files for i.MX27 available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.22/arch/arm/mach-mx27/`.

Table 14-1. Source Code

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 API definitions), associated with the low-level PM driver is available in `<ltib_dir>/rpm/BUILD/linux-2.6.22/include/asm-arm/arch-mxc/`.

14.5 Programming Interface

The API `mxc_pm_lowpower` is currently provided for low-power modes. This implements all the steps required to put the system under DOZE or SLEEP mode.

Chapter 15
Dynamic Frequency Scaling Driver (CPUFreq)

Frequency scaling is an important part of increasing the battery life of portable devices, but it also has a place in reducing power consumption.

This driver allows the CPU frequency to dynamically change according to the CPU load or to user demands. Optionally, it is also possible to change the CPU core voltage depending on the selected frequency to reduce power consumption even more.

15.1 Hardware Operation

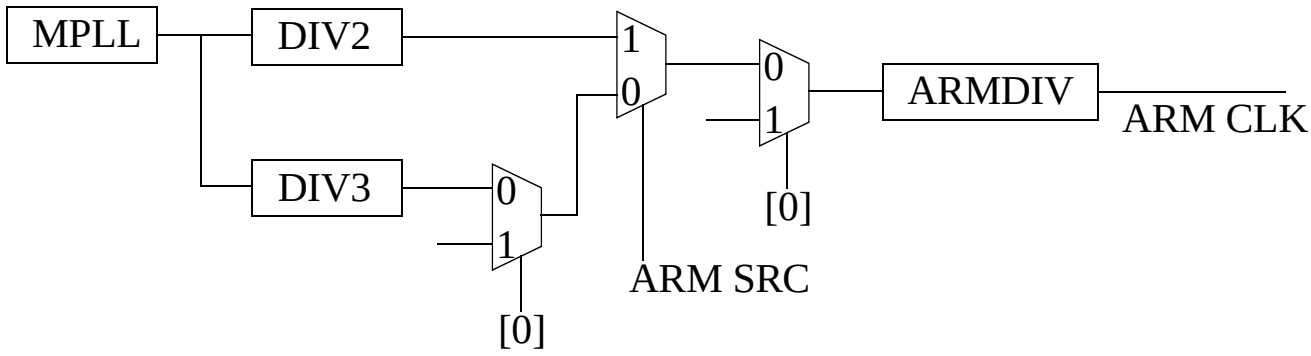


Figure 15-1. imx 27 CPU core clock diagram

The MPLL clock output 798 MHz signal. The ARMDIV block can divide by 1, 2, 3 and 4. So the available core frequency range is from 66.5 to 399 MHz. As the core frequency must be a multiple of the AHB bus frequency (133 MHz), the frequencies available for practical use are 133, 266 and 399 MHz.

Using the MC13783 power management driver, the CPU core voltage can be changed.

15.1.1 Software Operation

The imx27 dynamic frequency scaling driver is implemented as a CPUFreq driver (and registered using cpufreq_register_driver). All operations are done through the sysfs CPUFreq interface :

```
# cd /sys/devices/system/cpu/cpu0/cpufreq
```

To list availables governors :

```
# cat scaling_available_governors
conservative ondemand powersave userspace performance
```

To select a governor :

```
# echo <governor> > scaling_governor
```

Example to set the frequency to 133 MHz using the userspace governor :

```
# echo 133000 > scaling_setspeed
```

To see what is the current CPU frequency :

```
# cat cpuinfo_cur_freq
```

```
133000
```

How the governors work:

- performance sets the frequency to the maximum available
- powersave sets the lowest frequency
- userspace allows the user to set the frequency through sysfs
- ondemand changes the frequency depending on the system load. It increases the frequency quickly.
- conservative changes the frequency depending on the system load. It is more targeted at mobile devices and changes the frequency more graciously.

15.2 Source Code Structure

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

```
<ltib_dir>/rpm/BUILD/linux-2.6.22/arch/arm/mach-mx27/
```

Table 15-1. Source Code Files

File	Description
cpufreq.c	CPUFreq driver implementation.

15.3 Linux Menu Configuration Options

In CPU Frequency Scaling, select :

- **CPU Frequency Scaling**
- the governors you want to be able to use
- optionnally **enable voltage scaling for imx27** if you want to change the CPU voltage with the frequency

15.3.0.1 Board Configuration Options

There are no board configuration options for the Linux imx27 CPUFreq driver.

Chapter 16
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.

16.1 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 I2C 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 I2C client driver and a framebuffer driver in the Linux kernel. The I2C client driver implements the configurations to CH7024 registers through the I2C 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.

16.1.1 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 I2C-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.

16.1.2 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 I2C interface. The driver uses the common kernel I2C client driver to configure the CH7024 registers. The I2C client driver can not be accessed directly in user space.

The TV-out architecture diagram is shown in Figure 16-1.

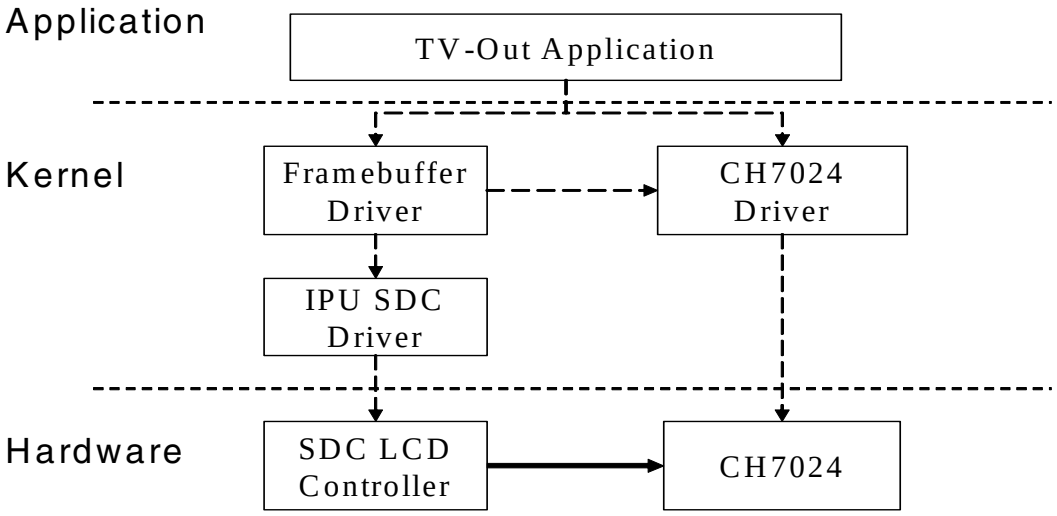


Figure 16-1. TVout Driver in the Architecture

16.2 Source Code Structure Configuration

Table 9-1 describes the source files associated with the TV-out driver, which are available in the directory `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/video/mxc`.

Table 16-1. TV-Out Driver Source File

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

Table 9-2 describes the source files source files associated with the framebuffer drivers which use TV-out driver are available in the directory `<ltib_dir>/rpm/BUILD/linux-/drivers/video/mxc`.

Table 16-2. Framebuffer Driver Source Files

File	Description
mx2fb.c	Source file for LCD framebuffer driver. Provides SDC LCD disable/enable interface to mx2fb_tv module for output device switching.

16.3 Linux Menu Configuration Options

The Linux kernel provides the configuration option for the CH7024 TV-Out driver. In `menuconfig`, this option is available under Device Drivers -> Graphics support -> Select TV out chip -> 3-stack CH7024 TV-Out Video.

Chapter 17
Video Processing Unit (VPU) Driver for MX27

The Video Processing Unit (VPU) is the multimedia video processing module. It supports a full duplex video codec with ~30 fps in VGA image resolution, multi-party call, and integrates multiple video processing standards together, which include H264 BP, MPEG4 SP and H263 P3 (including annex I, J, K and T).

It supports the following multimedia video stream processing features:

1. Multi-standard video codec
   a) MPEG-4 part-II simple profile encoding/decoding
   b) H.264/AVC baseline profile encoding/decoding
   c) H.263 P3 encoding/decoding
   d) Multi-party call: max processing 4 image/bitstream encoding and/or decoding simultaneously.
   e) Multi-format: for example, encodes MPEG4 bitstream, and decodes H.264 bitstream simultaneously.
2. Coding tools
   a) High-performance motion estimation.
      – Single reference frame for both MPEG4 and H.264 encoding.
      – Support 16 reference frame for H264 decoding.
      – Quarter-pel and half-pel accuracy motion estimation.
      – [+/-16, +/-16] Search range.
      – Unrestricted motion vector.
   b) All variable block sizes are supported (In case of encoding, 8x4, 4x8, and 4x4 block sizes are not supported).
   c) MPEG-4 AC/DC prediction & H.264 Intra prediction.
   d) H.263 Annex I, J, K(RS = 0 and ASO = 0), and T are supported. In case of encoding, the Annex I and K(RS = 1 or ASO = 1) are not supported.
   e) CIR (Cyclic Intra Refresh)/AIR (Adaptive Intra Refresh)
   f) Error resilience tools.
      – MPEG-4 re-synchronize marker & data-partitioning with RVLC (Fixed number of bits/macroblocks between macroblocks).
      – H.264/AVC FMO & ASO.
      – H.263 slice structured mode.
   g) Bit-rate control (CBR & VBR).
3. Pre/post rotation/mirroring
   a) 8 rotation/mirroring modes for image to be encoded.
   b) 8 rotation/mirroring modes for image to be displayed.

4. Programmability
   a) Embeds 16-bit DSP processor that is dedicated to processing bitstream and driving codec hardware.
   b) General purpose registers and interrupt generation for communication between system and Video Codec module.

17.1 Hardware Operation

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

The VPU hardware data flow can be shown by a mpeg4 decoder example, as shown in Figure 17-1.

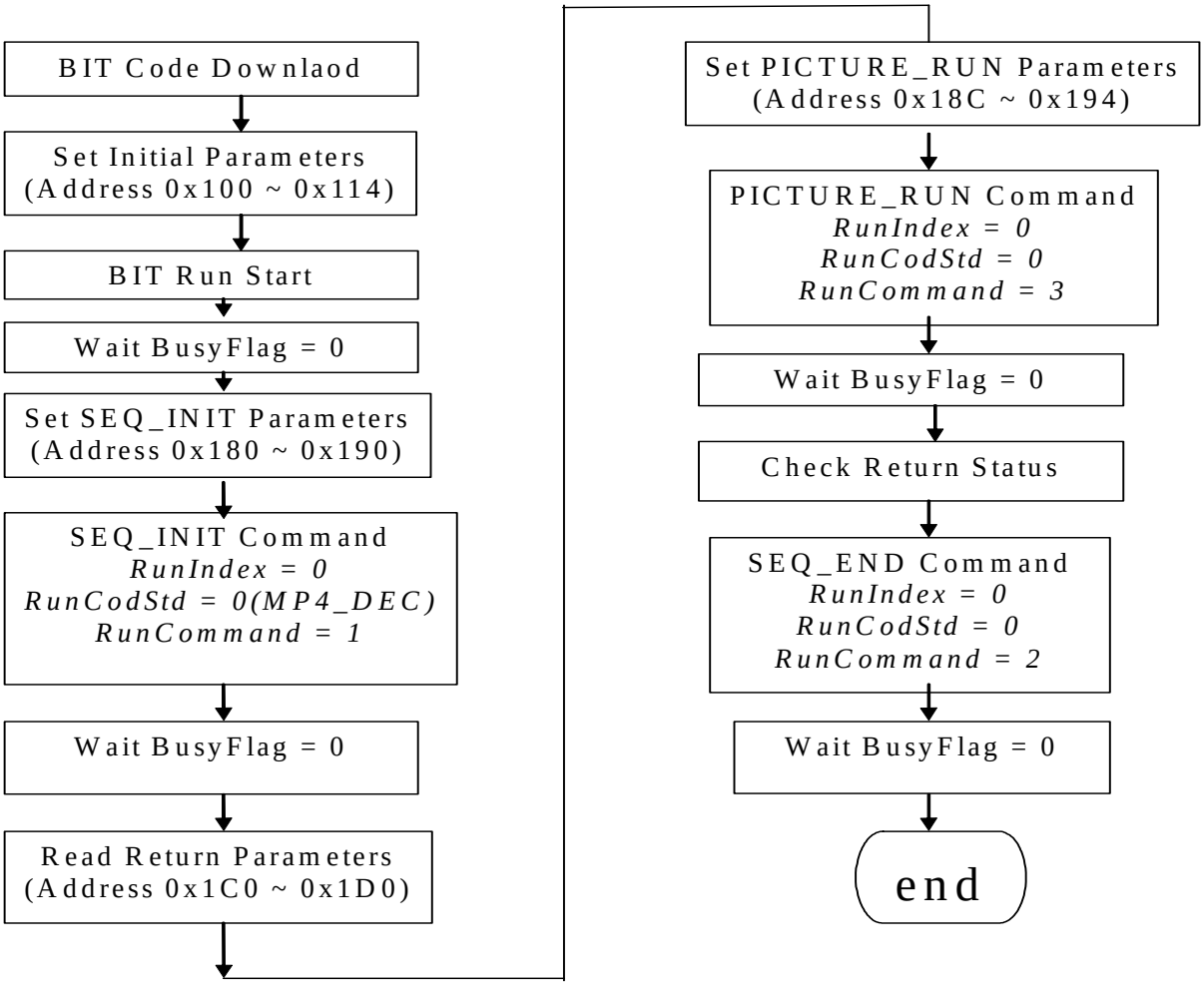


Figure 17-1. VPU Hardware Data Flow

17.2 Software Operation

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

The VPU kernel driver has the functions listed below:

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

The VPU user space driver has the functions listed below:

- Codec lib
  - Downloads executable bitcode for hardware
  - Initializes codec system
  - Sets codec system configuration
  - Controls codec system by command
  - Report codec status and result
- System IO operation
  - Request/Free memory
  - Map/Unmap memory/register to user space
  - Device management

17.3 Source Code Structure

Table 17-1 lists the kernel space source files available in the following files:

```
<ltib_dir>/rpm/BUILD/linux-2.6.22/include/asm-arm/arch-mxc/mxc_vpu.h
<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/mxc/vpu/mxc_vpu.c
```

Table 17-1. VPU Driver File List

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

17.4 Linux Menu Configuration Options

The Linux kernel configuration option, CONFIG_MXC_VPU, is provided for VPU driver. In menuconfig, this option is available under Device Drivers -> MXC support drivers -> MXC VPU (Video Processing Unit) support -> Support for MXC VPU (Video Processing Unit).

17.5 Programming Interface

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

Codec library APIs are as follows:

```
RetCode vpu_EncOpen(EncHandle * pHandle, EncOpenParam * pop);
RetCode vpu_EncClose(EncHandle encHandle);
RetCode vpu_EncGetInitialInfo(EncHandle encHandle, EncInitialInfo * initialInfo);
RetCode vpu_EncRegisterFrameBuffer(EncHandle encHandle, FrameBuffer * pBuffer, int num,
                                   int stride);
RetCode vpu_EncGetBitstreamBuffer(EncHandle handle, PhysicalAddress * prdPrt,
                                   PhysicalAddress * pwrPtr, Uint32 * size);
RetCode vpu_EncUpdateBitstreamBuffer(EncHandle handle, Uint32 size);
RetCode vpu_EncStartOneFrame(EncHandle encHandle, EncParam * pParam);
RetCode vpu_EncGetOutputInfo(EncHandle encHandle, EncOutputInfo * info);
RetCode vpu_EncGiveCommand (EncHandle pHandle, CodecCommand cmd, void * pParam);
RetCode vpu_DecOpen(DecHandle * pHandle, DecOpenParam * pop);
RetCode vpu_DecClose(DecHandle decHandle);
RetCode vpu_DecGetBitstreamBuffer(DecHandle pHandle, PhysicalAddress * pRdptr,
                          PhysicalAddress * pWrptr, Uint32 * size);
RetCode vpu_DecUpdateBitstreamBuffer(DecHandle decHandle, Uint32 size);
RetCode vpu_DecSetEscSeqInit(DecHandle pHandle, int escape);
RetCode vpu_DecGetInitialInfo(DecHandle decHandle, DecInitialInfo * info);
RetCode vpu_DecRegisterFrameBuffer(DecHandle decHandle, FrameBuffer * pBuffer, int num,
                                   int stride);
RetCode vpu_DecStartOneFrame(DecHandle handle, DecParam *param);
RetCode vpu_DecGetOutputInfo(DecHandle decHandle, DecOutputInfo * info);
RetCode vpu_DecBitBufferFlush(DecHandle handle);
RetCode vpu_DecGiveCommand(DecHandle pHandle, CodecCommand cmd, void * pParam);
```

System IO operations are listed as following.

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

For detailed function usage, please refer to the VPU parts of the doxygen files and the demo code under <ltib_dir>/rpm/BUILD/misc/test/mxc_vpu_test.

17.6 Defining an Application

The most important definition for an application is the codec memory descriptor. It is used for both request/free and mmap/munmap memory.

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

Chapter 18
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.

18.1 Hardware Operation

The OV2640FSL uses the serial camera control bus (SCCB) interface to control the sensor operation. It works as an I2C client, and the CSI interface of IPU works as the I2C master, which uses the I2C bus to control the camera's operation.

The CSI interface of IPU also provides the sensor clock to the camera when the camera is working so that the IPU can get image data from the camera through the CSI interface. The pixel clock, horizontal reference output, and vertical synchronization output generated from the camera are used by the CSI interface to get image data from the 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.

18.2 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.

18.3 Source Code Structure

Table 1-1 lists the camera driver source files available in the `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/media/video/mxc/capture` directory.

Table 18-1. Camera File List

File	Description
`ov2640.c`	camera driver implementation

18.4 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 19
Advanced Linux Sound Architecture (ALSA) Sound Driver with PMIC Hardware Support

NOTE

The i.MX27 PDK board doesn't support voice codec.

This section explains the Advanced Linux Sound Architecture (ALSA) driver in general. Additional documentation on ALSA can be located at www.alsa-project.org.

ALSA in general has the following components:

- ALSA `utils` (`Aplay. Arecord, Alsamixer`) – These are 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 – ALSA kernel drivers are 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`

19.1 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 below sections cover ALSA strictly as applicable to Freescale's platforms.

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)

19.1.1 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 and `/dev/sound/adsp` in OSS emulation mode. This interface is not available for i.MX27PDK.
- 8 kHz and 16 kHz mono recording on `/dev/snd/pcmC0D0c` in native mode and `/dev/sound/dsp` in OSS emulation 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:** Analog mixing is mixing the two streams in the analog domain after the DAC or the CODEC that is, after the streams have been decoded and before you pass it on to the output.
  - **Digital Mixing:** Digital mixing is mixing the 2 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.

19.1.2 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.

19.1.3 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.

19.2 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—Used to select 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—Used to transfer 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—Used to transmit and receive 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. The PMIC should be configured in master mode and the SSI in slave mode, 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.

19.3 Software Operation

In brief, the software performs the following steps:

19.3.1 Initialization

- Allocate sound card instance
- Create 2 PCM devices to support playback on ST-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 chips, 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.)

19.3.2 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.

19.3.3 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.

19.4 Source Code Structure

Table 19-1 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 19-1 are available in the `<ltib_dir>/rpm/BUILD/linux-2.6.22/sound/arm` directory in the Linux kernel source tree.

Table 19-1. 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 20
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.

20.1 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.

20.2 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 20-1 shows the block diagram for AUDMUX driver interactions.

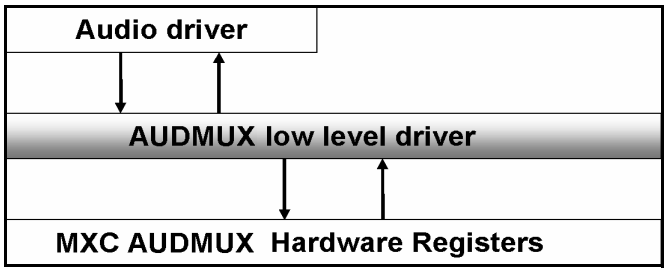


Figure 20-1. AUDMUX Driver Interactions

20.3 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.

20.4 Source Code Structure

Table 20-1 lists the source files available in the device directory:

`<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/mxc/dam`.

Table 20-1. 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

20.4.1 Linux Menu Configuration Options

The Linux kernel configuration option, CONFIG_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.

20.5 Programming Interface (Exported API)

The AUDMUX exported API allows the user to process standard AUDMUX operations.

Table 20-2. 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.

Table 20-2. AUDMUX Exported Functions (Continued)

Function	Description
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.

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.

20.6 Interrupt Requirements

No interrupts are generated by the digital audio multiplexer.

Chapter 21
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.

21.1 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 like 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). Over-sampling 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 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 over-sampling 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 Tx and Rx both).
- 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.

21.2 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 21-1 shows a block diagram of the software interaction.

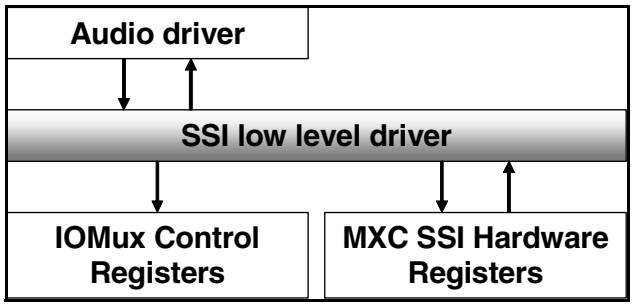


Figure 21-1. SSI Driver Interactions

21.3 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.

21.4 Source Code Structure

Table 21-1 lists the source files available in the devices directory:

`<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/mxc/ssi.`

Table 21-1. 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

21.4.1 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.

21.5 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 21-2. 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.
`ssi_interrupt_disable()`	This function disables the interrupts of the SSI module.
`ssi_interrupt_enable()`	This function enables the interrupts of the SSI module.

Table 21-2. SSI Exported Functions (Continued)

Function	Description
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.
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.

Table 21-2. SSI Exported Functions (Continued)

Function	Description
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.
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.

Table 21-2. SSI Exported Functions (Continued)

Function	Description
`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.

21.6 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 22
NAND Flash Memory Technology Device (MTD) Driver

22.1 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.

22.1.1 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 22.6, “Device-Specific Information.”

22.1.2 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 driver

- 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, 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/.

22.2 Requirements

This NAND Flash MTD driver implementation should meet the following requirements:

- The NAND MTD driver shall provide necessary hardware-specific information to the generic layer of the NAND MTD driver.
- The NAND MTD driver shall provide software Error Correction Code (ECC) support.
- The NAND MTD driver shall support both 16-bit and 8-bit NAND Flash
- The NAND MTD driver shall conform to the Linux coding standard.

22.3 Source Code Structure

Table 22-1 lists the source files available for the NAND MTD driver. These files are under the `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/mtd/nand` directory.

Table 22-1. 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

22.4 Configuration

The NAND MTD driver has the following Linux menu configuration options.

22.4.1 Linux Menu Configuration Options

In `menuconfig` the following options are available under `Device Drivers -> Memory Technology Device (MTD) support -> NAND Device Support -> MXC NAND`:

- 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.
- CONFIG_MTD_NAND_MXC_V3 - This is the configuration option for the NAND MTD driver for the MXC processors having NFC hardware version 3.

22.5 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.

22.6 Device-Specific Information

For more information on NFC hardware, refer to the L3 specifications of i.MX processors.Table 22-2 lists the NFC hardware version on different i.MX platforms.

Table 22-2. NFC Hardware Version across i.MX platforms

NFC Version	Platforms/SoC
1	i.MX27

Chapter 23
Low-Level Keypad Driver

The low-level keypad driver interfaces with the keypad port (KPP) in the i.MX application processors. The KPP interface in these processors is provided by the hardware. The low-level keypad driver is implemented as a standard Linux 2.6 keyboard driver, modified for the i.MX application processors.

The Low-Level 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 Low-Level Keypad Driver can be accessed through the `/dev/input/event0` device file.

23.1 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.

23.2 Software Operation

The low-level 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.

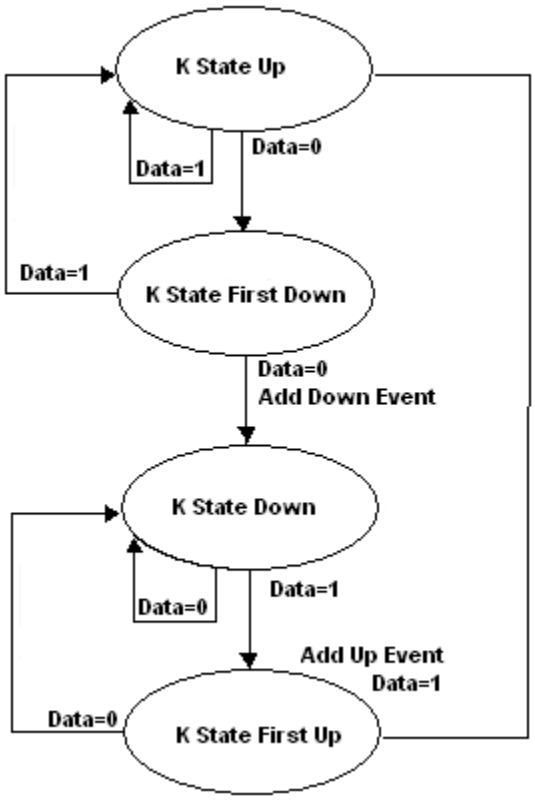


Figure 23-1. Keypad Driver State Machine

The low-level 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 Low-Level 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` executable, located in `<ltib_dir>/rpm/BUILD/linux-/drivers/input`, is compiled. The event interface is a generic input event interface. It passes the events generated in the kernel to the user space with timestamps. Blocking reads 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 and returns the time at which 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;
```

The following table describes key connection, key scancodes, and key mapcodes of the keys in the keypad.

Table 23-1. Key Connections in Keypad

Key	Row	Column	Scancode	Keycode
SEL	0	0	0	161
LEFT	0	1	1	103
DOWN	0	2	2	106
RIGHT	0	3	3	108
UP	0	4	4	105
KEY2	0	5	5	88
END	0	6	6	107
BACK	0	7	7	158
KEY1	1	0	8	59
SEND	1	1	9	145
HOME	1	2	10	102
APP1	1	3	11	64
VOL_UP	1	4	12	115
APP2	1	5	13	66
APP3	1	6	14	67
APP4	1	7	15	68
KEY_3	2	0	16	4
KEY_2	2	1	17	3
KEY_1	2	2	18	2
KEY_4	2	3	19	5
VOL_DOWN	2	4	20	114
KEY_7	2	5	21	8
KEY_5	2	6	22	6
KEY_6	2	7	23	7

Table 23-1. Key Connections in Keypad (Continued)

Key	Row	Column	Scancode	Keycode
KEY_9	3	0	24	10
Number_sign	3	1	25	87
KEY_8	3	2	26	9
ASTERISK	3	3	27	53
PLUS	3	4	28	78
RECORD	3	5	29	167
KEY_Q	3	6	30	16
KEY_W	3	7	31	17
KEY_A	4	0	32	30
KEY_S	4	1	33	31
KEY_D	4	2	34	32
KEY_E	4	3	35	18
KEY_F	4	4	36	33
KEY_R	4	5	37	19
KEY_T	4	6	38	20
KEY_Y	4	7	39	21
KEY_TAB	5	0	40	15
SYMB	5	1	41	65
CAPS	5	2	42	58
KEY_Z	5	3	43	44
KEY_X	5	4	44	45
KEY_C	5	5	45	46
KEY_V	5	6	46	47
KEY_G	5	7	47	34
KEY_B	6	0	48	48
KEY_H	6	1	49	35
KEY_N	6	2	50	49
KEY_M	6	3	51	50
KEY_J	6	4	52	36
KEY_K	6	5	53	37
KEY_U	6	6	54	22
KEY_I	6	7	55	23
SPACE	7	0	56	57

Table 23-1. Key Connections in Keypad (Continued)

Key	Row	Column	Scancode	Keycode
ON/OFF	7	1	57	64
PERIOD	7	2	58	52
ENTER	7	3	59	28
KEY_L	7	4	60	38
KEY_BS	7	5	61	14
KEY_P	7	6	62	25
KEY_O	7	7	63	24

Refer to the Device-Specific Information section for additional mapcodes and scancodes.

23.3 Requirements

The Keypad driver meets the following requirements:

- The keypad driver returns the input keycode for every key that is pressed or released.
- The keypad driver implements support for an interrupt driver for keypress or release.
- The keypad driver implements support for blocking and non-blocking reads.
- The keypad driver is implemented as a standard input device.

23.4 Source Code Structure

The source file, `mxc_keyb.c`, available in the `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/input/keyboard` directory. The file is a keypad lower level file.

The header file, `mxc_keyb.h`, is associated with the keypad driver, and is available in the `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/input/keyboard` directory.

23.5 Linux Menu Configuration Options

The following Linux kernel configurations are provided for 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 —Create the event interface. 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 MAXROW and MAXCOL macros defined in `include` the `mxc_keyb.h` file.
- Debounce delay: User can configure the debounce delay by changing the variable `KScanRate` defined in `mxc_keyb.c`

23.6 Programming Interface

All low-level keypad drivers define, typedefs, global variables, and functions are implemented in `mxc_keyb.c` and `mxc_keyb.h`. For more information, see the API documents for the programming interface.

23.7 Interrupt Requirements

Table 23-2 lists the keypad interrupt timer requirements.

Table 23-2. 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

23.8 Device-Specific Information

Table 23-3 shows key connections, key scancodes, and key mapcodes of the keys on the keypad for a specific platform.

Table 23-3. Key Connections for Keypad

Key	Row	Column	Scancode	Linux Key Code	Platform
EXTRA5	0	0	0	KEY_KP9	i.MX27 ADS
#	0	1	1	KEY_F3	i.MX27 ADS
0	0	2	2	KEY_0	i.MX27 ADS
*	0	3	3	KEY_KPASTERISK	i.MX27 ADS
RECORD	0	4	4	KEY_RECORD	i.MX27 ADS
POWER	0	5	5	KEY_POWER	i.MX27 ADS
EXTRA4	1	0	6	KEY_KP8	i.MX27 ADS
9	1	1	7	KEY_9	i.MX27 ADS
8	1	2	8	KEY_8	i.MX27 ADS
7	1	3	9	KEY_7	i.MX27 ADS
EXTRA1	1	4	10	KEY_KP5	i.MX27 ADS
VOL_DOWN	1	5	11	KEY_VOLUMEDOWN	i.MX27 ADS
EXTRA3	2	0	12	KEY_KP7	i.MX27 ADS

Table 23-3. Key Connections for Keypad (Continued)

Key	Row	Column	Scancode	Linux Key Code	Platform
6	2	1	13	KEY_6	i.MX27 ADS
5	2	2	14	KEY_5	i.MX27 ADS
4	2	3	15	KEY_4	i.MX27 ADS
APP4	2	4	16	KEY_KP4	i.MX27 ADS
VOL_UP	2	5	17	KEY_VOLUMEUP	i.MX27 ADS
EXTRA2	3	0	18	KEY_KP6	i.MX27 ADS
3	3	1	19	KEY_3	i.MX27 ADS
2	3	2	20	KEY_2	i.MX27 ADS
1	3	3	21	KEY_1	i.MX27 ADS
APP3	3	4	22	KEY_KP3	i.MX27 ADS
UP	0	0	0	KEY_UP	.MX27 PDK
DOWN	3	5	23	KEY_DOWN	i.MX27 ADS
DOWN	0	1	1	KEY_DOWN	MX27 PDK
BACK	4	0	24	KEY_BACK	i.MX27 ADS
RIGHT	4	1	25	KEY_RIGHT	i.MX27 ADS
RIGHT	1	0	8	KEY_RIGHT	MX27 PDK
ACTION	4	2	26	KEY_ENTER	i.MX27 ADS
LEFT	4	3	27	KEY_LEFT	i.MX27 ADS
LEFT	1	1	9	KEY_LEFT	MX27 PDK
HOME	4	4	28	KEY_HOME	i.MX27 ADS
APP2	4	5	29	KEY_KP2	i.MX27 ADS
END	5	0	30	KEY_END	i.MX27 ADS
KEY2	5	1	31	KEY_F2	i.MX27 ADS
UP	5	2	32	KEY_UP	i.MX27 ADS
KEY1	5	3	33	KEY_F1	i.MX27 ADS
SEND	5	4	34	KEY_F4	i.MX27 ADS
APP1	5	5	35	KEY_KP1	i.MX27 ADS
ENTER	1	2	10	KEY_ENTER	MX27 PDK
MENU1	2	0	16	KEY_F6 (APP1)	MX27 PDK
MENU2	2	1	17	KEY_F8 (APP2)	MX27 PDK
MENU3	2	2	18	KEY_F9 (APP3)	MX27 PDK
MENU4	2	3	19	KEY_F10 (APP4)	MX27 PDK

Chapter 24
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-2.6.22/drivers/net/Space.c` to probe for the device and to initialize it during boot.

24.1 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.

24.2 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

24.3 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.

24.4 Source Code Structure

Table 24-1 lists the source files available in the `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/net` directory:

Table 24-1. Ethernet File List

File	Description
`smsc911x.h`	Header file defining registers.
`smsc911x.c`	Linux driver for Ethernet LAN controller.

24.5 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 25
Fast Ethernet Controller (FEC) Driver

The Fast Ethernet Controller (FEC) driver performs the full set of IEEE 802.3/Ethernet CSMA/CD media access control and channel interface functions. The FEC requires an external interface adaptor and transceiver function to complete the interface to the Ethernet media. It supports half- or full-duplex operation on 10 Mbps- or 100 Mbps-related Ethernet networks.

The FEC driver has the following features:

- Supports full duplex operation
- Supports link status change detect
- Supports auto-negotiation (determines the network speed and full or half-duplex operation)
- Supports transmit features like automatic retransmission on collision and automatic CRC generation
- Supports obtaining statistics from the device, such as transmit collisions

This network adapter can be accessed through the `ifconfig` command with the interface name (that is, `eth0`). The driver will auto-probe the external adaptor (PHY device).

25.1 Hardware Operation

The FEC is an Ethernet controller interfaces the system to the LAN network. The FEC supports three different standard MAC-PHY (physical) interfaces for connection to an external Ethernet transceiver. The FEC supports the 10/100 Mbps MII and the 10 Mbps-only 7-wire interface, which uses a subset of the MII pins.

A brief overview of the device functionality is provided here. For details, see the *FEC Specification and the Ethernet Transceiver Data Sheet*.

The FEC supports both an MII interface for 10/100 Mbps Ethernet and a 7-wire serial network interface (SNI) for 10 Mbps Ethernet. In MII mode, there are 18 signals defined by the IEEE 802.3 standard and supported by the EMAC. SNI mode uses the subset of the 18 signals. These signals are listed in Table 25-1.

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

Direction	EMAC Pin Name	MII Usage	SNI Usage
in/out	FEC_MDIO	Management Data Input/Output	General I/O
out	FEC_MDC	Management Data Clock	General output
out	FEC_TXD[0]	Data out, bit 0	Data out
out	FEC_TXD[1]	Data out, bit 1	General output
out	FEC_TXD[2]	Data out, bit 2	General output
out	FEC_TXD[3]	Data out, bit 3	General output
out	FEC_TX_EN	Transmit Enable	Transmit Enable
out	FEC_TX_ER	Transmit Error	General output

Table 25-1. Pin Usage in MII and SNI Modes (Continued)

Direction	EMAC Pin Name	MII Usage	SNI Usage
in	FEC_CRS	Carrier Sense	Carrier Sense
in	FEC_COL	Collision	Collision
in	FEC_TX_CLK	Transmit Clock	Transmit Clock
in	FEC_RX_ER	Receive Error	General input
in	FEC_RX_CLK	Receive Clock	Receive Clock
in	FEC_RX_DV	Receive Data Valid	Not Used
in	FEC_RXD[0]	Data in, bit 0	Data in
in	FEC_RXD[1]	Data in, bit 1	General input
in	FEC_RXD[2]	Data in, bit 2	General input
in	FEC_RXD[3]	Data in, bit 3	General input

The MII management interface consists of two pins, FEC_MDIO and FEC_MDC. These pins are configured through GPIO setting.

- **Transmission**—The Ethernet transmitter is designed to work with almost no intervention from software. Once ECR[ETHER_EN] is asserted and data appears in the transmit FIFO, the Ethernet MAC is able to transmit onto the network. When the transmit FIFO fills to the watermark (defined by the TFWR), the MAC transmit logic will assert FEC_TX_EN and start transmitting the preamble (PA) sequence, the start frame delimiter (SFD), and then the frame information from the FIFO. However, the controller defers the transmission if the network is busy (FEC_CRS asserts). Before transmitting, the controller waits for carrier sense to become inactive, then determines if carrier sense stays inactive for 60 bit times. If the transmission begins after waiting an additional 36 bit times (96 bit times after carrier sense originally became inactive). Both buffer (TXB) and frame (TXF) interrupts may be generated as determined by the settings in the EIMR.
- **Reception—**The FEC receiver is designed to work with almost no intervention from the host and can perform address recognition, CRC checking, short frame checking, and maximum frame length checking. When the driver enables the FEC receiver by asserting ECR[ETHER_EN], it will immediately start processing receive frames. When FEC_RX_DV asserts, the receiver will first check for a valid PA/SFD header. If the PA/SFD is valid, it will be stripped and the frame will be processed by the receiver. If a valid PA/SFD is not found, the frame will be ignored. In MII mode, the receiver checks for at least one byte matching the SFD. Zero or more PA bytes may occur, but if a 00 bit sequence is detected prior to the SFD byte, the frame is ignored. After the first 6 bytes of the frame have been received, the FEC performs address recognition on the frame. During reception, the Ethernet controller checks for various error conditions and once the entire frame is written into the FIFO, a 32-bit frame status word is written into the FIFO. This status word contains the M, BC, MC, LG, NO, CR, OV, and TR status bits, and the frame length. Receive Buffer (RXB) and Frame Interrupts (RXF) may be generated if enabled by the EIMR register. When the receive frame is complete, the FEC sets the L-bit in the RxBD, writes the other frame status bits into the RxBD, and clears the E-bit. The Ethernet controller next generates a maskable interrupt (RXF bit

in EIR, maskable by RXF bit in EIMR), indicating that a frame has been received and is in memory. The Ethernet controller then waits for a new frame.

- **Interrupt management**—When an event occurs that sets a bit in the EIR, an interrupt will be generated if the corresponding bit in the interrupt mask register (EIMR) is also set. The bit in the EIR is cleared if a one is written to that bit position; writing zero has no effect. This register is cleared upon hardware reset. These interrupts can be divided into operational interrupts, transceiver/network error interrupts, and internal error interrupts. Interrupts which may occur in normal operation are GRA, TXF, TXB, RXF, RXB, and MII. Interrupts resulting from errors/problems detected in the network or transceiver are HBERR, BABR, BABT, LC, and RL. Interrupts resulting from internal errors are HBERR and UN. Some of the error interrupts are independently counted in the MIB block counters. Software may choose to mask off these interrupts as these errors will be visible to network management through the MIB counters. For PHY interrupt, which is interfaced through PBC (CPLD), it is optional for link status detect.

25.2 Software Operation

The FEC 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, such as `fec_enet_start_xmit` and for reception of Ethernet packets through the ISR, such as `fec_enet_interrupt`.
- Interrupt servicing routine—Supports events, such as TXF, RXF and MII.
- Miscellaneous routines—Different routines come under this category, such as `fec_timeout` for waking up network stack and `fec_enet_get_stats` to obtain statistics for the device.

25.3 Source Code Structure

Table 25-2 lists the source files available in the `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/net` directory.

Table 25-2. Ethernet File List

File	Description
`fec.h`	Header file defining registers.
`fec.c`	Linux driver for Ethernet LAN controller.

For more information about the generic Linux driver, see the `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/net/fec.c` source file.

25.4 Linux Menu Configuration Options

The Linux kernel configuration option, CONFIG_FEC, is provided for this module. This is the Ethernet driver used for the FEC. In `menuconfig`, this option is available under Device Drivers -> Network device support -> Ethernet (10 or 100Mbit) -> FEC ethernet controller.

25.5 Programming Interface

Table 25-2 lists the source files for the FEC Driver. The following sections show modifications that were required in the original Ethernet driver source for porting it to the i.MX family multimedia application processors.

25.5.1 Device-Specific Defines

Device-specific defines are added to the header file (`fec.h`) and they provide common board configuration options.

`fec.h` defines the struct for the register access and the struct for the buffer descriptor.

```
/*
 *      Define the buffer descriptor structure.
 */
typedef struct bufdesc {
        unsigned short  cbd_datlen;             /* Data length */
        unsigned short  cbd_sc;                 /* Control and status info */
        unsigned long   cbd_bufaddr;            /* Buffer address */
} cbd_t;
/*
 *      Define the register access structure.
 */
typedef struct fec {
        unsigned long   fec_reserved0;
        unsigned long   fec_ievent;             /* Interrupt event reg */
        unsigned long   fec_imask;              /* Interrupt mask reg */
        unsigned long   fec_reserved1;
        unsigned long   fec_r_des_active;       /* Receive descriptor reg */
        unsigned long   fec_x_des_active;       /* Transmit descriptor reg */
        unsigned long   fec_reserved2[3];
        unsigned long   fec_ecntrl;             /* Ethernet control reg */
        unsigned long   fec_reserved3[6];
        unsigned long   fec_mii_data;           /* MII manage frame reg */
        unsigned long   fec_mii_speed;          /* MII speed control reg */
        unsigned long   fec_reserved4[7];
        unsigned long   fec_mib_ctrlstat;       /* MIB control/status reg */
        unsigned long   fec_reserved5[7];
        unsigned long   fec_r_cntrl;            /* Receive control reg */
        unsigned long   fec_reserved6[15];
        unsigned long   fec_x_cntrl;            /* Transmit Control reg */
        unsigned long   fec_reserved7[7];
        unsigned long   fec_addr_low;           /* Low 32bits MAC address */
        unsigned long   fec_addr_high;          /* High 16bits MAC address */
        unsigned long   fec_opd;                /* Opcode + Pause duration */
        unsigned long   fec_reserved8[10];
        unsigned long   fec_hash_table_high;    /* High 32bits hash table */
        unsigned long   fec_hash_table_low;     /* Low 32bits hash table */
        unsigned long   fec_grp_hash_table_high;/* High 32bits hash table */
        unsigned long   fec_grp_hash_table_low; /* Low 32bits hash table */
        unsigned long   fec_reserved9[7];
        unsigned long   fec_x_wmrk;             /* FIFO transmit water mark */
        unsigned long   fec_reserved10;
        unsigned long   fec_r_bound;            /* FIFO receive bound reg */
        unsigned long   fec_r_fstart;           /* FIFO receive start reg */
```

```
        unsigned long   fec_reserved11[11];
        unsigned long   fec_r_des_start;        /* Receive descriptor ring */
        unsigned long   fec_x_des_start;        /* Transmit descriptor ring */
        unsigned long   fec_r_buff_size;        /* Maximum receive buff size */
} fec_t;
```

25.5.2 How to get a MAC Address?

```
static void __inline__ fec_get_mac(struct net_device *dev)
```

This gets the MAC address through IIM (IC Identification) by default for MX27. If the MAC address is not programmed, the driver will set the MAC address to 0x00:0x00:0x00:0x00:0x00:0x00:0x00: , which would not be acceptable. The MAC address can also be set by the REDBOOT command `fconfig`.

```
exec -c "noinitrd console=ttymxc0 root=/dev/nfs
nfsroot=10.192.223.211:/tools/rootfs/rootfs fec_mac=00:04:9f:00:98:2c rw ip=dhcp"
```

Chapter 26
Symmetric/Asymmetric Hashing and Random Accelerator (SAHARA) Drivers

This document provides an overall architectural view of the Sahara 2 driver. It presents how the driver is to be organized and the processes that must be handled to make the driver work.

26.1 Overview

This document adheres to the requirements specified in [1]. Although intended to be expandable and portable, a change in requirements could alter how operations are performed.

For example, if a requirement to use a black key were added, several possibilities would immediately occur.

1. If individual tasks could only handle red or black keys, they would register as a red or black key holder and little of the architecture would change.
2. If they could handle either, key information would have to accompany the key. Possibilities are:
   — A new object would be passed with the descriptor pointer.
   — The key object would pass with the descriptor.
   — The descriptors would be built in kernel space instead of user space, meaning that all user space parameters would be passed to kernel mode.

As such, this document becomes the proposed approach, conceptually, but not a constraint, as changes occur during design and implementation, requirement modifications happen, and familiarity with the hardware becomes better understood.

26.2 Software Operation

26.2.1 API Notes

- Kernel users should not use 'blocking mode' unless the code is doing work on behalf of the kernel process which needs to sleep. The reason is that blocking mode will attempt to put the current process to sleep. This means that blocking mode cannot be used from 'bottom half' code nor from interrupt code.
- Kernel users must provide a `kmalloc'ed` buffer address for all data types (key structures, context structures, input/output buffers, and so on)
- User-mode users should beware of (or even avoid) doing I/O to the stack, as cache line boundaries can cause problems. This can even be true for such simple things as having a context object on the stack, or retrieving a random number into a `uint32_t` stack variable. This goes for key structures, context structures, and input/output buffers.

26.2.2 Architecture

The conceptual model is shown in Figure 26-1.

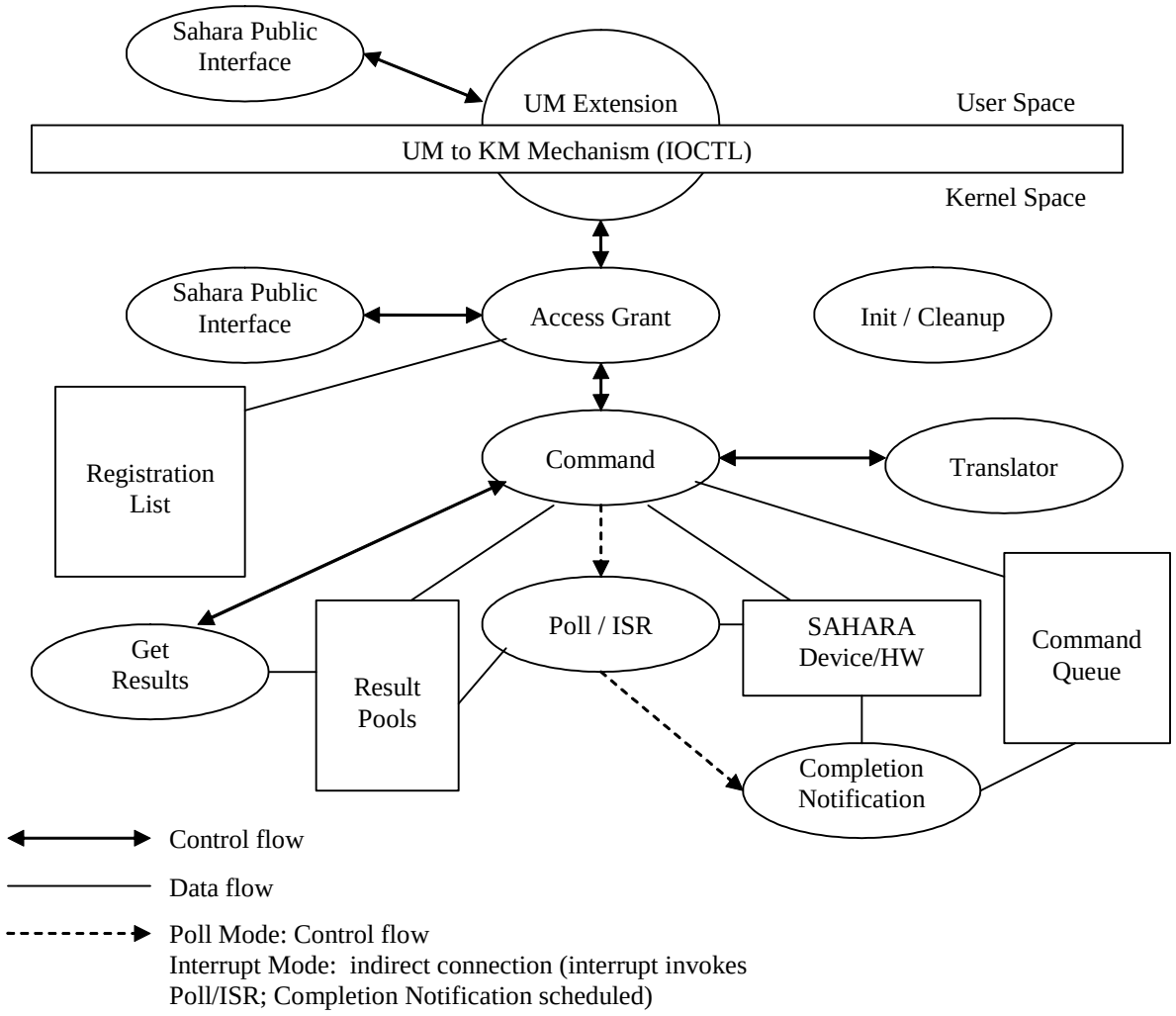


Figure 26-1. Architecture Overview

All of the processes in Figure 26-1 are implemented as common code, except for some or all of the following platform-centric processes: UM Extension, `Init/Cleanup`, Translator, and Completion Notification.

The driver operates in poll or interrupt mode, based on how the code is built (compile time option). The modes are mutually exclusive.

26.2.2.1 Registration List

This maintains a list of the tasks that are registered with the driver.

26.2.2.2 Command Queue

This maintains a list of commands (pointers to descriptor chains and their associated user) destined for the SAHARA Device/HW. A pointer is maintained to the current (active) command as well as where the next command is to be entered into the queue.

26.2.2.3 Result Pools

This maintains a list of completed commands. After the SAHARA Device/HW completes, the status, along with its user association is placed in this pool.

26.2.2.4 SAHARA Device / HW

This is the Sahara 2 hardware.

26.2.2.5 Init / Cleanup

This process is invoked by the OS to do the following tasks:

- Initialize the driver when the OS wishes to start the driver
- Clean up the driver when the OS wishes to shut it down.

To initialize the driver:

1. Map Sahara registers into kernel space
2. Check that Sahara's version number is 2
3. Attach handler to Sahara interrupt line (top half) if in interrupt mode
4. Initialize Sahara's interrupt
5. Seed the random number generator. Note that in the Control register that the RNG Auto Reseed cannot be set at startup (Hardware Erratum for RNG reseed). The available choices are:
   — Set this bit after the first random number is obtained
   — Never set this bit but rather detect when the `RNG Reseed Req` bit is set in the Status register and put the RNG in Seed Generation Mode. This is to be used for this architecture
   — Check for the reseed and set auto reseed when it becomes true
6. Install the tasklet (bottom half), if in interrupt mode (that is, install Complete Notification process)
7. Set up pointers to the command queue, the result pool, and the registration list
8. Populate the Capability Object with (most of this can be done at compile time) as follows:
   — Sahara version (2)
   — Command queue size
   — Result pool size
   — Registration list size
   — Driver version number
   — Algorithms & modes
   — Whatever else makes sense

9. Zero/initialize registration list, command queue, and result pool
10. Register as a device (to `IOCTL`)
11. If a failure is encountered anywhere, within the Init subprocess, it is terminated, the Cleanup subprocess is initiated, and an error is returned to the OS

To clean up a process:

1. Unregister as a device (to `IOCTL`)
2. Uninstall interrupt handler (top half) if in interrupt mode
3. Uninstall tasklet (bottom half) if in interrupt mode
4. Reset Sahara (leaves Sahara's interrupt disabled)
5. Null pointers to command queue, result pool, and registration list

26.2.2.6 Sahara Public Interface

This is the only access users are permitted to the Sahara functionality. (Refer to the API document included in doxygen format.) The interface is the same in both the User and Kernel Space.

- Converts service requests into descriptor chains, for those requests that required descriptors
  - For 'final' block of data, ensure that it is consumed correctly (hardware erratum for buffer length issue)
  - Descriptor pointers are created based on information in the SKO, HCO, and/or SCCO
  - Descriptor pointers reference input and output data buffers, fields in the SKO, HCO, and/or SCCO
- Returns error if input parameters are inconsistent or otherwise in error
- Passes pointers to a descriptor chain and a UCO to the next process
- Receives status information from the rest of the driver to return through the API's return value
- Passes raw descriptor chains through (still must be registered, that is, have a UCO). It will be necessary to determine if the hardware erratum for buffer length issue applies to this descriptor chain and, if so, modify the chain appropriately
- Passes information into, and receives from, the Access Grant or UM Extension process, as appropriate

26.2.2.7 UM Extension

When the Sahara public interface is built for user space, the user mode extension is included in the build to provide a way for the user space sahara public interface to communicate into kernel mode.

- Transports the information passed from the Sahara Public Interface from User to Kernel space and back
- Passes information into, and receives from, the Access Grant process
- Passes information into, and receives from, the Sahara Public Interface in User Space
- When signaled by the Completion Notification process, invokes the User's callback routine (the callback was acquired during registration)

26.2.2.8 Access Grant

All tasks must be registered with the driver before being able to request services.

- If this is a registration request, do the following:
  - Enter user information in Registration List, such as: ID, maximum number of outstanding commands possible (Result Pool size requested)
  - Populate its User Context Object
  - Return ‘success’ or ‘failure’ as appropriate
- If this is a deregistration request and the user is not registered, return ‘never registered’
- If this is a deregistration request and the user is registered
  - Remove user information in Registration List
  - Depopulate its User Context Object
  - Return ‘success’ or ‘failure’ as appropriate
- If this is not a registration or deregistration request (that is, any other User request), access the Registration List to see if the requestor is registered with the driver
  - If User is registered, pass the request to the Command process
  - If the User is not registered, return ‘unregistered’ error to the requesting process

26.2.2.9 Command

The Command determines what service was requested and directs the driver to fulfill that service. The system determines whether it is a service that the driver can fulfill without the use of the Sahara hardware or not.

For requests that involve Sahara hardware, the system does the following:

1. Checks that there is room in the Command Queue. If not, returns a ‘queue full’ status and terminates.
2. Checks that there’s room in the Result Pool. If not, returns a ‘pool full’ status and terminates.
3. Checks that this user has not reached its maximum number of outstanding requests. If it has, returns ‘request limit reached’.
4. Invokes the Translator process to convert received memory addresses.
5. If Command Queue is empty, enters descriptor chain pointer into the Sahara’s Descriptor Address Register (DAR). This starts the processing of descriptors which continues until the Command Queue is empty.
6. Enters the UCO and descriptor pointer in Command Queue, and whatever additional information may be needed, to await execution.
7. Checks if the `RNG Reseed Req` bit is set in the Status register and, if so, puts an RNG Reseed descriptor into the command queue to reseed the RNG.

User Blocking/Non-blocking requests are handled as listed Table 26-1.

Table 26-1. Blocking / Non-Blocking Definitions

Feature	Driver Poll Mode	Driver Interrupt Mode
User Blocking	User waits for request completion. Driver never gives up processor.	User waits for request completion. Driver queues request, suspends calling task, and releases processor (on completion the Driver un-suspends task / returns)
User Non-blocking w callback	Driver never gives up processor, therefore User is blocked until request completion (the callback will be invoked prior to request completion)	Driver queues request and returns. Upon request completion, the callback is invoked
User Non-blocking w no callback	Driver never gives up processor, therefore User is blocked until request completion	Driver queues request and returns. (User is not given any indication of completion. It enters a User Poll mode and polls for the results)

8. If in polling mode, transfers control to Poll/ISR process. (If in interrupt mode, the Poll/ISR process will be invoked through the interrupt mechanism.)

For requests that do not involve Sahara hardware, the system processes the request and immediately returns to the user. For example, if a ‘get results’ request is received, the list is populated with the user results and the driver returns to the user.

26.2.2.10 Translator

The translator has the following features:

- Translates pointer addresses from virtual addresses to physical addresses
- Ensure that blocks of data that have become fragmented due to page discontinuity are handled with links in the descriptor chain
- Lock pages so addresses remain stable
- Clear processor cache

26.2.2.11 Poll / ISR

Polling has the following features:

1. If polling, continuously check if operation is done (poll the State field in the Status Register) to determine when the SAHARA Device/HW has completed
2. Move the content of the in-progress element from the Command Queue to the Result Pool

3. Copy Sahara's Status and Error Status registers into the result pool
4. Write `Clr_Error` in Command register and flag as FAILED if State field in Status register is 010 or 110 (otherwise set to PASSED)
5. Load next command into Sahara, if one exists (keep Sahara loaded with two commands at a time if possible)
6. If in Interrupt mode, schedule tasklet (bottom half) process Completion Notification (that is, place it in the ready queue)
7. If polling, transfer control to process Completion Notification

26.2.2.12 Completion Notification

Using compiler switches, this will be built to run as a tasklet or be invoked as a function.

1. Invoke callback function, available in UCO, if in interrupt mode and callbacks are not suppressed by user
   - If the User is in Kernel Space, invoke callback
   - If the User is in User Space, signal UM Extension to invoke callback
2. Clean up memory, flags, and so on as needed
3. Unlock pages

26.2.2.13 Get Results

When a Get Results request is received (a request that does not involve Sahara hardware), the following are performed and the result is immediately returned to the user:

1. If no results are found for this user, return 'no results found' status
2. If at least one result is found for the user, populate the user supplied result list in whatever order the results are found in the Result Pool (return the lesser of the max number of requests or number of results in pool)
3. If the status is PASSED (set by Poll/ISR process), check and return the following:
   - Failed if State field in Status register is 011. Also sends notification to SAHARA Public Interface to reject all future calls to SAHARA driver.
   - Failed if SCC Fail bit in Status register is set.
   - Specific descriptor error from Error Source field in the Error Status register, if the Error bit in the Status register is set.
   - Failed if Error bit in Status register is set and the Error Source field in the Error Status register shows No Error.
   - Passed otherwise.
4. Clear result pool entry.
5. Adjust number of outstanding requests.

26.2.3 Symmetric Descriptors

The following demonstrates the descriptors needed to perform the symmetric operations required of the Sahara drivers.

26.2.3.1 Overview

All headers are defined in the SAHARA chapter of the *MCIMX27 Applications Processors Reference Manual*. For this section, the "Descriptor Header Format" figure in SAHARA chapter of the *MCIMX27 Applications Processors Reference Manual* assumes the following settings:

- All shaded ("Unimplemented or Reserved") bits are set to zero (the hardware will flag it as an error if not)
- The CHA is SKHA
- The LLO bit is set
- All Form values are available in the "Descriptor Header Encoding" table
- The Parity bit is used to force the header to odd parity
- The Disable Key Parity field is clear to force check for key parity errors for DES and TDES

Placing a null where an IV or key is expected leaves the internal IV and key registers at whatever value they currently contain. When the chip is reset or when a descriptor chain completes, the internal symmetric IV and key registers are filled with zeros.

In SKO, you can locate which algorithm is being requested. The mode being requested is located in the SCCO (note that 'mode' is not applicable to ARC4).

Unless otherwise noted, the order of the descriptors is as they would appear in a descriptor chain for the mode or operation indicated.

26.2.3.2 ECB

For ECB, follow these steps:

1. Set context (key & mode)

```
Desc # 1
Header encrypt/decrypt:
0x91880005/0x11880001 DES key length 64 bit
0x91880006/0x11880002 TDES key length 128 (two key) or 192 (three key) bit
0x11880004/0x91880000 AES key length 128 bit (only 128 bit supported by Sahara2)
Length 1: zero (not applicable)
Pointer 1: null (not applicable)
Length 2: key length obtained from SKO
Pointer 2: key is found in the SKO
```

2. Encrypt/decrypt:

```
Desc # 4
Header: 0x11850000
Length 1: plaintext/ciphertext length value that was passed in
Pointer 1: points to the plaintext/ciphertext buffer that was passed in
Length 2: ciphertext/plaintext length value is same as plaintext/ciphertext length
Pointer 2: points to the ciphertext/plaintext buffer that was passed in
```

26.2.3.3 CBC

For CBC, follow these steps:

1. Set context (key, mode, & IV)

```
Desc # 1
Header encrypt/decrypt:
0x9188000D/0x11880009 DES IV & key length 64 bit
0x1188000F/0x9188000B TDES IV length 64 bit; key length 128 (2 key) or 192 (3 key) bit
0x9188000C/0x11880008 AES IV & key length 128 bit, only 128 bit key supported by
Sahara2
Length 1: IV length obtained from: SCCO (128 bit) if Load flag is set; zero if Init
flag is set
Pointer 1: IV is found in: SCCO if Load flag is set; null (left zeros) if Init flag
is set
Length 2: key length obtained from SKO
Pointer 2: key is found in the SKO
```

2. Encrypt/decrypt

```
Desc # 4
Header: 0x11850000
Length 1: plaintext/ciphertext length value that was passed in
Pointer 1: points to the plaintext/ciphertext buffer that was passed in
Length 2: ciphertext/plaintext length value is same as plaintext/ciphertext length
Pointer 2: points to the ciphertext/plaintext buffer that was passed in
```

3. Save context

   If the save bit is set in the SCCO, save context (IV).

```
Desc # 5
Header: 0x91820000
Length 1: zero (not applicable)
Pointer 1: null (not applicable)
Length 2: IV length obtained from SCCO (64 bit)
Pointer 2: IV put in SCCO (write over previous value)
```

26.2.3.4 CTR

For CTR, follow these steps:

1. Set context (key, mode, & IV)

   The counter modulus is available in the SCCO. The header value shown here is arbitrarily chosen to be 2^8 (CTR Modulus Bits are all clear).

```
Desc # 1
Header encrypt/decrypt:
0x1188001C/0x91880018AES
Length 1: IV length obtained from: SCCO (128 bit) if Load flag is set; zero if Init
flag is set
Pointer 1: IV is found in: SCCO if Load flag is set; null (left zeros) if Init flag
is set
Length 2: key length obtained from SKO (only 128 bit key supported by Sahara2)
Pointer 2: key is found in the SKO
```

2. Encrypt/decrypt

```
Desc # 4
Header: 0x11850000
Length 1: plaintext/ciphertext length value that was passed in
Pointer 1: points to the plaintext/ciphertext buffer that was passed in
```

```
Length 2: ciphertext/plaintext length value is same as plaintext/ciphertext length
Pointer 2: points to the ciphertext/plaintext buffer that was passed in
```

3. Save context

   If the save bit is set in the SCCO, save context (IV).

```
Desc # 5
Header: 0x91820000
Length 1: zero (not applicable)
Pointer 1: null (not applicable)
Length 2: IV length obtained from SCCO (128 bit)
Pointer 2: IV put in SCCO (write over previous value)
```

26.2.3.5 CCM Encrypt

Note that the block size is 128 bits (16 bytes).

For CCM encrypt, follow these steps:

1. Set context (`Ctr` mode, modulus, Ctr_0, & Key)

   The modulus value is 2^128 for CCM.

```
Desc # 1
Header:
0x11881E1C  AES
Length 1: Ctr_0 length obtained from ACCO
Pointer 1: Ctr_0 value obtained from ACCO
Length 2: key length obtained from SKO (only 128 bit supported by Sahara2)
Pointer 2: key is found in the SKO
```

2. Adjust context (increments Ctr0 to Ctr_1)

```
Desc # 4
Header: 0x11850000
Length 1: don't care (1 byte <= length <= 1 block)
Pointer 1: don't care (any input)
Length 2: same as Length 1
Pointer 2: don't care (any reasonable output location)
```

3. Compute MAC (CBC mode)

   The key was set in **Set Context** (step 1), and the IV is zero due to chip reset or previous descriptor chain completion.

```
Desc # 2
Header:0x108D000C  AES
Length 1: B length ((u+1) * 16 bytes)
Pointer 1: link to B_0
Length 2: Y length (same as Length 1)
Pointer 2: Y

Link to B_0

Length 1: B_0 length (16 bytes)
Pointer 1: B_0 passed in
Next 1: Link to B_associated_data

Link to B_associated_data

Length 2: B_ad length passed in (u * 16 bytes)
Pointer 2: B_ad passed in
```

```
Next 2: Null
```

4. Compute Ciphertext (CCM mode)

   Counter (`Ctr`) is already set from step 2.

```
Desc # 2
Header encrypt:0x918D1E14  AES
Length 1: payload length
Pointer 1: payload
Length 2: ciphertext length
Pointer 2: ciphertext = (P XOR CIPH(Ctr))
```

5. Save MAC.

```
Desc # 5
Header: 0x91820000
Length 1: zero (not saving Ctr)
Pointer 1: null (not saving Ctr)
Length 2: MAC length (Tlen), from passed in parameter
Pointer 2: MAC (last Y value of interest), put into passed in parameter
```

6. Reestablish CTR mode (`Ctr` mode, Ctr_0, & key)

   This and the next descriptor encrypt MAC.

```
Desc # 1
Header encrypt:
0x11881E1C  AES
Length 1: Ctr_0 length (m) from ACCO
Pointer 1: Ctr_0 (to calculate S_0 = CIPH_k(Ctr0)), found in ACCO
Length 2: zero (not needed, key already established)
Pointer 2: null (not needed, key already established)
```

7. Compute output.

```
Desc # 4
Header: 0x11850000
Length 1: MAC length (Tlen)
Pointer 1: MAC from parameter passed in
Length 2: Encrypted MAC length (Tlen) that was passed in
Pointer 2: Encrypted MAC = (T XOR S_0), put into buffer that was passed in
```

26.2.3.6 CCM Decrypt

Note that the block size is 128 bits (16 bytes).

For CCM Decrypt, follow these steps:

1. Set context (CTR mode, `Ctr` modulus, Ctr_0, key)

```
Desc # 1
Header:0x11881E1C  AES
Length 1: Ctr_0 length is found in ACCO
Pointer 1: Ctr_0 is found in ACCO
Length 2: Key length is found in SKO
Pointer 2: Key is found in SKO
```

2. Derive decrypted MAC and increment `Ctr`

```
Desc # 4
Header: 0x11850000
Length 1: Encrypted MAC length that was passed in
Pointer 1: Encrypted MAC that was passed in
Length 2: MAC length that was passed in
Pointer 2: MAC put into buffer passed in
```

3. Calculate Y (CBC mode)

   The key was set in step 1 and the IV is zero due to chip reset or previous descriptor chain completion. Notice also that the LLO bit is cleared to allow links.

```
Desc # 2
Header:0x108D000C  AES
Length 1: B length ((u+1) * 16 bytes)
Pointer 1: link to B_0
Length 2: Y length (same as Length 1)
Pointer 2: Y

Link to B_0

Length 1: B_0 length (16 bytes)
Pointer 1: B_0 passed in
Next 1: Link to B_associated_data

Link to B_associated_data

Length 2: B_ad length passed in (u * 16 bytes)
Pointer 2: B_ad passed in
Next 2: Null
```

4. Decrypt payload (CCM mode)

```
Desc # 2
Header:0x118D1E10  AES
Length 1:  ciphertext length passed in
Pointer 1:  cipher from buffer passed in
Length 2:  plaintext length passed in
Pointer 2:  plaintext put in buffer passed in
```

5. Save context (MAC)

   This MAC should be compared to the MAC derived in section step 2. If they are not equal, return an INVALID status and zero out the plain text buffer.

```
Desc # 5
Header: 0x91820000
Length 1: zero
Pointer 1: null
Length 2: MAC length passed in
Pointer 2: MAC put in some scratchpad area
```

26.2.3.7 ARC4

Several combinations exist for ARC4. These are the descriptor chains for the possible symmetric cipher mode permutations.

Table 26-2. ARC4 Descriptor Chains

INIT	SAVE	LOAD	Descriptors
0	0	0	Return error ‘symmetric cipher mode error’ (nothing to do)
0	0	1	SBox In (Desc # 33) -> Data In/Out (Desc # 4)
0	1	0	Return error ‘symmetric cipher mode error’ (nothing to work with)
0	1	1	SBox In (Desc # 33) -> Data In/Out (Desc # 4) -> SBox Out (Desc # 34)

Table 26-2. ARC4 Descriptor Chains (Continued)

1	0	0	Key In (Desc # 35) -> Data In/Out (Desc # 4)
1	0	1	Return error ‘symmetric cipher mode error’ (not compatible)
1	1	0	Key In (Desc # 35) -> Data In/Out (Desc # 4) -> SBox Out (Desc # 34)
1	1	1	Return error ‘symmetric cipher mode error’ (not compatible)

NOTE

The following descriptors are not organized as part of a descriptor chain but are ordered by “Desc #” only. See Table 26-2 for how to order descriptor chains.

For ARC4, follow these steps:

1. Encrypt/decrypt:

   Context is loaded prior to this descriptor. If the passed in length (ciphertext or plaintext, whichever is appropriate), this descriptor is not built (not added to the chain).

   ```
   Desc # 4
   Header: 0x11850000
   Length 1: ciphertext/plaintext length value that was passed in
   Pointer 1: points to the ciphertext/plaintext buffer that was passed in
   Length 2: plaintext/ciphertext length value is same as plaintext/ciphertext length
   Pointer 2: points to the plaintext/ciphertext buffer that was passed in
   ```

2. Set the context (SBox context & SBox):

   ```
   Desc # 33
   Header encrypt/decrypt: 0x11890027/0x91890023
   Length 1: SBox context length obtained from SCCO (24 bit)
   Pointer 1: SBox context found in SCCO
   Length 2: SBox length
   Pointer 2: SBox found in SCCO
   ```

3. Save the context (SBox context & SBox)

   ```
   Desc # 34
   Header: 0x11860000
   Length 1: SBox context length obtained from SCCO (24 bit)
   Pointer 1: SBox context found in SCCO
   Length 2: SBox length found in SCCO
   Pointer 2: points to the SBox found in the SCCO
   ```

4. Set the key

   ```
   Desc # 35
   Header: 0x11830003
   Length 1: zero
   Pointer 1: null
   Length 2: key length from SKO (8 to 128 bit)
   Pointer 2: key found in SKO
   ```

26.2.4 Hash Descriptors

The following example shows the descriptors needed to perform the hashing operations required of the Sahara drivers.

26.2.4.1 Overview

All headers are defined in the SAHARA chapter of the *MCIMX27 Applications Processors Reference Manual*. For this section, the "Descriptor Header Format" figure, located in the SAHARA chapter of the *MCIMX27 Applications Processors Reference Manual*, assumes the following settings:

- All shaded ("Unimplemented or Reserved") bits are set to zero (the hardware will flag it as an error if not)
- The CHA is MDHA
- The LLO bit is set
- All Form values are available in the "Descriptor Header Encoding" table
- The Parity bit is used to force the header to odd parity

The following descriptors are ordered by "`Desc` #" only. See Table 26-3 for how to order descriptor chains.

26.2.4.2 Details

Several combinations exist for Hashing. These are the descriptor chains for the possible Hash permutations.

Table 26-3. Hash Descriptor Chains

Case	INIT	SAVE	LOAD	Final	Descriptors
a	0	1	1	0	Context In (Desc # 6) -> Data In/Context Out (Desc # 10)
b	1	1	0	0	Data In/Context Out (Desc # 8)
c	0	0	1	1	Context In (Desc # 6) -> Data In/Digest Out (Desc # 10)
d	0	1	1	1	Context In (Desc # 6) -> Data In/Digest Out (Desc # 10) -> Context Out (Desc # 11)
e	1	0	0	1	Data In/Digest Out (Desc # 8)
f	1	1	0	1	Data In/Digest Out (Desc # 8) -> Context Out (Desc # 11)
g	ALL OTHERS				Return error 'hash mode error'

Follow these steps:

1. Set context (set mode field & initial message digest)

   Settings per case

   — a: `INIT` bit clear, `PDATA` bit clear

   — c: `INIT` bit clear, `PDATA` bit set

   — d: `INIT` bit clear, `PDATA` bit set

```
Desc # 6
Header:
0xA1880021 MD5 context length is 24 bytes, digest (16) + context information
0x21880020 SHA1 context length is 24 bytes, digest (20) + context information
0x21880023 SHA224 context length is 36 bytes, digest (32) + context information
0xA1880022 SHA256 context length is 36 bytes, digest (32) + context information
Length 1: context length
Pointer 1: pointer to context found in SCCO
```

```
Length 2: zero (key not used)
Pointer 2: null (key not used)
```

2. Perform Hash (set mode field and perform digest)

   Settings per case

   — b: INIT bit set, PDATA bit clear

   — e: INIT bit set, PDATA bit set

   — f: INIT bit set, PDATA bit set

```
Desc # 8
Header:
0x218D0025 MD5 digest length up to 16 bytes
0xA18D0024 SHA1 digest length up to 20 bytes
0xA18D0027 SHA224 digest length up to 28 bytes
0x218D0026 SHA256 digest length up to 32 bytes
Length 1: message length that was passed in (multiple of 64 bytes if pad bit not set)
Pointer 1: pointer to message
Length 2: digest length
Pointer 2: puts digest into buffer passed in or context into HCO context field
```

3. Perform hash.

   This uses whatever context has been established. Note that on the final segment or ‘chunk’ of the calculation, the pad bit should be set.

```
Desc # 10
Header: 0x21850000
Length 1: message length that was passed in (multiple of 64 bytes if pad bit not set)
Pointer 1: pointer to message passed in
Length 2: digest (context) length
Pointer 2: puts digest (context) into buffer passed in
```

4. Save Digest.

```
Desc # 11
Header: 0xA1820000
Length 1: zero
Pointer 1: null
Length 2: context length
Pointer 2: puts context into HCO context field
```

26.2.5 HMAC Descriptors

The following section shows the descriptors needed to perform Keyed-Hashing for Message Authentication (HMAC) operations.

26.2.5.1 Overview

All headers are defined in the SAHARA chapter of the *MCIMX27 and MCIMX27L Applications Processors Reference Manual*. For this section, the Descriptor Header format assumes the following settings:

- All shaded (“Unimplemented or Reserved”) bits are set to zero (the hardware will flag it as an error if not)
- The CHA is MDHA
- The LLO bit is set

- All Form values are available in the “Descriptor Header Encoding”
- The Parity bit is used to force the header to odd parity

The following terminology is used within this section. IPad is a constant. Performing an exclusive-OR between IPad and a key produces an intermediate result. The hash of the intermediate result is referred to as the Inner Precompute in this appendix. The same follows for the constant OPad, with the resulting value referred to as the Outer Precompute.

The following descriptors are ordered by “Desc #”. See Table 26-4 for how to order descriptor chains. Note that the key used in these descriptors has a maximum length of 64 bytes.

26.2.5.2 Details

These are the supported descriptor chains for HMAC.

Table 26-4. HMAC Descriptor Chains

	API Flags					Content Descriptor		Header Bits			
Case	INIT	SAVE	LOAD	FINAL	KEY	Content (Descriptors #)		HMAC	PDATA	MAC_FULL	INIT
a	0	0	1	1	x	Context In (6) -> Data In/Digest Out (10		1	1	0	1
b	0	1	1	0	x	Context In (6) -> Data In/Context Out (10)		0	0	0	1
c	0	1	1	1	0	Context In (6) -> Data In/Digest Out (10) -> Context Out (11)		1	1	0	1
d	1	0	0	1	0	Precomps In (6) -> Data In/Digest Out (10)		1	1	0	1
e	1	0	0	1	1	Key In (6) -> Data In/Digest Out (10)		1	1	1	1
f	1	1	0	0	0	Precomps In (6) -> Data In/Context Out (10)		0	0	0	1
g	1	1	0	1	0	Precomps In (6) -> Data In/Digest Out (10) -> Context Out (11)		1	1	0	1
h	1	1	0	1	1	Key In (6) -> Data In/Digest Out (10) -> Context Out (11)		1	1	1	1
i	return ‘incompatible hmac flag’ error if (/* nothing to work with */ /* cannot do both */ ((init==0)&& (load==0)) ||((init == 1)&& (load == 1)) || /* need to have some o/p */ /* sahara cant handle this */ ((save==0)&&(final == 0)) || ((init==1)&&(save == 1) && (final==0)&&(key == 1)))							NA			
j	Flags have no effect on precompute call					Key In / Inner Precompute out (8) -> Key In / Outer Precompute out (8)	IPad = 1 OPad = 1	0	0	0	1

Follow these steps:

1. Set Context

If Final = 0, data must be a multiple of 64 bytes. If Final = 1, Padding is performed (PDATA set), so length does not have to be a multiple of 64 bytes.

The following is an example of case f.

```
Desc # 6:
Header (precompute):
0xA1880229 MD5
0x21880228 SHA1
0xA188002B SHA224
0x2188002A SHA256
Length 1: Inner Precompute length (32 bytes)
Pointer 1: pointer to Inner Precompute in HCO
Length 2: zero
Pointer 2: Null
```

2. Calculate Precompute

   When calculating the precomputes

   — Either IPAD or OPAD bit is set

   — INIT bit is set

   For the headers shown, IPAD is set (thus OPAD is clear).

NOTE

If one precompute follows another without changing the key, that the key can be specified as length zero and null in the second precompute (as the key will still be in Sahara's internal key register). Also, as the key comes through the FIFO, it can be any length.

```
Desc # 8
Header:
0x218D0061 MD5 24 byte precompute (context) length
0xA18D0060 SHA1 24 byte precompute (context) length
0xA18D0063 SHA224 36 byte precompute (context) length
0x218D0062 SHA256 36 byte precompute (context) length
Length 1: key length obtained from SKO
Pointer 1: key is found in the SKO
Length 2: precompute length (32 bytes)
Pointer 2: puts precompute into HCO
```

3. Hash from current context

   This uses whatever context is currently set.

```
Desc # 10
Header: 0x21850000
Length 1: message length
Pointer 1: pointer to message buffer
Length 2: message digest length
Pointer 2: pointer to message digest buffer or HCO's context area
```

4. Save context

```
Desc # 11
Header: 0xA1820000
Length 1: zero
Pointer 1: null
Length 2: context out length
Pointer 2: pointer to context out found in HCO
```

26.2.6 Random Number Descriptors

The following exemplifies the descriptors needed to perform random number operations required of the Sahara drivers.

26.2.6.1 Overview

All headers are defined in the SAHARA chapter of the *MCIMX27 Applications Processors Reference Manual*. For this appendix, the "Descriptor Header Format" figure, found in the SAHARA chapter of the *MCIMX27 Applications Processors Reference Manual*, assumes the following settings:

- All shaded ("Unimplemented or Reserved") bits are set to zero (the hardware will flag it as an error if not)
- The CHA is RNG
- The LLO bit is set
- All Form values are found in the "Descriptor Header Encoding"
- The Parity bit is used to force the header to odd parity

26.2.6.2 Details

For random number descriptors, follow these steps:

1. Get random numbers

```
Desc # 18
Header: 0xB18C0000
Length 1: number of bytes of random values wanted
Pointer 1: pointer to random value buffer
Length 2: zero (entropy not used)
Pointer 2: null (entropy not used)
```

2. Seed the RNG

```
Desc # 18
Header: 0x318C0001
Length 1: zero (don't get random numbers)
Pointer 1: null (don't get random numbers)
Length 2: zero (entropy not used)
Pointer 2: null (entropy not used)
```

26.3 Requirements

The SAHARA2 driver meets the following requirements:

- The driver supports hashing with MD5, SHA-1, SHA-224 and SHA-256 algorithms.
- The driver supports HMAC with the same algorithms as for hashing.
- The driver provides Symmetric cryptography support for AES, DES, and triple DES, in ECB, CBC, and CTR modes (though only AES is supported in CTR mode). ARC4 support is also provided.
- CCM is supported for AES.
- The driver includes support for the wrapped keys (that is, hiding keys in the SCC), using the SCC key to encrypt or decrypt, HMAC functions, or CCM.

- The Sahara2 driver allows the generation of an arbitrary number of bytes of random data.
- The FSL SHW API is provided in both user mode and kernel mode. Callbacks and non-callback non-blocking support are provided.

26.4 Source Code Structure

Table 26-5 lists the source files associated with the SAHARA2 driver that are available in the directory `<ltib_dir>/rpm/BUILD/linux-/drivers/mxc/security/sahara2`.

Table 26-5. Sahara2 Source File List

File	Description
`sah_driver_interface.c`	MXC Sahara2 low level driver
`sah_hardware_interface.c`	Provides an interface to the SAHARA hardware registers.
`sah_interrupt_handler.c`	MXC Sahara2 Interrupt Handler.
`sah_memory_mapper.c`	Re-creates SAHARA Data structures in kernel memory such that they are suitable for DMA.
`sah_queue.c`	Provides FIFO Queue implementation.
`sah_queue_manager.c`	This file provides a Queue Manager implementation.The Queue Manager manages additions and removal from the queue and updates the status of queue entries. It also calls `sah_HW_*` functions to interact with the hardware.
`sah_status_manager.c`	Contains functions which processes the SAHARA status registers.
`sf_util.c`	Security Functions component API - Utility functions
`fsl_shw_auth.c`	Contains the routines which do the combined encryption and authentication.
`fsl_shw_hash.c`	Implements Cryptographic Hashing functions of the API.
`fsl_shw_hmac.c`	Provides HMAC functions of the API.
`fsl_shw_rand.c`	Generates random numbers.
`fsl_shw_sym.c`	Provides Symmetric-Key encryption support for block cipher algorithms.
`fsl_shw_user.c`	Implements user and platform capabilities functions
`fsl_shw_wrap.c`	Implements Key-Wrap (Black Key) functions
`km_adaptor.c`	Adaptor provides interface to driver for kernel user.

Table 26-6 lists the header files associated with the SAHARA driver are found in the directory `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/mxc/security/sahara2/include`.

Table 26-6. Sahara2 Header File List

File	Description
`fsl_shw.h`	MXC Sahara2 Definition of the Freescale Security Hardware API.
`fsl_platform.h`	File to isolate code which might be platform-dependent
`sahara.h`	All of the defines used throughout user and kernel space
`sah_driver_common.h`	Provides types and defined values for use in the Driver Interface.
`sah_hardware_interface.h`	Provides an interface to the SAHARA hardware registers.
`sah_interrupt_handler.h`	Provides a hardware interrupt handling mechanism for device driver.
`sah_kernel.h`	Provides definitions for items that user-space and kernel-space share.
`sah_memory_mapper.h`	Re-creates SAHARA Data structures in kernel memory such that they are suitable for DMA.
`sah_queue_manager.h`	This file provides a Queue Manager implementation.The Queue Manager manages additions and removal from the queue and updates the status of queue entries. It also calls `sah_HW_*` functions to interact with the hardware.
`sah_status_manager.h`	SAHARA Status Manager Types and Function Prototypes
`sf_util.h`	Security Function Utility Functions
`diagnostic.h`	Macros for outputting kernel and user space diagnostics.
`adaptor.h`	The Adaptor component provides an interface to the device driver.

26.5 Configuration

NOTE: This section does not apply for the i.MX31 3-Stack Board.

The following Linux kernel configurations are provided for this module:

1. CONFIG_MXC_SAHARA - This configuration option for Sahara Hardware support. In `menuconfig` this option is found under Device Drivers -> MXC support drivers -> MXC Security Drivers -> SAHARA2 Security Hardware Support -> Security Hardware Support (FSL SHW). By default, this option is Y.
2. CONFIG_MXC_SAHARA_USER_MODE - The driver can be configured to provide an interface to user space (used by the library). (This configuration switch is currently ignored, and the user space interface is currently always provided.). In `menuconfig` this option is found under Device Drivers -> MXC support drivers -> MXC Security Drivers -> SAHARA2 Security Hardware Support -> User Mode API for FSL SHW. By default, this option is Y.

3. CONFIG_MXC_SAHARA_POLL_MODE - The driver can be configured to poll the Sahara2 hardware device for end-of-operation status, or it can (by default) process an interrupt for end-of-operation. In the `menuconfig` this option is found under Device Drivers -> MXC support drivers -> MXC Security Drivers -> SAHARA2 Security Hardware Support -> Security Hardware Support. By default, this option is N.
4. CONFIG_MXC_SAHARA_POLL_MODE_TIMEOUT - To avoid infinite polling, a time-out is provided. Should the time-out be reached, a fault is reported, indicating something must have gone wrong with SAHARA. This causes SAHARA to be reset. This time-out period is configurable. When polling mode is selected, the value for CONFIG_MXC_SAHARA_POLL_MODE_TIMEOUT appears and can be modified. Polling mode works nearly the same as interrupt mode, that is, blocking mode returns the result of the descriptor chain (succeeded, erred, and so on); non-blocking mode queues results in a results pool and `fsl_shw_get_results()` retrieves them; callback mode (non-blocking mode only) the callback is made just BEFORE control is returned from the API call (in interrupt mode it is some time after). In the `menuconfig` this option is found under Device Drivers -> MXC support drivers -> MXC Security Drivers -> SAHARA2 Security Hardware Support -> Force driver to POLL for hardware completion.

26.6 Programming Interface

This driver implements all the methods that will be required by the Linux serial API to interface with the MXC SAHARA2 Driver. It implements and provides a set of control methods to the core SAHARA2 driver present in Linux. Refer to the API document (included doxygen document) for more information on the methods implemented in the driver.

26.7 Interrupt Requirements

There is no interrupt requirement in this module.

Chapter 27
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, bi-directional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices.

27.1 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 is not supported by this driver.

27.2 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.

27.3 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.

27.4 Software Operation

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

27.4.1 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.

27.4.2 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.

27.5 Requirements

The MXC I^2C driver meets the following requirements:

- The driver supports the I^2C communication protocol.
- The driver supports the I^2C master mode of operation.
- The driver does not support the I^2C slave mode of operation.

27.6 Source Code Structure

Table 27-1 lists the I^{2}C bus driver source files available in the directory, `<ltib_dir>/rpm/BUILD/linux-/drivers/i2c/busses`.

Table 27-1. I^{2}C Bus Driver Files

File	Description
`mxc_i2c.c`	I^{2}C bus driver source file

27.7 Configuration

27.7.1 Linux Menu Configuration Options

In order to get to the I^{2}C 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`.

27.8 Programming Interface

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

27.9 Interrupt Requirements

The I^{2}C module generates many kinds of interrupts. The highest interrupt rate is associated with the transfer complete interrupt.

Table 27-2. I^{2}C 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^{2}C interface).

27.10 Device-Specific Information

The x in I^{2}Cx denotes the individual I^{2}C number.

Table 27-3. Default Configuration

Option	I2C_NR	I2C1_FRQ_DIV	I2C2_FRQ_DIV	I2C3_FRQ_DIV
i.MX27	1	0x17	N/A	N/A

Chapter 28
I2C Slave Driver

I^2C is a two-wire, bi-directional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices.

The MXC I^2C slave driver is divided into two layers: the I^2C slave core and the I^2C slave chip driver. The I^2C core driver stays at the top level and is the interface for the registered I^2C slave device to the Linux device driver model. The I^2C slave chip driver handles the low level hardware operation.

28.1 I^2C Slave Core Overview

The I^2C slave core is the interface to the Linux driver model. It receives the I^2C slave chip driver's register requests, and creates device files for the registered I^2C slave chip. It supports the open, read, write and ioctl requests to the chip. It also provides a ring buffer API to the I^2C slave chip driver.

28.2 I^2C Slave Chip Driver Overview

The I^2C slave chip driver is responsible for the hardware operation, It initializes the I^2C slave hardware and registers itself to the I^2C slave core. Then it waits for the I^2C slave core to open them. After being opened, it sends out the data that I^2C slave core asked for, or receives I^2C data that send to it according to the I^2C address and delivers the received data to the I^2C core.

28.3 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.

For more details, see the IC specification.

28.4 Software Operation

The I^2C slave driver provides standard `open`, `read`, `write` and `ioctl` operations for the user space application. when the I^2C slave module is loaded, there are some device files like `/dev/slave-i2c-X` created for the registered I^2C slave chip, while X stands for the registered I^2C slave index. The user space application uses `open` to initialize and start the specific i2c slave device, and uses `read` to get the received data and uses `write` to send I^2C data.

28.5 Requirements

The MXC I^2C driver meets the following requirements:

- The driver supports the I^2C communication protocol.
- The driver supports the I^2C slave mode of operation.

28.6 Source Code Structure

Table 28-1 lists the I^2C bus driver source files available in the directory, `<ltib_dir>/rpm/BUILD/linux-/drivers/i2c-slave`.

Table 28-1. I^2C Bus Driver Files

File	Description
`i2c_slave_core.c`	i2c slave core file
`i2c_slave_device.c`	interface between i2c slave core and chip driver
`i2c_slave_ring_buffer.c`	ring buffer source file
`mxc_i2c_slave.c`	i2c slave chip driver for mx35 i2c slave
`i2c_slave_client.c`	I^2C master client drive which makes the i2c master on the some board can get access to our i2c slave.

28.7 Configuration

28.7.1 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 following Linux kernel configurations are provided for this module:

- CONFIG_I2C_SLAVE - This option is available under `Device Drivers -> I2C Slave support.`
- CONFIG_MXC_I2C_SLAVE - This option is available under `Device Drivers -> I2C Slave support.`
- CONFIG_I2C_SLAVE_CLIENT - This option is available under `Device Drivers -> I2C Slave support.`

28.8 Programming Interface

The I^2C device driver could use the standard `open`, `read`, `write` and `ioctl` to operate the I^2C slave device.

Chapter 29
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-driven transmit/receive of bytes
- Supports multiple master controller interface
- Supports the multiple slaves select
- Supports multi-client requests

29.1 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

29.2 Software Operation

29.2.1 SPI Sub-System in Linux

The CSPI driver layer is located between the client layer (PMIC is one of the clients) and the MXC hardware access layer. Figure 29-1 shows the block diagram for SPI subsystem in Linux.

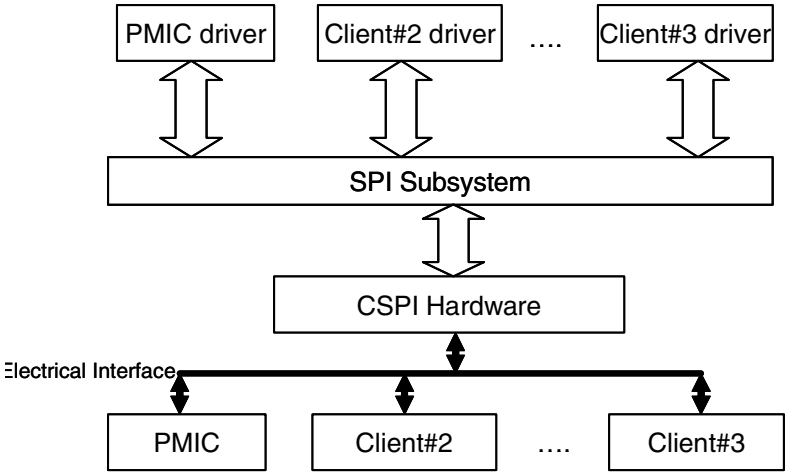


Figure 29-1. SPI Sub-system

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.

Each SPI client 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 29-2 shows how the different SPI drivers are layered in the SPI subsystem.

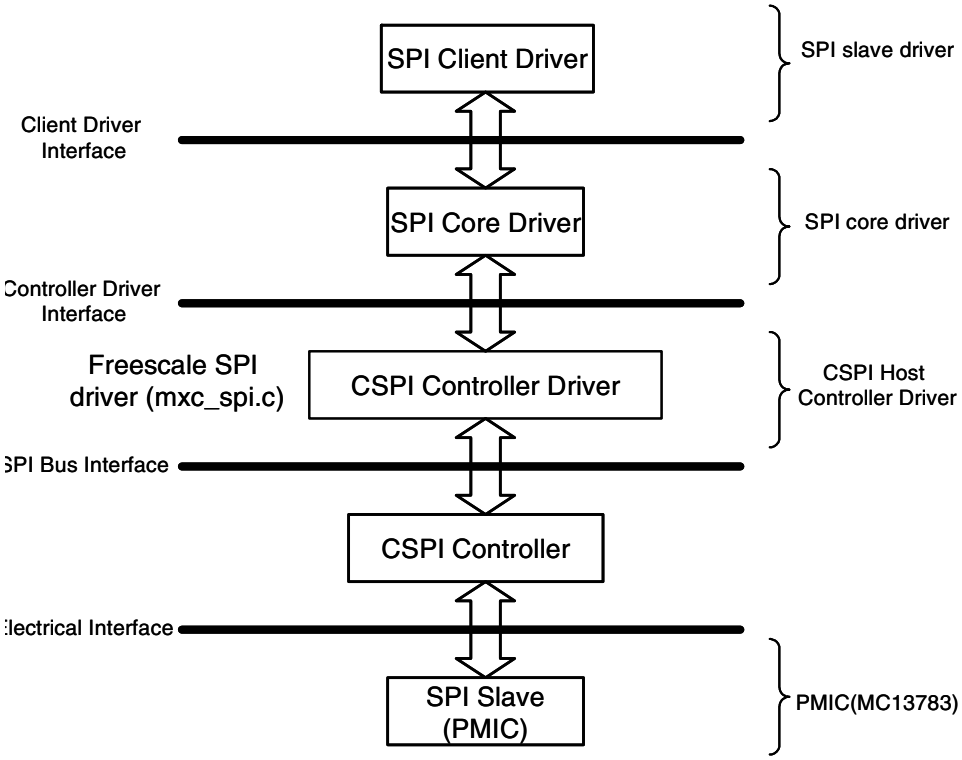


Figure 29-2. Layering of SPI Drivers in SPI subsystem

29.2.2 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 user space interface with the help of device node entry but supports a `sysfs` interface.

29.2.3 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( )`—Initialize the current SPI device.
6. SPI driver ISR `mxc_spi_isr( )`—Called when the data transfer operation is completed and an interrupt is generated.

29.2.4 CSPI Synchronous Operation

Figure 29-3 shows how CSPI provides synchronous read/write operations.

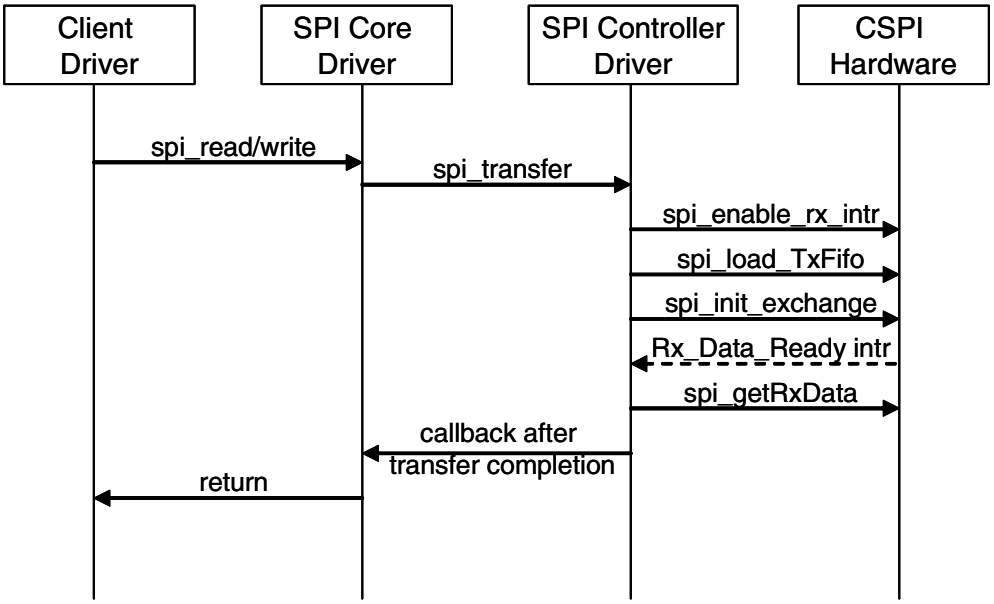


Figure 29-3. CSPI Synchronous Operation

29.2.5 PMIC Access

Figure 29-4 shows the how PMIC can be accessed through the SPI subsystem.

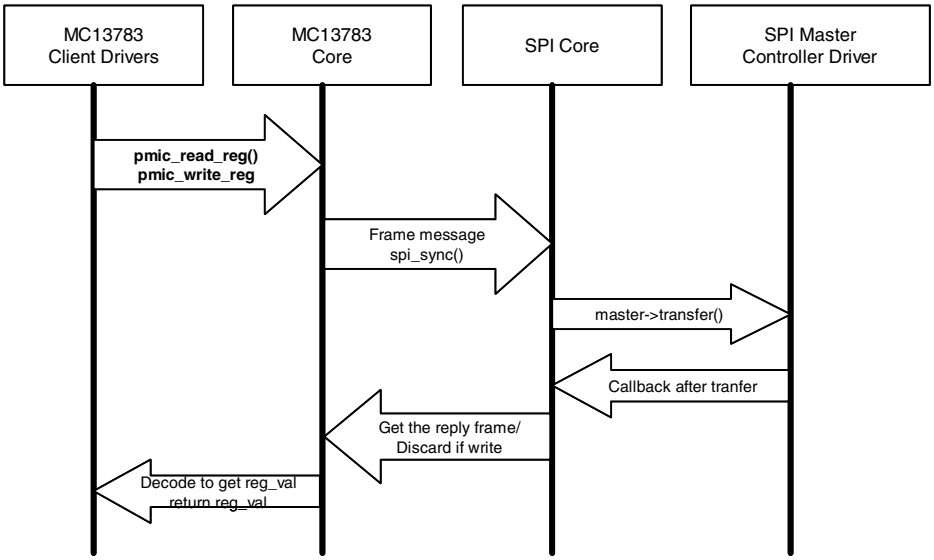


Figure 29-4. PMIC Access through SPI

29.3 Requirements

The CSPI module implements the following requirements:

- It implements each of the functions required by a CSPI module to interface to Linux.
- It provides support for multiple SPI master controllers.
- It provides support to handle multi-client synchronous requests.

29.4 Source Code Structure

Table 29-1 lists the source files available in the devices directory:

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

Table 29-1. CSPI Source File List

File	Description
`mxc_spi.h`	Header file defining registers.
`mxc_spi.c`	Freescale SPI Master Controller driver

29.5 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->MXC CSPI Controller as SPI Master.
- 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.

29.6 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 the doxygen.

29.7 Interrupt Requirements

The SPI interface generates interrupts. CSPI interrupt requirements are listed in Table 29-2.

Table 29-2. 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.

29.8 Device-Specific Information

Table 29-3 lists the number of CSPI controllers in different i.MX family platforms.

Table 29-3. CSPI Controllers in i.MX Family Platforms

Platforms (SOC)	No. of CSPI Controllers
i.MX27	3

Chapter 30
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.

The MMC driver has 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

30.1 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 support 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/from the cards. The SD memory card system defines two alternative communication protocols: SD and SPI. The SDHC or eSDHC only support the SD bus protocol.

For SDHC module, the SDHC command number and 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 DMA request when the FIFO is full.
2. Upon receiving this request, DMA 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 DMA request whenever the transmit FIFO is empty.
2. Upon receiving this request, the DMA 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, they 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.

30.2 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 30-1 shows how the MMC-related drivers are layered.

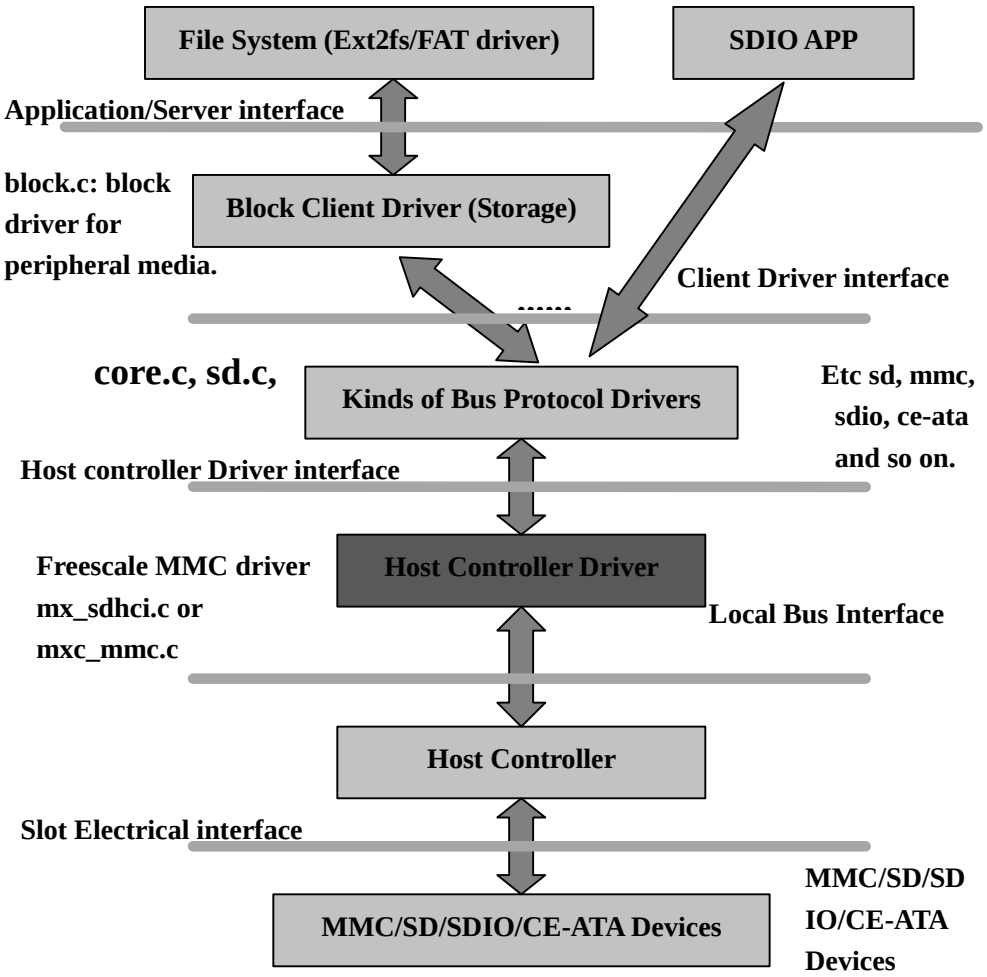


Figure 30-1. Layering of MMC drivers

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 DMA 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.

30.3 Requirements

- The MMC driver provides support for multiple SDHC modules.
- The MMC driver provides all the entry points to interface with the Linux MMC core driver.
- The MMC driver supports MMC and SDcards.
- The MMC driver recognizes data transfer errors like command time outs and CRC errors.
- The MMC driver supports power management.
- The MMC driver conforms to the Linux coding standards.

30.4 Source Code Structure

Table 30-1 lists the SDHC source files available in the source directory `<ltib_dir>/rpm/BUILD/linux-/drivers/mmc/host/`.

Table 30-1. SDHC Driver File List

File	Description
`mxc_mmc.h`	Header file defining registers
`mxc_mmc.c`	SDHC driver

30.5 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 applied for i.MX27 platforms.

30.6 Programming Interface

This driver implements the functions required by the MMC bus protocol to interface with the i.MX SDHC and eSDHC module. For additional information, see the *BSP API Document.*

Chapter 31
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 DMA-driven transmit/receive of characters.
- Supports standard Linux baud rates up to 1.5Mbps.
- 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. Supports only 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.

31.1 UART Driver Hardware Operation

Refer to the IC/Hardware Specification to determine the number of UART modules available in your 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.

They also support a variety of maskable interrupts when the data level in each FIFO reaches a programmed threshold level and when there is a change in state in the modem signals. Each UART can be programmed to be in DCE or DTE mode.

31.2 UART Driver Software Operation

The Linux OS contains a core UART driver that handles a lot of the serial operations that are common across UART drivers for various platforms. The low-level UART driver is responsible for supplying to this core UART driver such information as the UART port information and a set of control functions. 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 2 DMA channels for the UARTs that need DMA transfer. On a receive transaction, the driver copies the data from the DMA receive buffer to the TTY Flip Buffer.

While using DMA to transmit, the driver copies the data from the UART transmit buffer to the DMA transmit buffer and sends this buffer to the DMA system. The user should use hardware-driven hardware flow control when using DMA data transfer. For more information, see the Linux documentation on the serial driver in the kernel source tree.

The low-level driver supports both interrupt-driven software-controlled hardware flow control and hardware-driven hardware flow control. The hardware flow control method can be configured using the options provided in the header file. The user has the capability to de-assert the CTS line using the available IOCTL calls. If the user wishes to assert the CTS line then control is transferred back to the receiver, as long as the driver has been configured to use hardware-driven hardware flow control.

31.3 UART Driver Requirements

The UART driver meets the following requirements:

- Supports baud rates up to 1.5Mbps.
- Recognizes frame and parity errors only in interrupt-driven mode. The UART driver does not recognize these errors in DMA-driven mode.
- Sends and 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.

31.4 UART Driver Source Code Structure

Table 31-1 lists the source files associated with the UART driver that are available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/serial`.

Table 31-1. 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

Table 31-2 lists the header files associated with the UART driver that are available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.22/include/asm-arm/arch-mxc`.

Table 31-2. UART Global Header File List

File	Description
`mxc_uart.h`	UART header that contains UART configuration data structure definitions
`board-xxxx.h`	Holds some UART board specific configuration options

The source file, `serial.c/serial.h`, is associated with the UART driver that are available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.22/arch/arm/mach-xxxx`. The source file contains UART configuration data and calls to register the device with the platform bus.

31.5 UART Driver Configuration

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

31.5.1 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 Characters->Serial. 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 `menuconfig` this option is available under Characters->Serial. By default, this option is N for all architectures.

31.5.2 Source Code Configuration Options

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

31.5.2.1 Chip Configuration Options

The following chip-specific configuration options are provided in `mxc_uart.h`:

- Number of UART Ports (`UART_NR`)—Number of UART ports in the platform.
- UART Interrupts Muxed (`UARTx_MUX_INTS`)—Specifies which set of interrupts is integrated with the ARM core, the muxed ANDed interrupt lines or the individual interrupts from the UART port.
- UART TX Interrupt / UART Muxed Interrupt Number (`UARTx_INT1`)—Specifies the transmit interrupt number, or the interrupt number of the ANDed interrupt in case the interrupts are muxed.
- UART RX interrupt (`UARTx_INT2`)—Specifies the receiver interrupt number. In case the interrupts are muxed, set this option to a value of -1.
- UART MINT Interrupt (`UARTx_INT3`)—Specifies the master interrupt number. If the interrupts are muxed, set this option to a value of -1.
- UART HW Flow control (`UARTx_HW_FLOW`)–Allows the user to choose either an interrupt-driven software-controlled hardware flow, or a hardware-driven hardware-controlled flow. When DMA is enabled, this option should always be specified.
- UART CTS Threshold (`UARTx_UCR4_CTSTL`–Specifies the threshold at which the CTS pin is de-asserted by the receiver.
- UART DMA Enable/Disable (`UARTx_DMA_ENABLE`)—Specifies whether to use DMA-driven data transfer. Setting this option to 1 enables DMA-driven data transfer.
- UART DMA RX Buffer size (`UARTx_DMA_RXBUFSIZE`)—Specifies the size of the DMA receive buffer. The minimum size is 512 and the size should be a multiple of 256.
- UART RX Threshold (`UARTx_UFCR_RXTL`)—Specifies the trigger level of the UART receive FIFO. This option controls the threshold at which a receive interrupt is generated. Set this option to any value between 0 and 32.
- UART TX Threshold (`UARTx_UFCR_TXTL`)—Specifies the trigger level of the UART transmit FIFO. This option controls the threshold at which a transmit interrupt is generated. Set this option to any value between 0 and 32
- UART Shared Peripheral (`UARTx_SHARED_PERI`)—Specifies whether the UART is a shared peripheral. The value should be set to the UART shared peripheral number, or to -1 if the UART is not a shared peripheral.

The x in `UARTx` denotes the individual UART number. The default configuration for each individual UART number is listed in Table 31-5.

31.5.2.2 Board Configuration Options

The following board-specific configuration options for the driver can be set within `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 will not be registered in the file system and the user will not be able to access it.
- MAX_UART_BAUDRATE—Specify the maximum baud rate that you wish to support on your 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 31-5.

31.6 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 (included doxygen document).

31.7 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 31-3.

Table 31-3. UART Interrupt Requirements

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

31.8 Device-Specific Information

31.8.1 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 31-4.

31.8.2 Board Setup Configuration

For **i.MX27ADS,** either UART1 or UART 2 can be enabled through DIP switches; the same applies for UART3/UART4 and UART5/UART6. UART5 and UART6 are not functional when the CMOS sensor is plugged in. UART4 is default disabled, for its pins are shared with USB.

Table 31-4. UART General Configuration

Platform	UART_NR	MAX BAUDRATE
i.MX27ADS	6	1500000

Table 31-5. UART Active/Inactive Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	1	1	1	0	1	1

Table 31-6. UART IRDA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	NO_IRDA	NO_IRDA	IRDA	NO_IRDA	NO_IRDA	NO_IRDA

Table 31-7. UART Mode Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	MODE_DCE	MODE_DCE	MODE_DCE	MODE_DTE	MODE_DTE	MODE_DTE

Table 31-8. UART Shared Peripheral Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	-1	-1	-1	-1	-1	-1

Table 31-9. UART Hardware Flow Control Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	1	1	1	1	1	1

Table 31-10. UART DMA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	0	0	0	0	0	0

Table 31-11. UART DMA RX Buffer Size Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	0	0	0	0	0	0

Table 31-12. UART UCR4_CTSTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	16	16	16	16	16	16

Table 31-13. UART UFCR_RXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	16	16	16	16	16	16

Table 31-14. UART UFCR_TXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	16	16	16	16	16	16

Table 31-15. UART Interrupt Mux Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	INTS_MUXED	INTS_MUXED	INTS_MUXED	INTS_MUXED	INTS_MUXED	INTS_MUXED

Table 31-16. UART Interrupt 1 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	INT_UART1	INT_UART2	INT_UART3	INT_UART4	INT_UART5	INT_UART6

Table 31-17. UART Interrupt 2 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	-1	-1	-1	-1	-1	-1

Table 31-18. UART interrupt 3 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX27	-1	-1	-1	-1	-1	-1

31.9 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 32
16C652 Universal Asynchronous Receiver/Transmitter (UART) Driver

The external SC16C652/ST16C2552 universal asynchronous receiver/transmitter (UART) runs on Freescale i.MX boards. The serial driver files for the Linux 2.6 kernel have been modified.

The following are the features of the UART for the i.MX family of processors:

- Interrupt-driven transmit/receive of characters
- Supports standard Linux baud rates from 115K baud down to 50 baud
- Supports two UART ports on 16C652 and four UART ports on 15C6552
- Supports transmitting and receiving characters with 7-bit and 8-bit character lengths
- Supports transmitting 1, 1.5, or 2 stop bits
- Supports odd and even parity
- Supports XON/XOFF software flow control
- Supports CTS/RTS hardware flow control
- Send and receive break characters through the standard Linux serial API
- Recognizes frame and parity errors
- Ability to ignore characters with break, parity and frame errors
- Get and set UART port information through the TIOCGSSERIAL and TIOCSSERIAL TTY ioctls
- Supports the standard TTY layer ioctl calls
- Includes console support that is needed to bring up the command prompt through one of the UART ports

Power management, autobaud detection, and DMA are not supported by the UART.

32.1 SC16C652/ST16C2552 UART Hardware Operation

UART controller is the key component of the serial communications subsystem of a computer. At the destination, a second UART re-assembles the bits into complete bytes. During transmission, the UART converts the bytes from the processors parallel bus to the serial bit stream. During receiving, the UART builds the serial bits into a parallel byte.

The UART is interfaced to the Peripheral Bus Controller (PBC). The PBC is used to interface the CPU board bus with the busses used by peripherals. It provides an LPC (Low Pin Count) interface for access to the on-board UART controller.

The block diagram of the PBC interface is shown in Figure 32-1:

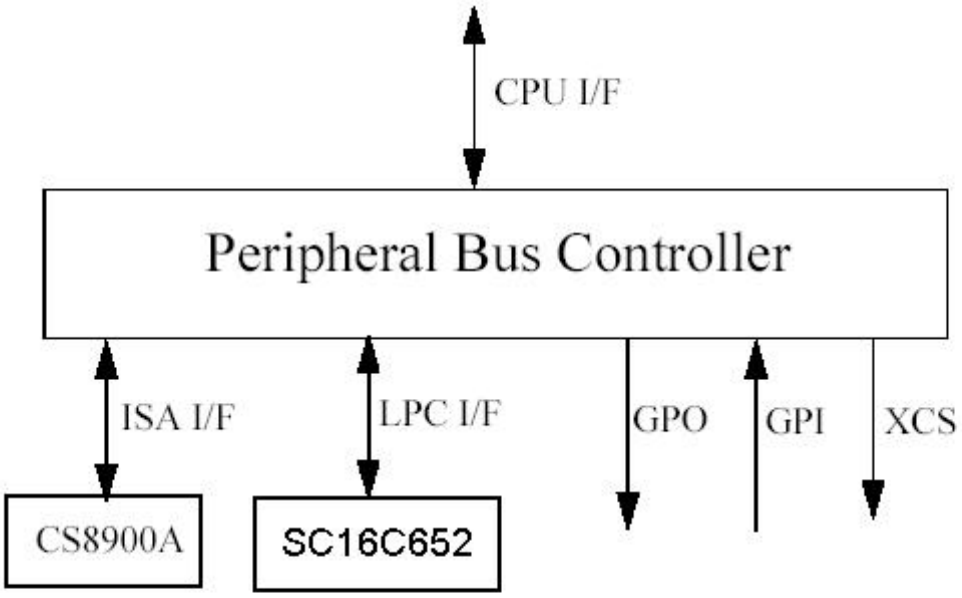


Figure 32-1. SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) PCB Interface

The block diagram of SC16C652/ST16C6552 is shown in Figure 32-2.

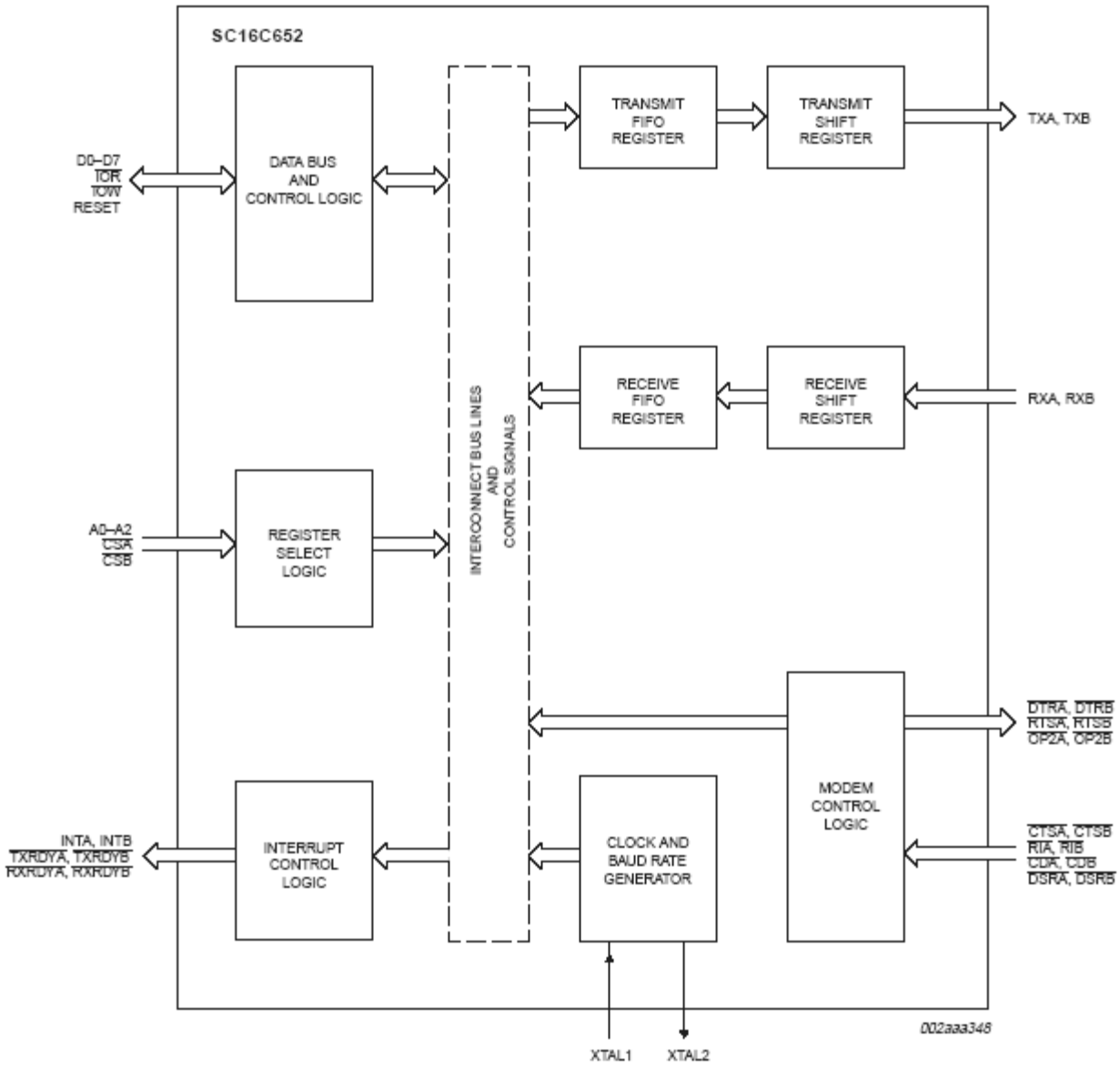


Figure 32-2. SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) Block Diagram

The block diagram of 16C652 UART driver is shown in Figure 32-3.

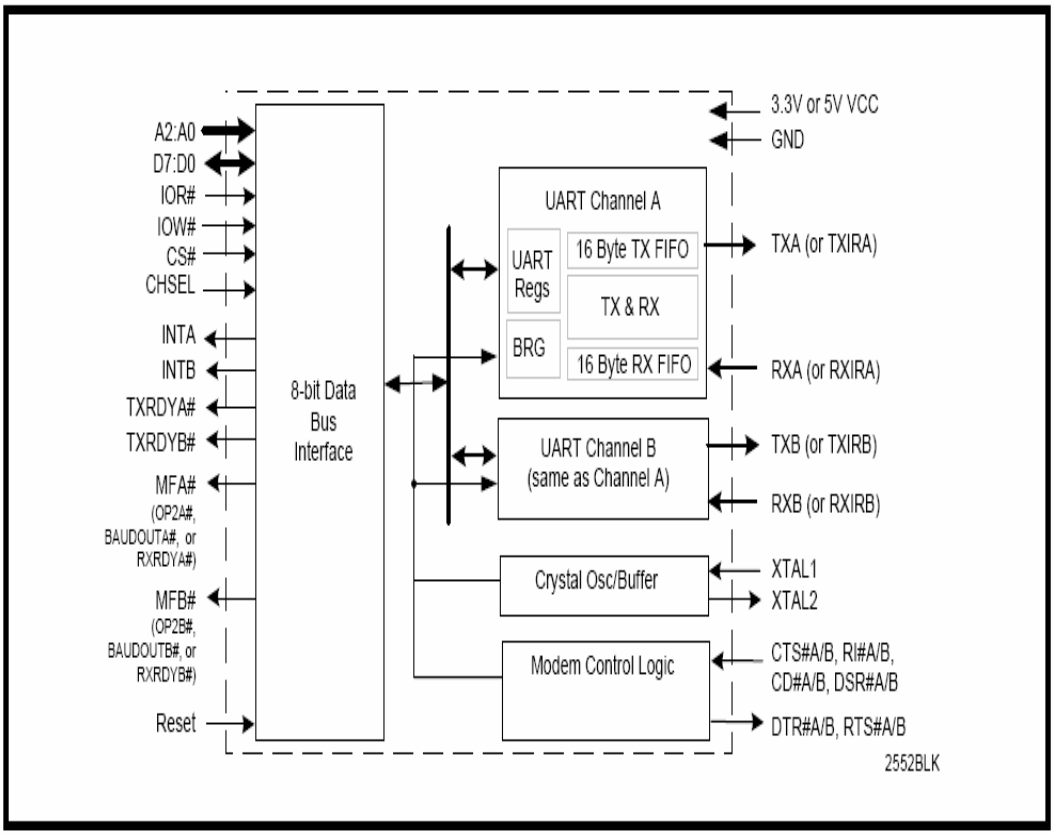


Figure 32-3. 16C652 UART Driver Block Diagram

The base address and the offset give the location of the UART register. Addressing of the accessible registers of the Serial Port is listed in Table 32-1.

Table 32-1. Serial Port Register Addressing for the SC16C652/ST16C2552 UART

DLAB[a]	A2	A1	A0	Register Name
0	0	0	0	Receive Buffer (read/write)
0	0	0	0	Transmit Buffer (read/write)
0	0	0	1	Interrupt Enable (read/write)
X	0	1	0	Interrupt Identification (read/write)
X	0	1	0	FIFO Control (read/write)
X	0	1	1	Line Control (read/write)
X	1	0	0	Modem Control (read/write)
X	1	0	1	Line Status (read/write)
X	1	1	0	Modem Status (read/write)
X	1	1	1	Scratch pad (read/write)

Table 32-1. Serial Port Register Addressing for the SC16C652/ST16C2552 UART (Continued)

DLAB[a]	A2	A1	A0	Register Name
1	0	0	0	Divisor LSB (read/write)
1	0	0	1	Divisor MSB (read/write)

[a] DLAB – 7th bit of Line Control Register.

32.2 SC16C652/ST16C2552 UART Software Operation

The SC16C652/ST16C2552 UART provides standard serial driver interface to the Linux operating system. The driver works in interrupt mode and makes use of the UART FIFO for optimum operation.

For more information, see the documentation for *Linux 2.6 Serial Port Drivers*.

32.3 SC16C652/ST16C2552 UART Requirements

The SC16C652/ST16C2552 UART has the following requirements:

- The 16C652 UART is used as a debug port to bring up the console for development purposes.
- The SC16C652/ST16C2552 UART driver, at a minimum must support baud rates up to 115200bps with no handshaking, 8-bit character length, 1 Stop bit, and no parity.
- The 16C652 UART driver must support a minimum of one 16C652 UART.

32.4 SC16C652/ST16C2552 UART Source Code Structure

Table 32-2 lists the files associated with SC16C652/ST16C2552 UART development:

Table 32-2. 16552 UART Driver File List

File	Description
`8250.h`	Driver header file
`8250.c`	Driver source file
`serial.h`	Architecture specific configuration header file

32.5 16C652 UART Driver Configuration

This section documents the configuration options available for the external UART.

32.5.1 Linux Menu Configuration Options

The following menu options are available for the external 16C652 UART. These options are available under Device Drivers -> Characters devices -> Serial drivers:

1. 8250/16550 and compatible serial support = Y
2. Console on 8250/16550 and compatible serial port = Y
3. Maximum number of non-legacy 8250/16550 serial ports = two (one for ST16C6552)

By default, for all architectures, the first two options are N and the last option is four.

32.5.2 Source Code Configuration Options

32.5.2.1 Chip Configuration Options

The following chip-specific configuration options for the driver are provided in `serial.h`:

- UART1 interrupt (IRQ_UARTINT0) - This specifies the UART1 interrupt number.
- UART2 interrupt (IRQ_UARTINT1) - This specifies the UART2 interrupt number.
- UART1 base address (SERIAL1_BASE_ADDRESS) - This specifies the UART1 base address.
- UART2 base address (SERIAL2_BASE_ADDRESS) - This specifies the UART2 base address.
- PBC Base address (PBC_BASE_ADDRESS) - This specifies the PBC base address.
- PBC IRQ status (IRQSTAT) - PBC interrupt status register.

32.6 SC16C652/ST16C2552 UART Programming Interface

The SC16C652/ST16C2552 UART implements all the methods that are required by the Linux serial API to interface with the UART ports. It implements and provides a set of control methods to the core UART driver present in Linux. For more information, see the API documents for the programming interface.

32.7 16C652 UART Driver Interrupt Requirements

The system requirements are listen in Table 32-3.

Table 32-3. 16C652 UART Interrupt Requirements

Parameter	Equation	Typical	Worst-Case
Rate	(BaudRate / 10)) × (1 / Rxtl + 1 / 32)	5940 /sec	118800 /sec
Latency	320 / BaudRate	5.6 msec	213.33 msec

The typical values are based on a baud rate of 57600 with a receiver trigger level (Rxtl) of 1. Transmitter trigger level is not set. The worst-case is based on a baud rate of 1.152 Mbps (max supported by the UART interface) with an Rxtl of 1. The interrupt latency for typical (9600) and lowest (50) baud rates.

32.8 Device-Specific Information

Features of the SC16C652/ST16C2552 UART for the i.MX family of processors supports one UART port on the MX27 board.

The following menu options are available for the external 16C652 UART. These options are found under Device Drivers -> Characters devices -> Serial drivers:

Maximum number of non-legacy 8250/16550 serial ports = two (one for an i.MX27 board)

Chapter 33
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

This driver is present only in the application processor architectures.

33.1 Architectural Overview

A USB host system is composed of a number of hardware and software layers. Figure 33-1 illustrates a conceptual block diagram of the building block layers in a host system that work in concert to support USB 2.0.

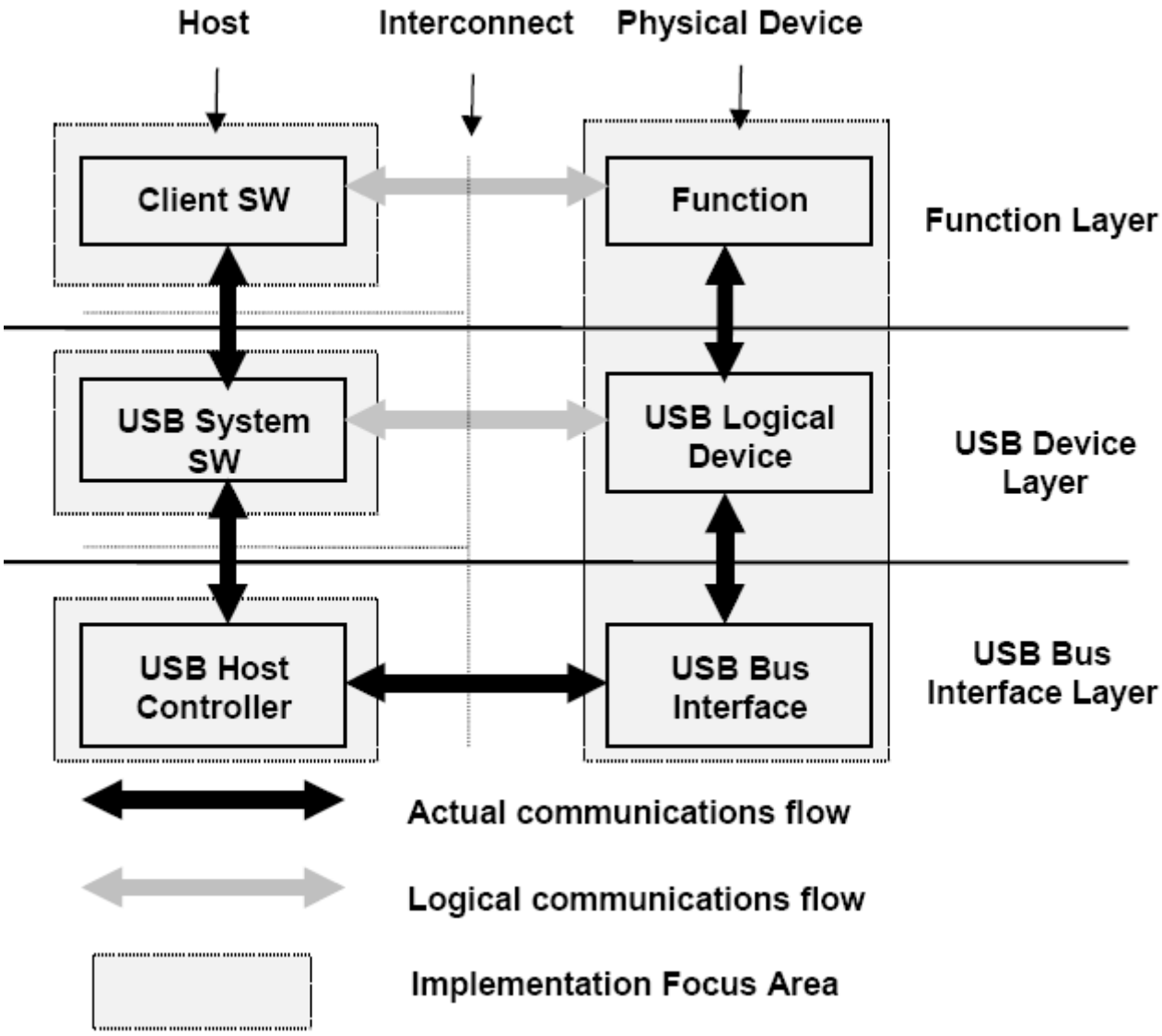


Figure 33-1. Block Diagram

33.2 Hardware Operation

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

33.3 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,
};
1. fsl_ep_enable():configure endpoint, making it usable.
2. fsl_ep_disable(): endpoint is no longer usable
3. fsl_alloc_request(): allocate a request object to use with this endpoint.
4. fsl_free_request(): frees a request object.
5. arcotg_ep_queue():queues (submits) an I/O request to an endpoint.
6. arcotg_ep_dequeue(): dequeues (cancels, unlinks) an I/O request from an endpoint
7. arcotg_ep_set_halt(): sets the endpoint halt feature.
8. 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.

33.4 Requirements

- The USB stack shall support USB device mode
- The USB stack shall support mass storage device profile – subclass 8-1. (RBC set)
- The USB stack shall support USB host mode
- The USB stack shall support HID host profile – subclasses 3-1-1 and 3-1-2. (USB mouse and keyboard)
- The USB stack shall support mass storage host profile – subclass 8-1
- The USB stack shall support Ethernet USB profile – subclass 2
- The USB stack shall support DC PTP transfer

- The USB stack shall support MTP device mode

33.5 Source Code Structure

Table 33-1 lists the source files available in the source directory, `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/usb`.

Table 33-1. USB Driver File List

File	Description
host/ehci-hcd.c	host driver source file.
host/ehci-arc.c	host driver source file.
host/ehci-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.
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 33-2 lists the platform related source files in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.22/include/asm-arm/arch-mxc`.

Table 33-2. USB Platform Source File List

File	Description
arc_otg.h	USB register define.

Table 33-3 lists the platform-related source files in the directories:

`<ltib_dir>/rpm/BUILD/linux-2.6.22/arch/arm/mach-mx27/`

Table 33-3. USB Platform Header File List

File	Description
usb.c	Platform-related initialization

Table 33-4 lists the common platform source files in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.22/arch/arm/plat-mxc`.

Table 33-4. USB Common Platform File List

File	Description
isp1301xc.c	ISP1301 USB driver
isp1504xc.c	ISP1504 USB driver
mc13783_xc.c	mc13783USB d river
serialxc.c	internal serial transceiver driver
usb_common.c	common platform related part of USB driver

33.6 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_USB_OTG—Build support for otg, allows to switch from device to host role using the ID pin of the usb connector.
- 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_OTG—Build support for selecting the ARC EHCI OTG host. In menuconfig, this option is available under Device Drivers --> USB support --> Support for OTG host port on Freescale controller. By default, this option is Y.
- 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.
- 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 M.
- 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_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 M.
- CONFIG_USB_GADGET_ARC—Build support for ARC USB gadget. In menuconfig, this option is available under Device Drivers --> USB support --> USB Gadget Support ---> 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 --> Support for OTG peripheral port on Freescale controller. By default, this option is Y.
- CONFIG_USB_ETH—Build support for Ethernet gadget. In menuconfig, this option is available under Device Drivers --> USB support --> USB Gadget Support --> Ethernet Gadget. By default, this option is M.
- CONFIG_USB_ETH_RNDIS—Build support for Ethernet RNDIS protocol. In menuconfig, this option is available under Device Drivers --> USB support --> USB Gadget Support --> RNDIS support (EXPERIMENTAL). By default, this option is Y.
- CONFIG_USB_FILE_STORAG—Build support for Mass Storage gadget. In menuconfig, this option is available under Device Drivers --> USB support --> USB Gadget Support --> File-backed Storage Gadget. By default, this option is M.
- CONFIG_USB_G_SERIAL—Build support for ACM gadget. In menuconfig, this option is available under Device Drivers --> USB support --> USB Gadget Support --> Serial Gadget. By default, this option is M.
- 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 OTG transceiver --> 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 OTG transceiver --> 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 OTG transceiver --> Philips ISP1301”.
- CONFIG_USB_EHCI_FSL_MC13783 -Build support for selecting MC13783 transceiver for Host. In menuconfig, “Device Drivers --> USB support --> Select OTG transceiver --> Freescale MC13783”.

33.7 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.

33.7.1 Notes

Table 33-5. Default USB Settings

Default value	OTG HS	OTG FS	Host1(FS)	Host2(HS)	Host2(FS)
i.MX27 3-Stack	enabled	disabled	N/A	N/A	N/A

To enable OTG HS: CONFIG_USB_EHCI_ARC_OTGHS = y

To enable Host2: CONFIG_USB_EHCI_ARC_H2 = y

To use OTG host and device : CONFIG_USB = y, CONFIG_USB_EHCI_HCD = m, CONFIG_USB_OTG = y, CONFIG_USB_GADGET = m, CONFIG_USB_OTG_WHITELIST = n, CONFIG_USB_GADGET_ARC_OTG = y. Note : when USB_OTG is enabled, USB_EHCI_HCD and USB_GADGET cannot be used as built-in. The OTG module (isp1504_arc.ko) must be loaded first, before the host (ehci-hcd.ko) and the device (arcotg_udc.ko) module.

Chapter 34
ATA Driver

34.1 Hardware Operation

The detailed hardware operation of ATA is detailed in the hardware documentation.

34.2 Software Operation

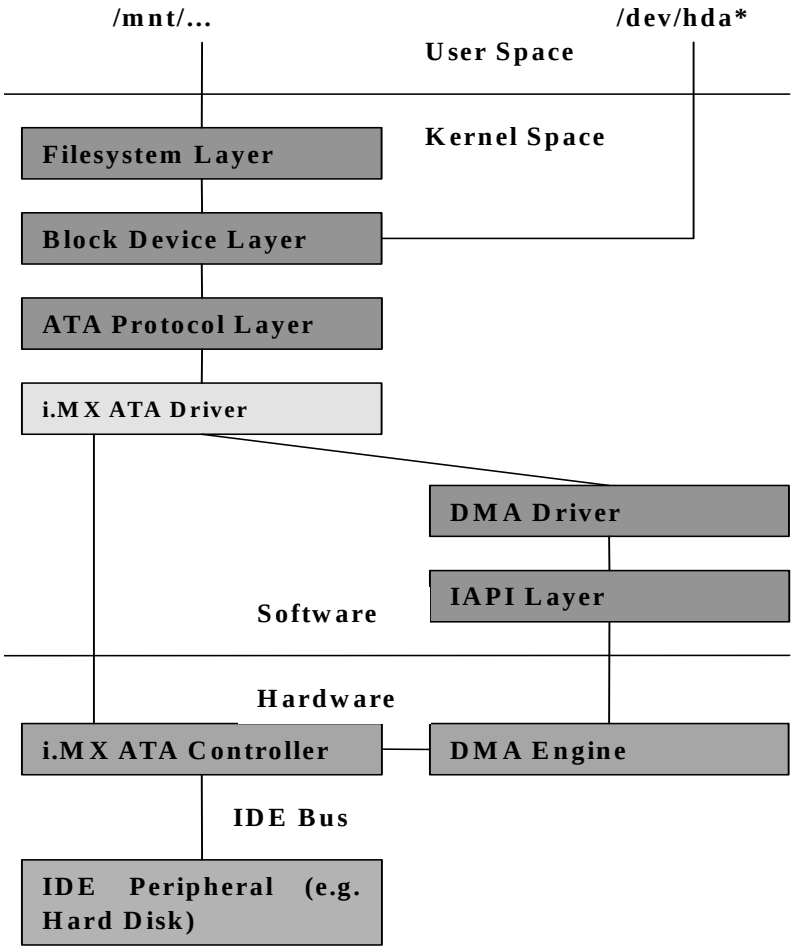


Figure 34-1.

The i.MX ATA/IDE driver sits beneath the IDE layer of the block device infrastructure of the Linux kernel (see the figure listed above). It handles the details of the integrated ATA controller, while the IDE layer understands and executes the IDE and ATA protocols. The IDE device, such as a hard disk, is exposed to the application in user space by the /dev/hda* interface. Filesystems are built upon the block device. The integrated DMA engine, which assists the ATA controller hardware in the DMA transfer modes, is accessed through the Linux DMA driver.

34.3 Source Code Structure Configuration

Table 34-1 lists the source files available in the `linux/drivers/ide/arm` directory.

Table 34-1. ATA Driver File List

File	Description
`mxc_ide.h`	ATA driver header file
`mxc_ide.c`	ATA driver source code

34.4 Linux Menu Configuration Options

Enable the following packages:

- hdparm
- e2fsprogs
- Set modutils "module-init-tools"

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

Enable the following kernel configuration options as either modules (M) or built-in to the kernel (Y). These options are all under "Device Drivers ---> ATA/ATAPI/MFM/RLL support":

- ATA/ATAPI/MFM/RLL support
- Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support
- Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support ---> Include IDE/ATA-2 DISK support

Enhanced IDE/MFM/RLL disk/cdrom/tape/floppy support ---> Freescale MXC IDE support

34.5 Board Configuration Options

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

34.6 Programming Interface

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

34.7 Usage Example

1. After building the kernel and the ATA driver and deploying, boot the target, and log in as root.
2. On the target, run these commands if you configured the driver as modules. If you built the driver into the kernel, you'll see similar messages in the console boot log.

```
# modprobe mxc_ide
# modprobe ide-disk
```

You should see messages similar to the following:

```
freescale# modprobe mxc_ide
Uniform Multi-Platform E-IDE driver Revision: 7.00alpha2
ide: Assuming 50MHz system bus speed for PIO modes; override with
```

```
idebus=xx
MXC: IDE driver, (c) 2004-2005 Freescale Semiconductor
mxc_ide_resetproc: resetting ATA controller
hda: HTC426020G7AT00, ATA DISK drive
ide0 at 0xd408c0a0-0xd408c0a7,0xd408c0d8 on irq 15
hda: max request size: 512KiB
hda: 39070080 sectors (20003 MB), CHS=16383/255/63
hda: cache flushes supported
hda: hda1
ide0: read chan=31 (42 BDs), write chan=30 (42 BDs)
hda: enabling UDMA3 mode
```

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

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

34.8 Usage Example

Create Partitons

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

```
$fdisk -l /dev/hda
```

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

```
$fdisk /dev/hda
```

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

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

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

```
$dd if=/dev/zero of=/dev/hda1 bs=1024 count=5
```

The response should be as follows:

5+0 records in

5+0 records out

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

```
$dd if=file_name of=/dev/hda1
```

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

```
$dd if=/dev/hda1 of=output_file bs=1024 count=1
```

Files System Tests

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

```
$mkfs.ext2 /dev/hda1
$mkfs.vfat /dev/hda1
```

Mount the file system as follows:

```
$mkdir /mnt/hda1
$mount -t ext2 /dev/hda1 /mnt/hda1
```

After mounting, file/directory, operations can be performed in `/mnt/hda1`.

Unmount the filesystem as follows:

```
$umount /mnt/hda1
```

Chapter 35
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)

35.1 Hardware Operation

The RTC's 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.

35.2 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 2Hz, 4Hz, 8Hz, 16Hz, 32Hz, 64Hz, 128Hz, 256Hz, and 512Hz given the clock input of 32.768 kHz (Other clock input frequencies are not supported by the driver.) The 1Hz periodic interrupt is also called "update interrupt" (UIE).

NOTE

The i.MX RTC driver implementation follows what is stated in the `rtc.txt` file under Linux kernel `Documentation` directory that "Programming and/or enabling interrupt frequencies greater than 64 Hz is only allowed by root."

35.3 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.

35.4 Source Code Structure

The RTC module is implemented in the following files:

```
<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/rtc/rtc-mxc.c
```

The source file, `mxc-rtc.c`, for the RTC specifies the RTC function implementations.

35.5 Programming Interface

All the Linux RTC functions are implemented in the `time.c` file.

NOTE

The `<ltib_dir>/rpm/BUILD/linux-2.6.22/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     -
```

For more information, see the API documentation for the detailed programming interface.

Chapter 36
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.

36.1 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.

36.2 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.

36.2.1 Generic WDOG driver

This is implemented in the `<ltib_dir>/rpm/BUILD/linux-2.6.22/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.

36.2.1.1 Requirements

This WDOG implementation meets the following requirements:

- The WDOG module generates the reset signal if it is enabled but not serviced within a predefined timeout value.
- The WDOG module does not generate the reset signal if it is serviced within a predefined timeout value.
- The WDOG module provides IOCTL/read/write required by the standard WDOG subsystem.

36.2.1.2 Source Code Structure

The WDOG source code is in `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/watchdog/mxc_wdt.c` and `mxc_wdt.h`.

Table 36-1 lists the source files for WDOG.

Table 36-1. WDOG File List

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

36.2.1.3 Programming Interface

For more information, see the API documentation for the detailed programming interface.

36.2.2 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 36-1 shows the flow chart for the operation. It applies to all platforms with two WDOGs.

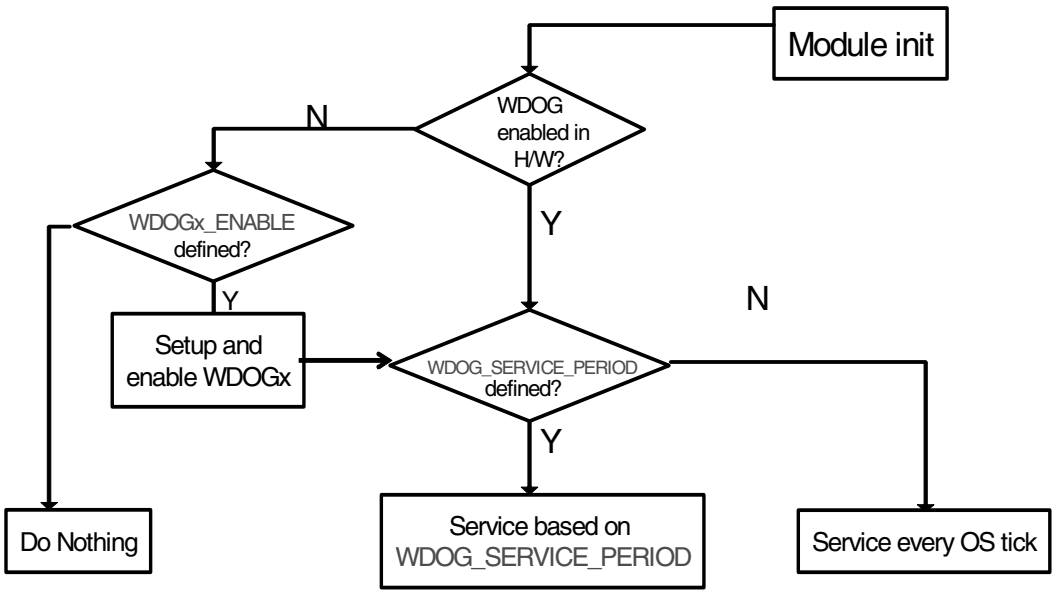


Figure 36-1. WDOG Software Operation Flow Chart

36.2.2.1 Requirements

This WDOG implementation meets the following requirements:

- The WDOG module generates the reset signal if it is enabled but not serviced within a predefined timeout value.

- The WDOG module does not generate the reset signal if it is serviced within a predefined timeout value.
- The second WDOG (when present) shall generate an interrupt if it is enabled but not serviced within a predefined timeout value.

36.2.2.2 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.

The source files for WDOG is `time.c`, which specifies WDOG function implementations.

36.2.2.3 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 37
FM Driver

Si4702 is adopted 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.

37.1 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.

Capabilities:

- 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

37.1.1 Hardware Operation

Si4702 supports both three-wire control and two-wire control. Two-wire control is chosen by drive SEN pin high while chip boot up, which is done by hardware design.

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, comprised of 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 will be requested. Data transfer ceases with the STOP command. After every STOP command, the internal address counter is reset.

FM analog signals will directly connect to audio chip which will route it out to headset.

37.1.2 Software Operation

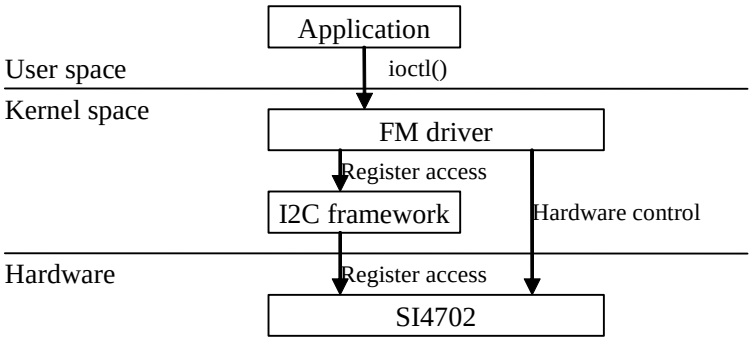


Figure 37-1. Software Operation

The FM driver serves as an interface between kernel and user space. The driver can control hardware directly, such as reset operation, but most of the functional operation is taken using I2C 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 I2C 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 command from user space, execute them and feed back information if there is any.

37.2 Source Code Structure Configuration

Table 37-1 lists the source files associated with the FM driver that are available in the directory, `<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/char/`.

Table 37-1. FM Driver Source and Header File List

File	Description
`si4702.c`	Source file for SI4702 FM driver
`si4702.h`	Header file for SI4702 FM driver

37.3 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 si4702.ko
```

It is located in `/lib/modules/2.6.22-pdk27_rel1/kernel/drivers/char`.

Chapter 38
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.

38.1 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

38.2 Driver Requirements

The MMA7450L driver is based on the I2C driver, and also makes use of the hardware monitoring system and the input polling device, so the driver must be supported in the Linux kernel.

38.3 Driver Architecture

The software design will touch several part of Linux kernel, such as, I2C driver, input poll device and hardware monitor system. See the following illustration for the complete software architecture.

The MMA7450L provides two methods of register access, I2C and SPI. This driver uses the I2C. At driver initial phase, a I2C client is registered to the I2C system, and it will be used during the whole process of MMA7450L operations.

The MMA7450L registers itself to the hardware monitor system and the input polling device that provide user access facilities.

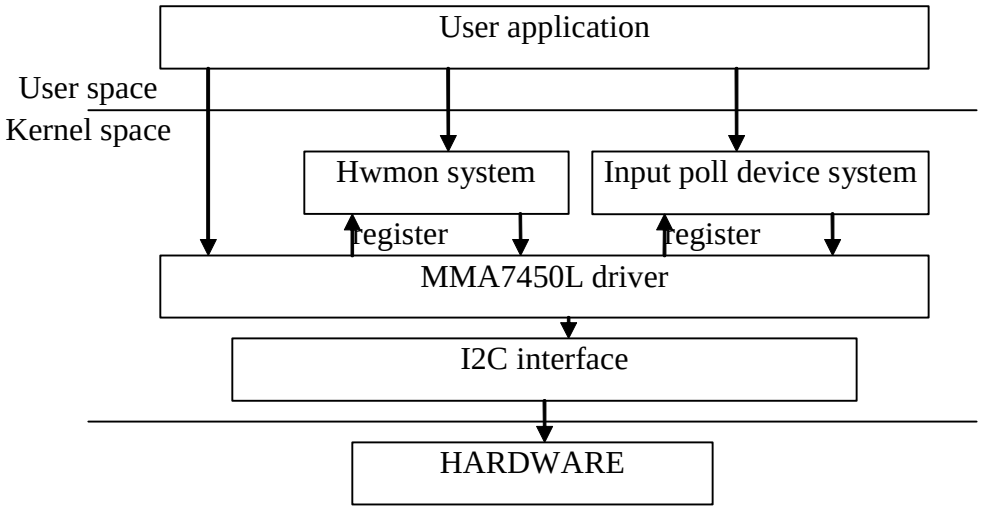


Figure 38-1. Driver Architecture

38.4 Driver Source Code Structure

The driver source code structure contains only one file:

`<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/hwmon/mxc_mma7450.c`

Table 38-1. Driver Source Code Structure File

File	Description
mxc_mma7450.c	Implementation of the mma7450 accelerometer driver.

38.5 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module. In order to get to the mma7450 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.

Device Drivers->Hardware Monitoring support->MMA7450 device driver

Chapter 39
Global Positioning System (GPS) Driver

39.1 GPS Driver Overview

An external global positioning system (GPS) module can be supported through the serial port and necessary GPIO resources. Currently, the PDK supports Broadcom's Barracuda Single Chip A-GPS Solution. Since this chip set features a host-based architecture, several software components need to be loaded on the platform to enable full operation. Figure 39-1 shows a coarse block diagram of the complete GPS system architecture consisting of the BCM4750 Barracuda GPS IC and the host CPU.

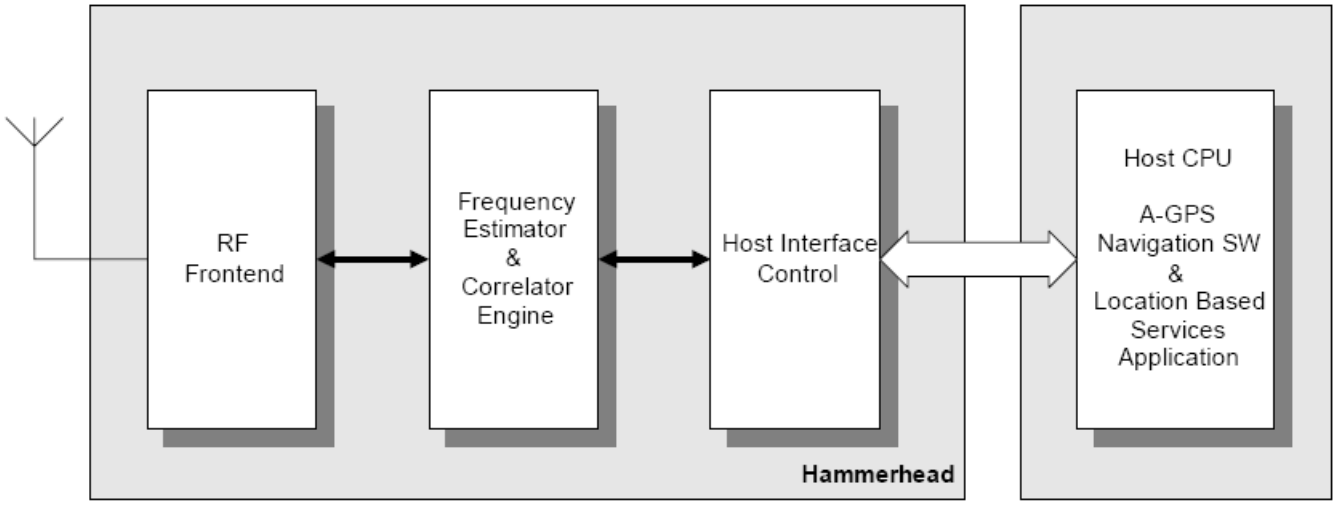


Figure 39-1. Barracuda GPS Coarse System Architecture including Host CPU

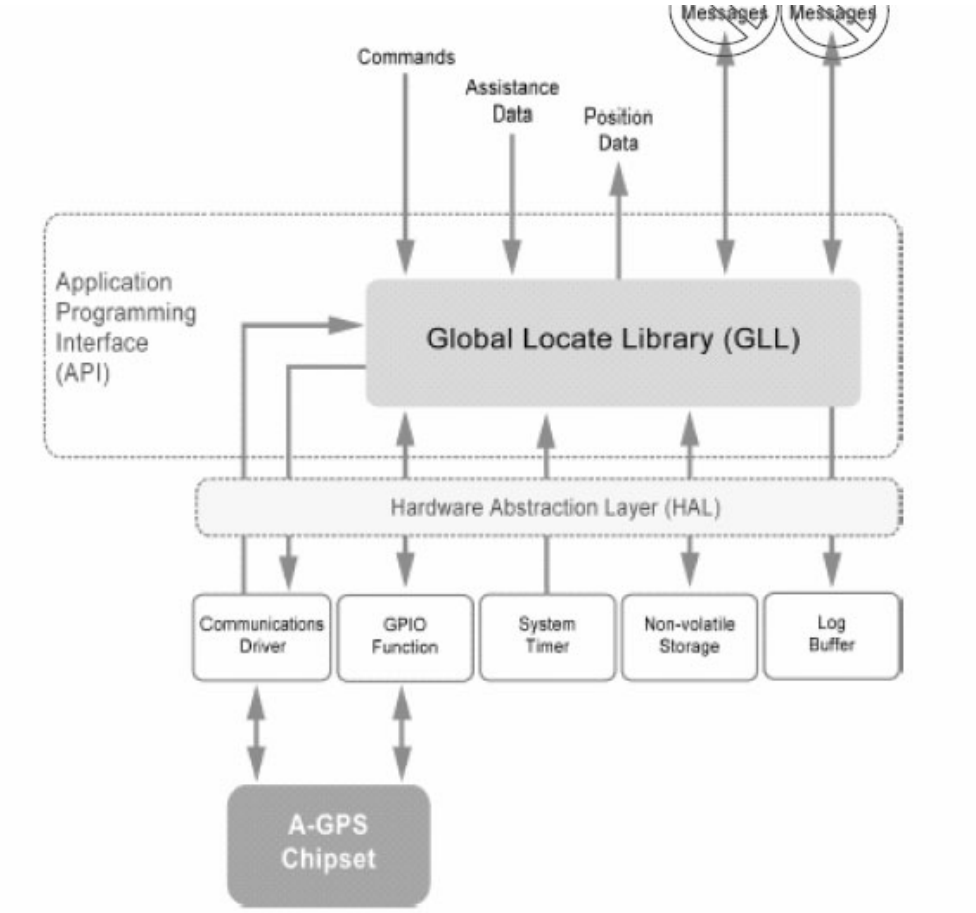


Figure 39-2. GL GPS SW Architecture

Figure 39-2 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, is used to provide 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 used to track 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.

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.

39.2 Hardware Operation

The GPS daughter board should connect to the UARTx slot of the platform hardware. Since GPS is very sensitive to timing accuracy, the platform should provide 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 must be done. The reset pin is asserted for a few milliseconds, then de-asserted. It's done when the gpiodrv is loaded. When GPS is launched, the power pin is asserted.

39.2.1 UART Port

The GPS module uses UARTx to communicate with the host; the device name in Linux is shown below.

Table 39-1. UART Port

Platform	UART	Device Name
i.MX27	UART2	/dev/ttymxc1

39.2.2 GPIO Control

GPIO pins are used to control the GPS module.

Table 39-2. GPIO Control Signals

GPIO Name PIN	Value	Description
MX27_PIN_I2C2_SDA		0: Reset of GPS module is asserted; 1: Reset of GPS module is de-asserted
MX27_PIN_USBH2_CLK		1: GPS module is power on; 0: GPS module is power off

39.2.3 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 39-3. 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

39.3 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 assistance data as showing in architecture chart.

Software applications communicate with the GPS module through a NMEA pipe where you need to read the incoming NMEA data. The GPS application will create a NMEA pipe then get all the NMEA sentences from this pipe.

39.3.1 GLGPS Configuration

The GLGPS reads configuration from an XML file. This configuration file defines the hardware-dependent settings, paths, and different tasks.

The XML file should be like this:

```
<?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>
```

Hal tag defines the glhal settings (serial COM settings, enable hal log, some other paths). Parameters are explained on Table 39-4.

Table 39-4. hal Attributes

hal Attributes	Description	Comment
acPortName	serial COM port	
lBaudRate	baud rate	9600,19200,38400 ,57600,115200
acLogDirectory	directory where the logs will be placed	

gll defines gll specific parameters, as explained in Table 39-5.

Table 39-5. 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

In this example different tasks are configured. Task named "*normal*" makes autonomous fix every second and "cold" starts autonomous fixes with no assisted data.

39.3.2 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.

39.3.2.1 Linux Menu Configuration Options

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. In menuconfig, it can be located under Device Drivers-> MXC Support Drivers-> Broadcom GPS ioctrl support -> GPS ioctrl support. By default, this option is M for all architectures.

39.3.3 Source Code

Table 39-6 lists the source files available in the source directory

```
<ltib_dir>/rpm/BUILD/linux-2.6.22/drivers/mxc/gps_ioctrl
```

Table 39-6. 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

39.3.4 LTO Feature (optional)

The Long Term Orbit (LTO) feature is a way to get assistance data for the device. The assistance data used in the PDK is valid for several days, giving PDK users the advantages of assisted GPS performance while preserving the freedom of autonomous GPS operation.

The PDK should be connected through a NFS server and you should put the LTO file in the folder that the GPS driver is pointing to. It is the responsibility of the customer to implement the way to get a TCP/IP connection on the platform (Bluetooth, IP over USB).

39.3.4.1 Enabling The LTO Feature on the Platform

You must contact Broadcom in order to obtain the necessary license for enabling this feature on the Freescale PDK platform. You must submit the following information to obtain the license: platform name and platform type.

You must also ensure that the LTO feature is enabled through the different parameters that the GPS driver can take as an input.

39.3.5 Power Management

The GPS driver manages automatically the power management of the Broadcom chip set by toggling the different IOs of the chip set. The GPS chip set will go in Low Power Standby mode when you are stopping the GPS driver.

39.3.6 irm Commands.

While the glgps is running some command 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 ingore 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 40
Unit Tests

The unit tests are tests written by the Linux BSP developers to check the drivers they have added. They can be useful in understanding how to use a particular driver and in checking that changes have not broken anything.

40.1 Enabling the Unit Tests

In order to enable the unit test packages with all the test applications that will be mentioned in the following sections, enter the following commands:

```
./ltib -c
Package list -> mxc-misc
```

40.2 Running Tests That Write to a File

On most platforms, the unit tests directory is on a read-only partition. One way to create a writable partition is to do the following:

```
mkdir /tmp/tmp2
mount -t tmpfs whatever /tmp/tmp2
export TMP=/tmp/tmp2
```

Then run the tests from `/tmp/tmp2` by specifying the full path: `/unit_tests/<test name>`.

40.3 MSL Tests

This module is not testable except by noting the correct behavior when other modules use it.

40.4 Dynamic Frequency Scaling Tests

These test cases demonstrate the possibility for an application to dynamically change the processor's frequency. This allows power saving when the CPU idle. This can be achieved using the command /sys/devices/system/cpu/cpu0/cpufreq.

The test cases in Section 40.4, "Dynamic Frequency Scaling Tests" are available for i.MX27 platforms.

Test ID : FSL-UT-CPUFREQ-0010

This test case displays all the avilable governors and put the system under the «userspace» governor. If the «userspace» governor is not listed as an available governor, you need to enable it in the kernel configuration prior to running this test.

```
$ cd /sys/devices/system/cpu/cpu0/cpufreq
$ cat scaling_available_governors
conservative ondemand powersave userspace performance
$cat cpuinfo_cur_freq
399000
$ echo userspace > scaling_governor
```

End of Test FSL-UT-CPUFREQ-0010

Test ID : FSL-UT-CPUFREQ-0020

This test case shows how to change the CPU frequency to 266 MHz.

```
$ echo 266000 > scaling_setspeed
$ cat cpu_info_curfreq
266000
```

End of Test FSL-UT-CPUFREQ-0020

Test ID : FSL-UT-CPUFREQ-030

This test case shows how to change the CPU frequency to 133 MHz.

```
$ echo 133000 > scaling_setspeed
$ cat cpu_info_curfreq
133000
```

End of Test FSL-UT-CPUFREQ-030

Test ID : FSL-UT-CPUFREQ-040

This test case put the system under the «conservative» governor. It then shows how the frequency is dynamically adjusted when overloading the system by performing a MD5 sum computation.

```
$ echo conservative > scaling_governor
$ cat cpu_info_curfreq
133000
$ md5sum /dev/urandom &
$ cat cpu_info_curfreq
399000
$ killall md5sum
$ cat cpu_info_freq
```

End of Test FSL-UT-CPUFREQ-040

40.5 DMAC Tests

The test cases in Section 40.5, “DMAC Tests“ are available for i.MX27 platforms.

There is a DMA test application under `<ltib_dir>/rpm/BUILD/imx-test-2.3.0/test/mxc_udma_test` directory, and it needs the support from a DMA test driver `mxc_udma_testdriver.c` which is under `<ltib_dir>/rpm/BUILD/imx-test-2.3.0/module_test/mxc_udma_testdriver.c`. The test could be executed as following:

Test ID: FSL-UT-UDMA-0010

```
$cd unit_tests
$insmod mxc_udma_testdriver.ko
register virtual dma driver.
dma major number = 252
$mknod /tmp/dma c 252 0
$cd ../mxc_dma_test/
$./mxc_dma_test.out 0 /*1D memory to memory dma is tested*/
...
Verify 1d mem c7202000,x=-81,p[10]=-81
Verify Success
begin dma free
End dma free
```

End of Test FSL-UT-UDMA-0010

Test ID: FSL-UT-UDMA-0020

```
$./mxc_dma_test.out 1 /*1D memory to 2D memory dma is tested*/
...
Verify 2d mem c7172000,x=-87,p[2*128+1+0]=-87
Verify Success
begin dma free
End dma free
```

End of Test FSL-UT-UDMA-0020

40.6 PMIC Protocol Tests

The test cases in Section 40.6, “PMIC Protocol Tests“” are available for i.MX27, platforms.

The source code for the unit test applications is available only after running the following command on ltib:

`/<ltib_dir>/ltib -p imx-test -m prep`

then, the source code for the unit test applications is available at the following location:

`<ltib_dir>/rpm/BUILD/misc/test/mxc_pmic_test/protocol_tests/`

The built unit test applications are available at `/<ltib_dir>/rootfs/unit_tests`.

Use the `pmic_testapp.out` program to test the protocol driver.

40.6.1 MC13783 PMIC Protocol Driver Read/Write Unit Test

The RW unit test checks the read/write functionality of the MC13783 protocol driver on a register.

```
Type: ./pmic_testapp.out -T RW
```

If the test has run correctly, the result is as follows:

```
mc13783_testapp    1  PASS  : mc13783_testapp test worked as expected
```

40.6.2 PMIC Subscribe/Unsubscribe to an Event Unit Test

The SU unit test checks the ability to subscribe to an event using the MC13783 PMIC.

```
Type : ./pmic_testapp.out -T SU
```

If the test has run correctly, the result is as follows:

```
mc13783_testapp 1  PASS  : mc13783_testapp test worked as expected
```

40.6.3 PMIC Subscribe, Interrupt, Unsubscribe Unit Test

Test ID: FSL-UT-PMIC-Protocol-0010

The S_IT_U unit test checks the interrupt handling capability of the MC13783 protocol driver using the TSI (touchscreen) event.

```
Type : ./pmic_testapp.out -T S_IT_U
```

If the test has run correctly, the result is as follows:

```
Press PWR button on the KeyBoard or MC13783
you should see IT callback info
Press Enter to continue after pressing the button

DETECTED PMIC EVENT : 27

Did you see the callback info  [Y/N]Y
Test subscribe/unsubscribe 2 event = 27

Press PWR button on the KeyBoard or MC13783
you should see IT callback info twice.
Press Enter to continue after pressing the button

 DETECTED PMIC EVENT : 27
 DETECTED PMIC EVENT : 27

Did you see the callback info twice [Y/N]Y
mc13783_testapp    1  PASS  :  mc13783_testapp test worked as expected
```

End of Test FSL-UT-PMIC-Protocol-0010

40.6.4 PMIC Open/Close Unit Test

The OC unit test checks the ability to Open and Close API of the MC13783 PMIC.

Test ID: FSL-UT-PMIC-0010

```
Type : ./pmic_testapp.out -T OC
```

If the test has run correctly, the result is as follows:

```
mc13783_testapp 1  PASS  : mc13783_testapp test worked as expected
```

End of Test FSL-UT-PMIC-0010

40.6.5 PMIC Concurrent Access Unit Test

The CA unit test checks the ability for Concurrent Access API of MC13783 PMIC.

Test ID: FSL-UT-PMIC-0010

```
Type : ./pmic_testapp.out -T CA
```

If the test has run correctly, the result is as follows:

```
mc13783_testapp 1  PASS  : mc13783_testapp test worked as expected
```

End of Test FSL-UT-PMIC-0010

40.7 PMIC Audio Tests

The test cases in Section 40.7, “PMIC Audio Tests“” are available for i.MX27 and i.MX31 platforms.

ALSA test applications either in the native mode or in the OSS emulation mode should presently suffice to verify the PMIC Audio driver APIs.

40.8 PMIC Digitizer Tests

The test cases in Section 40.8, “PMIC Digitizer Tests“” are available for i.MX27 and i.MX31 platforms.

Use the `pmic_testapp_adc.out` program to test the MC13783-specific version of this device driver.

40.8.1 PMIC Read ADC Value Tests

This test returns the last correct touch screen value of MC13783 ADC.

Test ID: FSL-UT-PMIC-ADC-0010

Type the following:

```
./pmic_testapp_adc.out -T CONV
```

The system returns:

```
pmic_testapp_adc    0  INFO  :  Testing if 47292 test case is OK

========== TESTING PMIC adc DRIVER ==========
Select a channel [0-15] :
=>0
Test convert ADC functions
Channel 0: 0
Convert 8x channel 0 - 0: 0
Convert 8x channel 0 - 1: 0
Convert 8x channel 0 - 2: 0
Convert 8x channel 0 - 3: 0
Convert 8x channel 0 - 4: 0
Convert 8x channel 0 - 5: 0
```

```
Convert 8x channel 0 - 6: 0
Convert 8x channel 0 - 7: 0
MULTICHANNEL From channel 0 - 0: 0
MULTICHANNEL From channel 0 - 1: 0
MULTICHANNEL From channel 0 - 2: 860
MULTICHANNEL From channel 0 - 3: 416
MULTICHANNEL From channel 0 - 4: 1023
MULTICHANNEL From channel 0 - 5: 31
MULTICHANNEL From channel 0 - 6: 2
MULTICHANNEL From channel 0 - 7: 290
pmic_testapp_adc    1  PASS  :  47292 test case worked as expected
```

End of Test FSL-UT-PMIC-ADC-0010

40.8.2 PMIC Monitoring Tests

This test returns monitoring touch screen value of MC13783 ADC.

Test ID: FSL-UT-PMIC-ADC-0020

Type the following:

```
./pmic_testapp_adc.out -T MON
```

The system returns:

```
pmic_testapp_adc    0  INFO  :  Testing if 50232 test case is OK

Test monitoring ADC functions
pmic_testapp_adc    1  PASS  :  50232 test case worked as expected
```

End of Test FSL-UT-PMIC-ADC-0020

40.9 PMIC Power Management Tests

The test cases in Section 40.9, “PMIC Power Management Tests“” are available for i.MX27, i.MX31 and i.MX32 platform.

Use the “`pmic_testapp_power`” program to test the MC13783-specific version of this device driver.

Test ID: FSL-UT-PMIC-Power-MOD-0010

To run this test application, the test module `mxc_pmic_power_testmod.ko` must be inserted.

End of Test FSL-UT-PMIC-Power-MOD-0010

NOTE

After upgrade to kernel 2.6.24, the PMIC power driver is re-impelemented under the common regualtor driver architecture. So the test cases in this section are obsolete for i.MX31 3-Stack and i.MX27 3-Stack.

40.9.1 PMIC Regulator ON/OFF Tests

Test ID: FSL-UT-PMIC-Power-0010

This test checks the Regulator ON/OFF API function of the PMIC power.

1. Load this module:

```
modprobe mxc_pmic_power_testmod
```

2. For the PMIC ON/OFF test, type this line:

```
./pmic_testapp_power.out -T 1
```

The following result appears for a successful test:

```
Testing if REGULATOR ON is OK
REGULATOR 4 : ON
REGULATOR 5 : ON
REGULATOR 6 : ON
REGULATOR 7 : ON
REGULATOR 8 : ON
REGULATOR 9 : ON
REGULATOR 10 : ON
REGULATOR 11 : ON
REGULATOR 12 : ON
REGULATOR 13 : ON
REGULATOR 14 : ON
REGULATOR 15 : ON
REGULATOR 16 : ON
REGULATOR 17 : ON
REGULATOR 18 : ON
REGULATOR 19 : ON
REGULATOR 20 : ON
REGULATOR 21 : ON
REGULATOR 22 : ON
Test PASSED
Testing if REGULATOR OFF is OK
Switching OFF is restricted to limited of Regulator
as its' dangerous to switch-off regulators randomly.
REGULATOR 4 : OFF
REGULATOR 5 : OFF
REGULATOR 6 : OFF
REGULATOR 16 : OFF
REGULATOR 18 : OFF
Test PASSED
```

End of Test FSL-UT-PMIC-Power-0010

40.9.2 PMIC Switcher and Regulator Configuration Tests

Test ID: FSL-UT-PMIC-Power-0020

This test checks switches/Regulator Configuration API function of PMIC power.

NOTE

Make sure the configuration file `mc13783_power.cfg` is in the working directory. It is available in `<ltib_dir>/rpm/BUILD/misc/test/mxc_pmic_test/pmic_testapp_power/mc13783_power.cfg`.

For the PMIC Switcher/Regulator Configuration test, type this line:

```
./pmic_testapp_power.out -T 2
```

The following result appears for a successful test:

```
Testing if SWITCHES/REGULATOR configurations are OK
REGULATOR 0 : Test PASSED
REGULATOR 0 : Test PASSED
REGULATOR 1 : Test PASSED
REGULATOR 1 : Test PASSED
REGULATOR 2 : Test PASSED
REGULATOR 2 : Test PASSED
REGULATOR 3 : Test PASSED
REGULATOR 3 : Test PASSED
REGULATOR 4 : Test PASSED
REGULATOR 5 : Test PASSED
REGULATOR 6 : Test PASSED
REGULATOR 7 : Test PASSED
REGULATOR 8 : Test PASSED
REGULATOR 8 : Test PASSED
REGULATOR 9 : Test PASSED
REGULATOR 9 : Test PASSED
REGULATOR 10 : Test PASSED
REGULATOR 10 : Test PASSED
REGULATOR 11 : Test PASSED
REGULATOR 11 : Test PASSED
REGULATOR 12 : Test PASSED
REGULATOR 12 : Test PASSED
REGULATOR 13 : Test PASSED
REGULATOR 13 : Test PASSED
REGULATOR 14 : Test PASSED
REGULATOR 14 : Test PASSED
REGULATOR 15 : Test PASSED
REGULATOR 15 : Test PASSED
REGULATOR 16 : Test PASSED
REGULATOR 16 : Test PASSED
REGULATOR 17 : Test PASSED
REGULATOR 19 : Test PASSED
REGULATOR 19 : Test PASSED
REGULATOR 20 : Test PASSED
REGULATOR 20 : Test PASSED
REGULATOR 21 : Test PASSED
REGULATOR 21 : Test PASSED
REGULATOR 22 : Test PASSED
REGULATOR 22 : Test PASSED
TEST CASE PASSED
```

End of Test FSL-UT-PMIC-Power-0020

40.9.3 PMIC Miscellaneous Feature Tests

Test ID: FSL-UT-PMIC-Power-0030

This test checks regulator API function of MC13783 power.

For the PMIC Switcher test, type this line:

```
./pmic_testapp_power.out -T 3
```

The following result appears for a successful test:

```
Testing if Miscellaneous features are OK
Testing if Configuration of System Reset Buttons are OK
Test PASSED
Testing if Auto Reset is configured
Test PASSED
Testing if ESIM control voltage values are configured\
Test PASSED
Testing if Regen polarity is configured
Test PASSED
Testing if battery detect enable is configured
Test PASSED
Testing if AUTO_VBKUP2 enable is configured
Test PASSED
Testing if Power Control Configurations are OK
Test PASSED
TEST CASE PASSED
```

End of Test FSL-UT-PMIC-Power-0030

40.10 PMIC Connectivity Tests

These test applications can only be run on i.MX27 PDK.

40.11 PMIC Battery Tests

The test cases in Section 40.11, “PMIC Battery Tests“” are available for i.MX27, platform.

Use the “pmic_testapp_battery” program to test this module.

Test ID: FSL-UT-PMIC-0010

This is the auto test. The auto test executes pmic_testapp_battery -T 0 through 5, which completes the tests called for in sections 40.11.1 through 40.11.6.

Execute the auto test as follows:

```
./autorun-pmic.sh
```

The result should be "Exiting PASS"

End of Test FSL-UT-PMIC-0010

40.11.1 Charger Management Tests

This test checks charger management API function of PMIC battery.

```
Type : ./pmic_testapp_battery.out -T 0
```

If the result is correct you will see:

```
pmic_battery_testapp    0  INFO  :  Test main charger control function of PMIC Battery
pmic_battery_testapp    0  INFO  :  VOLTAGE = 0  CURRENT = 0
pmic_battery_testapp    0  INFO  :  VOLTAGE = 0  CURRENT = 1
.....
```

```
pmic_battery_testapp    0  INFO  :  VOLTAGE = 7  CURRENT = 6
pmic_battery_testapp    0  INFO  :  VOLTAGE = 7  CURRENT = 7
pmic_battery_testapp    0  INFO  :  Test disable charger control function of PMIC Battery
driver
pmic_battery_testapp    0  INFO  :  Test COIN cell charger control function of PMIC Battery
pmic_battery_testapp    0  INFO  :  VOLTAGE = 0
....
pmic_battery_testapp    0  INFO  :  VOLTAGE = 3
pmic_battery_testapp    0  INFO  :  Test disable charger control function of PMIC Battery
driver
pmic_battery_testapp    0  INFO  :  Test TRICKLE charger control function of PMIC Battery
pmic_battery_testapp    0  INFO  :  CURRENT = 0
......
pmic_battery_testapp    0  INFO  :  CURRENT = 7
pmic_battery_testapp    0  INFO  :  Test disable charger control function of PMIC Battery
driver
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_0 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_0 test case working as expected
```

40.11.2 EOL Comparator Tests

This test checks charger management API function of PMIC battery.

```
Type : ./pmic_testapp_battery.out -T 1
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test enable eol function of PMIC Battery
pmic_battery_testapp    0  INFO  :  THRESHOLD = 0
pmic_battery_testapp    0  INFO  :  THRESHOLD = 1
pmic_battery_testapp    0  INFO  :  THRESHOLD = 2
pmic_battery_testapp    0  INFO  :  THRESHOLD = 3
pmic_battery_testapp    0  INFO  :  Test disable eol function of PMIC Battery
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_1 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_1 test case working as expected
```

40.11.3 LED Management Tests

This test checks LED management API function of PMIC battery.

```
Type : ./pmic_testapp_battery.out -T 2
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test led control function of PMIC Battery
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_2 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_2 test case working as expected
```

40.11.4 Reverse Supply and Unregulated Modes Tests

This test checks Reverse Supply and Unregulated Modes API function of PMIC battery.

```
Type : ./pmic_testapp_battery.out -T 3
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test set reverse supply function of PMIC Battery
pmic_battery_testapp    0  INFO  :  Test set reverse supply function of PMIC Battery
pmic_battery_testapp    0  INFO  :  Test unregulated function of PMIC Battery enable
```

```
pmic_battery_testapp    0  INFO  :  Test unregulated function of PMIC Battery disable
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_3 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_3 test case working as expected
```

40.11.5 Set Out Control Tests

This test checks Set Out Control API function of PMIC battery.

```
Type : ./pmic_testapp_battery.out -T 4
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test set out control function of PMIC Battery in
CONTROL_BPFET_LOW
pmic_battery_testapp    0  INFO  :  Test set out control function of PMIC Battery in
CONTROL_BPFET_HIGH
pmic_battery_testapp    0  INFO  :  Test set out control function of PMIC Battery in
CONTROL_HARDWARE
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_4 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_4 test case working as expected
```

40.11.6 Set Over Voltage Threshold

This test checks Set Over Voltage Threshold API function of PMIC battery.

```
Type : ./pmic_testapp_battery.out -T 5
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test set threshold function of PMIC Battery
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_5 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_5 test case working as expected
```

40.11.7 Get Charger Current

Test ID: FSL-UT-PMIC-Battery-0010

This test checks Get Charger Current API function of PMIC battery.

```
Type : ./pmic_testapp_battery.out -T 6
```

If the result is correct you can see:

```
charger current : 1023.
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_6 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_6 test case working as expected
```

End of Test FSL-UT-PMIC-Battery-0010

40.12 Low-level Power Management Tests on i.MX27

Test ID: FSL-UT-PM-MX27-0010

Low-level PM driver is tested on the hardware. A test module called `mxc_pm_test.c` was written to test the kernel space API provided by the low-level PM driver. This test module provides IOCTL namely, MXCTEST_PM_LOWPOWER, which is used to test DOZE or SLEEP low-power modes.

To perform this test, JTAG must be disconnected from the hardware, and the kernel should be executed using the command-line option, “jtag = off”, which is the default option.

The unit test application and the test module are `called mxc_pm_test.out` and `mxc_pm_test.ko`, respectively, and can be found in the ROOTFS.

Table 40-1.

File	Location
mxc_pm_test.ko	/unit_tests/modules
mxc_pm_test.out	/unit_tests/mxc_pm_test

When executing low power mode tests, ensure that the keypad is connected, because only a key press will wake up the core from SLEEP low power mode.

For low power modes, three options are available: WAIT, DOZE, and STOP. WAIT and DOZE internally implement the DOZE mode of i.MX27 and STOP internally implement SLEEP mode of i.MX27.

For executing the test case, perform the follow steps:

```
insmod mxc_pm_test.ko
./mxc_pm_test.out
root@freescale /unit_tests/mxc_pm_test$ ./mxc_pm_test.out
```

========== TESTING Low-level PM DRIVER ==========

Note that some option #'s may be skipped depending on platform. So, enter the option number corresponding to each option correctly.

Enter any of the following options:

```
1. Test Integer Scaling
2. Test PLL Scaling
3. Test Int/PLL Scaling (choice decided by PM driver)
4. Test Low Power Modes
5. Select CKOH output
9. Infinite loop cycling through operating points.
4
1. WAIT mode
2. DOZE mode
3. STOP mode
4. DSM mode
Enter a valid choice:
3
```

A key press will wake up the core for choice 3 (STOP mode).

End of Test FSL-UT-PM-MX27-0010

40.13 LCDC Tests

The test cases in Section 40.13, “LCDC Tests“” are only available for i.MX27 platform.

Test ID: FSL-UT-LCDC-FB-0010

Framebuffer Tests:

- Saving an image from the framebuffer device:

```
# cat /dev/fb0 > framebuffer.bin
```

- Redirecting an image directly to the background framebuffer device:

```
# cat image.bin > /dev/fb0
```

End of Test FSL-UT-LCDC-FB-0010

40.14 CH7024 TV-Out Tests

The unit test application for the CH7024 TV-Out can be found in the `ROOTFS/unit_tests/mxc_fb_test` directory. The name of the application is `mxc_tvout_test.out`.

The test cases in Section 40.14, “CH7024 TV-Out Tests“ are available for i.MX27 platforms.

Test ID : FSL-UT-CH7024-0010

This test case switches the display from the LCD panel to the TV-Out mode. The «-m» parameter allows to choose either the TV-Out NTSC («n») mode or the TV-Out PAL («p») mode.

```
$ cd /unit_tests/mxc_fb_test/
$ ./mxc_tvout_test.out -m n
$ ./mxc_tvout_test.out -e y
```

Note that the frame buffer dimensions are changed from 480x640 to 640x480, hence the application needs to be restarted to take the new resolution into account.

End of Test FSL-UT-CH7024-0010

Test ID : FSL-UT-CH7024-0020

This test case uses `/cat/proc` commands to display information about the CH7024 driver.

```
$ cat /proc/driver/ch7024/attribute
TV Format : NTSC
Brightness : 144
Hue       : 64
Saturation : 64
Contrast  : 64
Sharpness : 4
```

End of Test FSL-UT-CH7024-0020

Test ID : FSL-UT-CH7024-0030

This test cases shows how to change the CH7024 driver settings using `/cat/proc` commands.

```
$ echo «Brightness=200» > /proc/driver/ch7024/attribute
$ echo «Contrast=100» > /proc/driver/ch7024/attribute
$ echo «Hue=100» > /proc/driver/ch7024/attribute
$ echo «Sharpness=2» > /proc/driver/ch7024/attribute
$ echo «Saturation=120» > /proc/driver/ch7024/attribute
```

You should see the image change accordingly.

End of Test FSL-UT-CH7024-030

Test ID : FSL-UT-CH7024-0040

This test case switches back the display from TV-Out to the LCD panel.

```
$ ./mxc_tvout_test.out -e n
```

End of Test FSL-UT-CH7024-0040

40.15 OmniVision Camera Tests

The test cases in Section 40.15, “OmniVision Camera Tests“ are available for i.MX27, i.MX31, i.MX32, and i.MX35 platforms.

To test the camera, use the following command to install the kernel modules of the camera driver, if it was not loaded automatically on boot up:

```
insmod ipu_prp_enc.ko
insmod ipu_prp_vf_sdc.ko
insmod ipu_prp_vf_sdc_bg.ko
insmod ipu_still.ko
insmod ov2640_camera.ko
insmod mxc_v4l2_capture.ko
```

Then run the following test cases to test it.

Test ID: FSL-UT-V4L2-capture-0010

```
# mxc_v4l2_capture.out -w 352 -h 288 -r 0 -c 50 -fr 30 test.yuv
```

Capture the camera and store the 50 frames of YUV420 (QQVGA size) to the test.yuv file, and then set the frame rate to 30 fps. For information about usage, see mxc_v4l2_capture.out -help.

End of Test FSL-UT-V4L2-capture-0010

Test ID: FSL-UT-V4L2-overlay-sdc-0010

```
# mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 160 -oh 160 -ot 20 -ol 20 -r 0 -t
50 -d 0 -fg -fr 30
```

Direct preview the camera to SDC foreground, and set the frame rate to 30 fps. The window of interest is 640 X 480, with a starting offset of (0,0). The preview size is 160 X 160 with a starting offset of (20,20). For more information, see mxc_v4l2_overlay.out -help .

End of Test FSL-UT-V4L2-overlay-sdc-0010

Test ID: FSL-UT-V4L2-overlay-sdc-0020

```
# mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 160 -oh 160 -ot 20 -ol 20 -r 4 -t
50 -d 0 -fr 30
```

Direct preview (90 degree rotation) the camera to SDC background, and set the frame rate to 30 fps.

End of Test FSL-UT-V4L2-overlay-sdc-0020

40.16 i.MX27 VPU Tests

The unit test application for VPU can be found in the `ROOTFS/unit_tests/mxc_vpu_test` directory. The name of the application is `mxc_vpu_test.out`.

Executing the application displays the following help:

```
./mxc_vpu_test.out
Usage: ./test -D "<decode options>" -E "<encode options>" -L "<loopback options>
" -C <config file> -H display this help

decode options
   -i <input file> Read input from file
        input file is mandatory for decode
   -o <output file> Write output to file
        If no output is specified, default is LCD
   -f <format> 0 - MPEG4, 1 - H.263, 2 - H.264, 3 - VC1
        If no format specified, default is 0 (MPEG4)
   -c <count> Number of frames to decode
   -d <deblocking> Enable deblock - 1. Not for MX27
        default deblock is disabled (0).
   -r <rotation angle> 0, 90, 180, 270
        default rotation is disabled (0)
   -m <mirror direction> 0, 1, 2, 3
        default no mirroring (0)

encode options
   -i <input file> Read input from file (yuv)
        If no input file specified, default is camera
   -o <output file> Write output to file
        If no output is specified, def files are created
   -f <format> 0 - MPEG4, 1 - H.263, 2 - H.264, 3 - VC1
        If no format specified, default is 0 (MPEG4)
   -c <count> Number of frames to encode
   -r <rotation angle> 0, 90, 180, 270
        default rotation is disabled (0)
   -m <mirror direction> 0, 1, 2, 3
        default no mirroring (0)
   -w <width> capture image width
        default is 176.
```

```
   -h <height>capture image height
        default is 144
   -b <bitrate in kbps>
        default is auto (0)
   -g <gop size>
        default is 0

loopback options
   -f <format> 0 - MPEG4, 1 - H.263, 2 - H.264, 3 - VC1
        If no format specified, default is 0 (MPEG4)
   -w <width> capture image width
        default is 176.
   -h <height>capture image height
        default is 144

config file - Use config file for specifying options


Some examples:

For H.264 decode
         ./mxc_vpu_test.out -D "-i /usr/vectors/h264/vga.264 -f 2"

For MPEG4 decode with rotation
         ./mxc_vpu_test.out -D "-i /usr/vectors/mp4/d1.mpeg4 -f 0 -r 90"

Using the config file
         ./mxc_vpu_test.out -C config_dec

For H.264 encode
         ./mxc_vpu_test.out -E "-o out.264 -w 240 -h 320 -f 2"

Doing a simultaneous encode and decode
         ./mxc_vpu_test.out -E "-o enc.264 -w 176 -h 144 -f 0" -D "-i /vectors/vga.264 -f 2"

Using the loopback test (The enocoded image from the camera is decoded immediately)
         ./mxc_vpu_test.out -L "-f 2 -w 240 -h 320"
```

40.17 ARC USB Tests

40.17.1 USB host

40.17.1.1 PTP

Test ID: FSL-UT-USB-ARC-HOST-PTP-0010

gPhoto2 could be used for PTP tests. You could download gPhoto2 software from http://www.gphoto.org/

gPhoto2 is a free, redistributable, ready to use set of digital camera software applications for Unix-like systems, written by a whole team of dedicated volunteers around the world. It supports more than 900 cameras. gPhoto2 runs on a large range of UNIX-like operating system, including Linux, FreeBSD, and NetBSD. gPhoto is provided by major Linux distributions like Debian GNU/Linux, Ubuntu, Gentoo, Fedora, SUSE Linux, and Mandriva.

End of Test FSL-UT-USB-ARC-HOST-PTP-0010

40.17.1.2 MSC

Test ID: FSL-UT-USB-ARC-HOST-MSC-0020

The steps are:

1. `modprobe isp1504-arc`
2. `modprobe ehci-hcd`
3. `mkdir /mnt/udisk`
4. Plug in a U-disk
5. `cat /proc/partitions`, for example:
   major minor #blocks name
   31 0 256 mtdblock0
   8 0 127744 sdb
6. Find partitions with a major number of 8; in this example, the sdb is U-disk partition.

```
mount -t vfat /dev/sdb1 /mnt/udisk
OR
mount -t vfat /dev/sda1 /mnt/udisk
```

If this function is not used, unload the following drivers:

- `umount /mnt/udisk`
- `rmmod ehci-hcd`
- `rmmod isp1504-arc`

End of Test FSL-UT-USB-ARC-HOST-MSC-0020

40.17.1.3 HID

Material: USB mouse or USB keyboard.

Test ID: FSL-UT-USB-ARC-HOST-HID-0030

The steps are:

1. `modprobe isp1504-arc`
2. `modprobe ehci-hcd`
3. `modprobe usbhid`
   Insert USB mouse or USB keyboard.
4. `/evtest /dev/input/event2`

If this function is not used, unload the following drivers.

- `rmmod usbhid`
- `rmmod ehci-hcd`
- `rmmod isp1504-arc`

Notes:

1. For HID build as module:
   USB support --> USB Human Interface Device (full HID) support. Select it as a module.

```
CONFIG_USB_HID=m
```

USB support --> HID input layer support. Select as built in. `CONFIG_USB_HIDINPUT = y`.

```
modprobe isp1504-arc
modprobe ehci-hcd
insmod usbhid.ko
```

2. `The evtest` tool is a popular Linux open source tool, and can be located on the Internet. It is also included in LTIB package `"input-utils"`.

End of Test FSL-UT-USB-ARC-HOST-HID-0030

40.17.2 Peripheral Mode

40.17.2.1 MSC

Test ID: FSL-UT-USB-ARC-Peripheral-MSC-0010

To use MMC/SD card as a U-Disk:

1. Plug a MMC/SD card into the board.
2. Boot the board.
3. Load MMC/SD modules:

```
insmod mx_sdhci
OR
insmod mxc_mmc
```

4. Load USB modules:

```
modprobe arcotg_udc
modprobe g_file_storage file="/dev/mmcblk0"
```

5. Connect the board to the PC with a USB cable.
6. The PC should recognize the U-disk.

   If this is the first time it has been used, it will have to be formatted.

   On a PC, enter Computer Management tools, storage-> Disk Management. Following the PC tutor, get a raw disk, create a partition on it, and format it with FAT or FAT32.
7. Files can now be transferred between the PC and the board.

To use a partition of the hard disk as a U disk:

1. Get a “raw” hard disk, and connect it to the ATA interface of board.
2. `dd if=/dev/zero of=/dev/hda bs=1M count=100`
3. Reboot this board.
4. `modprobe arcotg_udc`
5. `modprobe g_file_storage file="/dev/hda"` if IDE interface is used.
   or `modprobe g_file_storage file="/dev/sda"` if LibATA interface is used.
6. Use the PC to create a partition table, and format every partition.

   On a PC using Windows, open "Computer Management", enter storage -> Disk Management

   Enter Initialize and Convert Disk Wizard. Everything should be set to the default. Create a partition table using the PC with Windows. After creating a partition table, input the "sync" command in the Linux console.

7. Reboot this board, and look at the partition table (`fdisk -l`)

```
root@10 ~$ fdisk -l
Disk /dev/hda: 20.0 GB, 20003880960 bytes
```

255 heads, 63 sectors/track, 2432 cylinders

Units = cylinders of 16065 * 512 = 8225280 bytes

```
   Device Boot      Start         End      Blocks   Id  System
/dev/hda1               1         127     1020096    b  W95 FAT32
/dev/hda2             128        2431    18506880    f  W95 Ext'd (LBA)
/dev/hda5             128         254     1020096    6  FAT16
/dev/hda6             255         636     3068383+   b  W95 FAT32
```

The above steps need to be done only once.

After that, perform the following steps when using ATA as U-Disk:

1. `modprobe arcotg_udc`
2. Set the hard disk partition as a USB disk.

   ```
   modprobe g_file_storage file="/dev/hda"
   ```
3. Files may now be transferred between the PC and the board.
4. After finishing the file transfer, it can mounted:
5. `mount -t vfat /dev/hda6 /mnt/test`
6. `ls /mnt/test`, everything here.

NOTE

- Do not use the Linux format command (for example, fdisk) to create a hard disk partition table. PC Windows must be used to create it if one partition is to be used as a U disk.
- Do not use this partition on the PC and the board at the same time.
- Use device "`/dev/sda`" instead of "`/dev/hda`" if the LibATA interface is used.
- If this function is not used, unload the following drivers

  ```
  umount /mnt/test
  rmmod g_file_storage
  rmmod arcotg_udc
  ```

End of Test FSL-UT-USB-ARC-Peripheral-MSC-0010

40.17.2.2 Ethernet

Test ID: FSL-UT-USB-ARC-Peripheral-ETH-0020

Steps:

1. `modprobe arcotg_udc`
2. `modprobe g_ether`
3. `ifconfig usb0`
4. `ifconfig usb0  192.168.0.2 up`

When first using the USB Ethernet, and the PC OS is Windows XP, a driver must be installed. Use the BSP file: `linux\Documentation\usb\linux.inf` to install the driver.

5. On the PC (if running Windows), find a device named Linux USB Ethernet/RNDIS Gadget
6. Configure it with a fixed IP.
7. Press the right button on the mouse, and click on TCP/IP ' Properties '
8. Use the following IP address and settings:

   192.168.0.1

   255.255.255.0

   192.168.0.1
9. Save these settings.
10. From the PC, ping 192.168.0.2.

If this function is not used

```
ifconfig usb0 down
```

If this function is not used, unload the following drivers

- `rmmod g_ether`
- `rmmod arcotg_udc`

End of Test FSL-UT-USB-ARC-Peripheral-ETH-0020

40.17.2.3 ACM

Test ID: FSL-UT-USB-ARC-Gadget-ACM-0030

Testing the serial gadget requires that the windows ACM driver be installed. This requires gserial.inf and usbser.sys. gserial.inf can be obtained from `Documentation/usb/gadget_serial.txt`. usbser.sys can be obtained using:

```
Expand "C:\WINDOWS\Driver Cache\i386\driver.cab" -F:usbser.sys .
```

Copy gserial.inf and usbser.sys to a single directory.

1. `modprobe arcotg_udc`

   ARC USBOTG Device Controller driver version 1 August 2005 init
   ARC USBOTG h/w ID=0x5 revision=0x40
2. `mknod /dev/ttygs c 127 0`
3. `modprobe g_serial use_acm=1`

   ```
   gs_bind: Gadget Serial v2.2 bound
   arcotg_udc: gadget arc_udc bound to driver g_serial
   gs_module_init: Gadget Serial v2.2 loaded
   ```
4. Connect the board to the PC with a USB cable.

   `gs_disconnect:` Gadget Serial disconnected
   `gs_disconnect:` Gadget Serial disconnected
   `gs_set_config:` Gadget Serial configured, high speed CDC-ACM config
5. Windows will recognize the serial gadget. Choose to install a driver from a specific location and select gserial.inf. This will install usbser.sys as the ACM driver in windows.

6. On a Windows host configure a new HyperTerminal session to use the COM port assigned to Gadget Serial. The "Port Settings" will be set automatically when HyperTerminal connects to the gadget serial device, so you can leave them set to the default values--these settings mostly do not matter.
7. `echo "hi" > /dev/ttygs`
   String "hi" should be displayed in Gadget Serial terminal
8. `cat /dev/ttygs`

Anything you type on the Gadget Serial terminal should appear in serial port terminal window. For further details refer to linux documentation for serial gadget in Documentation/usb/gadget_serial.txt.

End of Test FSL-UT-USB-ARC-Gadget-ACM-0030

40.18 ALSA Tests

Accessing or testing ALSA with Freescale BSP. The test cases in Section 40.18, "ALSA Tests"" are available for i.MX27, i.MX31, and i.MX32 platforms.

40.18.1 ALSA Native Mode

Make sure the utilities `aplay`, `arecord`, and `alsamixer` are present in `/usr/bin` of ROOTFS.

`libasound.so.2.0.0` exists in `usr/bin` and a softlink `libasound.so.2` is present in the same directory pointing to `libasound.so.2.0.0`

```
usr/share/alsa directory exists
```

It is populated with all configuration information.

All the above would be present in the ROOTFS and user need not do anything on his part.

40.18.2 Playback on Stereo DAC:

Test ID: FSL-UT-ALSA-0010

```
aplay –N –M file.wav
```

- -N indicates non-block mode for device open
- -M Indicates MMAP mode. (Playback works even without –M but performance is good with –M especially for high sample rates)
- file.wav can be any mono or stereo file from 8k through 96k sample rate. Supported rates are 8000, 11025, 16000, 22050, 24000, 32000, 44100, 48000 and 96000.

Sound Files can be located in the `test/mxc_ssi_test/sound_samples` directory.

End of Test FSL-UT-ALSA-0010

40.18.3 Recording

Test ID: FSL-UT-ALSA-0030

```
arecord –r 8000 –c 1 –f S16_LE –N –M –d 20 file.wav
```

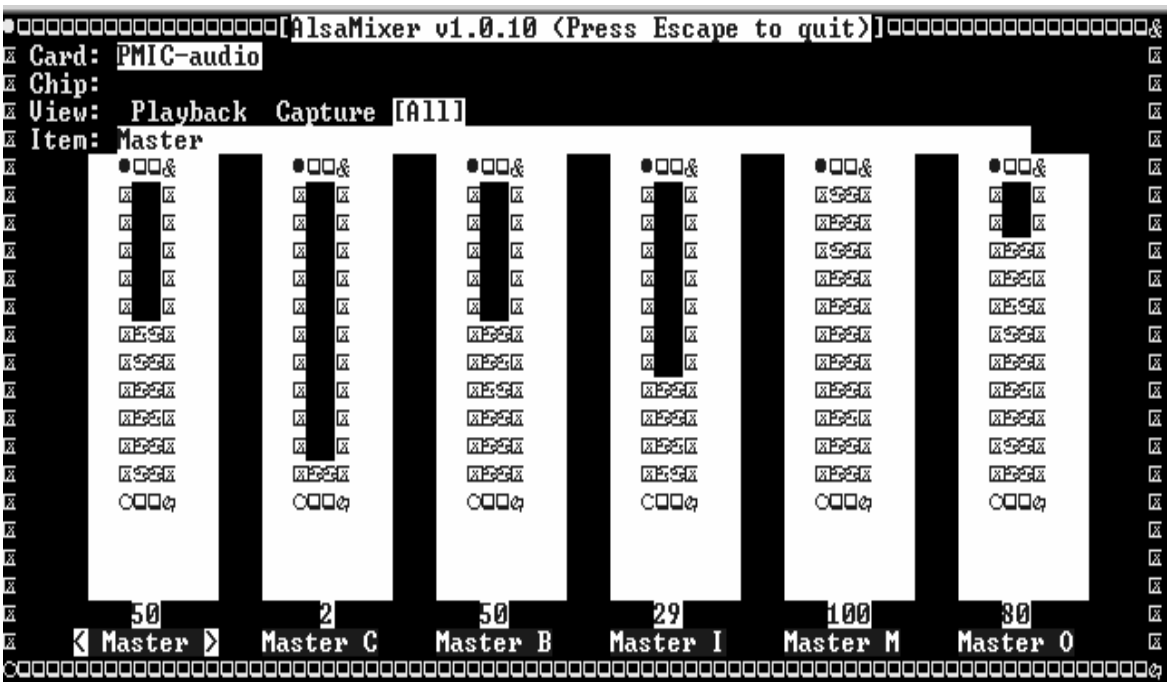
- -r – sample rare can be either 8000 or 16000
- -c – 1 – `Num` of channels and has to be 1 as only mono recording is supported
- -f is the bit format. Has to be Signed 16 bit Little Endian
- -N and –M same as in `aplay`
- -d – duration of recording in seconds
- File.wav is the one that would be created with recorded voice

End of Test FSL-UT-ALSA-0030

40.18.4 Mixer

Test ID: FSL-UT-ALSA-0040

"`alsamixer -V all`" command displays all the controls on the console. (It uses an `ncurses` library that needs to be present). The GUI window looks as follows:



Use TAB to switch among playback, capture, and All menu. Use up/down arrows to increase or decrease values and left/right arrows to navigate among controls.

Starting from left, navigation is

- Master Playback volume ranges from 0 to 100 and is internally mapped to PMIC dB gains.
- Master Capture volume (displayed as Master C). This ranges from 0 through 100. (Capture only control)

- Master B or master playback balance. This ranges from 0 through 100. A value of 50 indicates both L and R equal. 0 attenuates one channel to the maximum and 100 attenuates the other. (Playback only control)
- Master I or master input device selection. This control ranges from 0 through 7 (three bits with LSB representing handset `Mic`, bit 1 representing headset `Mic` and bit 2 representing Line IN). A value of 0 in the bit position resets the corresponding device and 1 set the same. For example, a value of 5 that is, 101 in binary sets handset MIC and Line IN. It is scaled to 0 to 100 and displayed but there are exactly 7 steps (corresponding to 000 through 111 in binary) (Capture only control)
- Master M or master mono adder configuration (Playback only control). This ranges in value from 0 through 3 (Again scaled to 0 to 100 range with 3 steps). This represents L and R separate, L and R added, R 180 degree phase shifted with respect to L and both added and R phase shifted with respect to L.
- Master O or master output device selection. This control ranges from 0 through 15 (four bits with LSB representing earphone, bit 1 representing hands free, bit 2 representing Headset and bit 3 representing Lineout). A value of 0 in the bit position resets the corresponding device and 1 set the same. For example, a value of 5 that is, 0101 in binary sets headset and earphone. It is scaled to 0 to 100 and displayed but there are exactly 15 steps (corresponding to 000 through 1111 in binary)
- A value of 15 or 1111 sets all output devices and 0 resets everything.(Playback only control).

End of Test FSL-UT-ALSA-0040

40.18.4.1 Testing Audio Record Capabilities

The goal is to test recording sound from Microphone and Line Inputs, both Mono and Stereo.

The recording capability uses external memory, not IRAM, for the audio record buffers. However, recording can work with or without the audio playback buffers located in IRAM.

40.18.4.1.1 Stereo Record Testing

1. Set the record level for Microphone
   a) alsamixer -V all
   b) Select Item: Capture and increase to at least 75
   c) Select Item: Left Input Mixer +20db and hit M to toggle it On
   d) Select Item: Right Input Mixer +20db and hit M to toggle it On
   e) Hit ESC to exit
2. Recording sound from Microphone. Speak into MIC while running the following:
   a) arecord -d 5 -c 2 -f S16_LE -r 8000 /tmp/mic8k.wav
   b) arecord -d 5 -c 2 -f S16_LE -r 16000 /tmp/mic16k.wav
   c) arecord -d 5 -c 2 -f S16_LE -r 24000 /tmp/mic24k.wav
   d) arecord -d 5 -c 2 -f S16_LE -r 32000 /tmp/mic32k.wav
   e) arecord -d 5 -c 2 -f S16_LE -r 48000 /tmp/mic48k.wav
   f) arecord -d 5 -c 2 -f S16_LE -r 11025 /tmp/mic11k.wav

g) arecord -d 5 -c 2 -f S16_LE -r 22050 /tmp/mic22k.wav
h) arecord -d 5 -c 2 -f S16_LE -r 44100 /tmp/mic44k.wav
i) arecord -d 5 -c 2 -f S16_LE -r 96000 /tmp/mic96k.wav
j) arecord -d 5 -c 2 -f S16_LE -r 64000 /tmp/mic64k.wav

3. Set the record level for Line Inputs
   a) alsamixer -V all
   b) Select Item: Capture and increase to 75
   c) Select Item: Left Capture Mixer AUX Capture and increase to 75
   d) Select Item: Right Capture Mixer AUX Capture and increase to 75
   e) Hit ESC to exit
4. Recording sound from line inputs. Drive sound into stereo line inputs to codec, while executing the following commands. Ensure that the signal level at the codec inputs is within the specifications of the codec.
   a) arecord -d 5 -c 2 -f S16_LE -r 8000 /tmp/lin8k.wav
   b) arecord -d 5 -c 2 -f S16_LE -r 16000 /tmp/lin16k.wav
   c) arecord -d 5 -c 2 -f S16_LE -r 24000 /tmp/lin24k.wav
   d) arecord -d 5 -c 2 -f S16_LE -r 32000 /tmp/lin32k.wav
   e) arecord -d 5 -c 2 -f S16_LE -r 48000 /tmp/lin48k.wav
   f) arecord -d 5 -c 2 -f S16_LE -r 11025 /tmp/lin11k.wav
   g) arecord -d 5 -c 2 -f S16_LE -r 22050 /tmp/lin22k.wav
   h) arecord -d 5 -c 2 -f S16_LE -r 44100 /tmp/lin44k.wav
   i) arecord -d 5 -c 2 -f S16_LE -r 96000 /tmp/lin96k.wav
   j) arecord -d 5 -c 2 -f S16_LE -r 64000 /tmp/lin64k.wav
5. Playback the files using aplay filename.wav

40.18.4.1.2 Mono Record Testing

1. Set the record level for Microphone and mute the Left Headphone
   a) alsamixer -V all
   b) Select Item: Capture and increase to at least 75
   c) Select Item: Left Input Mixer +20db and hit M to toggle it On
   d) Select Item: Right Input Mixer +20db and hit M to toggle it On
   e) Select Item: Left Playback Mixer Playback Sw and hit M to toggle it Off to mute Left DAC input to the Left Playback Mixer.
   f) Select Item: Left Playback Mixer Right Playback Sw and hit M to toggle it On to unmute the Right DAC input to Left Playback Mixer.
   g) Hit ESC to exit
2. Recording sound from Microphone. Speak into MIC while running the following:
   a) arecord -d 5 -c 1 -f S16_LE -r 8000 /tmp/mic8kM.wav

b) arecord -d 5 -c 1 -f S16_LE -r 16000 /tmp/mic16kM.wav
c) arecord -d 5 -c 1 -f S16_LE -r 24000 /tmp/mic24kM.wav
d) arecord -d 5 -c 1 -f S16_LE -r 32000 /tmp/mic32kM.wav
e) arecord -d 5 -c 1 -f S16_LE -r 48000 /tmp/mic48kM.wav
f) arecord -d 5 -c 1 -f S16_LE -r 11025 /tmp/mic11kM.wav
g) arecord -d 5 -c 1 -f S16_LE -r 22050 /tmp/mic22kM.wav
h) arecord -d 5 -c 1 -f S16_LE -r 44100 /tmp/mic44kM.wav
i) arecord -d 5 -c 1 -f S16_LE -r 96000 /tmp/mic96kM.wav
j) arecord -d 5 -c 1 -f S16_LE -r 64000 /tmp/mic64kM.wav

3. Set the record level for Line Inputs and mute the Left Headphone
   a) alsamixer -V all
   b) Select Item: Capture and increase to 75
   c) Select Item: Left Capture Mixer AUX Capture and increase to 75
   d) Select Item: Right Capture Mixer AUX Capture and increase to 75
   e) Select Item: Left Playback Mixer Playback Sw and hit M to toggle it Off to mute Left DAC input to the Left Playback Mixer.
   f) Select Item: Left Playback Mixer Right Playback Sw and hit M to toggle it On to unmute the Right DAC input to Left Playback Mixer.
   g) Hit ESC to exit
4. Recording sound from line inputs. Drive sound into right line input to codec, while executing the following commands. Ensure that the signal level at the codec inputs is within the specifications of the codec.
   a) arecord -d 5 -c 1 -f S16_LE -r 8000 /tmp/lin8kM.wav
   b) arecord -d 5 -c 1 -f S16_LE -r 16000 /tmp/lin16kM.wav
   c) arecord -d 5 -c 1 -f S16_LE -r 24000 /tmp/lin24kM.wav
   d) arecord -d 5 -c 1 -f S16_LE -r 32000 /tmp/lin32kM.wav
   e) arecord -d 5 -c 1 -f S16_LE -r 48000 /tmp/lin48kM.wav
   f) arecord -d 5 -c 1 -f S16_LE -r 11025 /tmp/lin11kM.wav
   g) arecord -d 5 -c 1 -f S16_LE -r 22050 /tmp/lin22kM.wav
   h) arecord -d 5 -c 1 -f S16_LE -r 44100 /tmp/lin44kM.wav
   i) arecord -d 5 -c 1 -f S16_LE -r 96000 /tmp/lin96kM.wav
   j) arecord -d 5 -c 1 -f S16_LE -r 64000 /tmp/lin64kM.wav
5. Playback the files using aplay filename.wav
6. To restore ability to hear the left channel headphone for stereo testing:
   a) alsamixer -V all
   b) Select Item: Left Playback Mixer Playback Sw and hit M to toggle it On to unmute Left DAC input to the Left Playback Mixer.

c) Select Item: Left Playback Mixer Right Playback Sw and hit M to toggle it Off to mute the Right DAC input to the Left Playback Mixer.

d) Hit ESC to exit

40.18.5 Audio Loop Back

Test ID: FSL-UT-ALSA-0050

```
arecord -c 1 -f S16_LE -r 8000/16000 | aplay -D hw:0,1
```

This command configures Voice Codec for recording and loops back the recorded samples onto the Voice Codec for playback. The sample rate can be either 8000 or 16000.

End of Test FSL-UT-ALSA-0050

40.18.6 OSS Emulation Mode

40.18.6.1 Playback on ST-DAC

Test ID: FSL-UT-OSS-Emulation-0010

```
/dev/sound/dsp file represents ST-DAC playback.
```

Use check_audio.out with the command (in `/unit_tests/mxc_sound_test`)

```
./check_audio.out 1 file.wav
```

1 as the first argument opens ST-DAC

End of Test FSL-UT-OSS-Emulation-0010

40.19 AUDMUX Test

ALSA test applications either in the native mode or in the OSS emulation mode should presently suffice to verify the AUDMUX driver APIs.

40.20 SSI Test

ALSA test applications either in the native mode or in the OSS emulation mode should presently suffice to verify the SSI driver APIs.

Note: The AUDIO test tests the SSI driver.

40.21 NAND Tests

Test ID: FSL-UT-NAND-MTD-0010

After the system boots up with the NAND MTD driver, it may be tested as follows:

JFFS2:

1. Find out the available partition(s) in the running Linux system. The above partition information comprises both NOR and NAND device. Typically partition on NAND device starts with the name “IPL-SPL”, so user can opt to test partition after this. For example:

```
root@freescale ~$ cat /proc/mtd
dev:    size   erasesize  name
mtd0: 00040000 00020000 "RedBoot"
mtd1: 01900000 00020000 "rootfs"
mtd2: 0001f000 00008000 "FIS directory"
mtd3: 00001000 00008000 "RedBoot config"
mtd4: 00020000 00004000 "IPL-SPL"
mtd5: 00400000 00004000 "nand.kernel"
mtd6: 01600000 00004000 "nand.rootfs"
mtd7: 005e0000 00004000 "nand.userfs"
```

2. Erase one of the NAND Flash partitions using `flash_eraseall.(X-> Partition number)`

```
$ flash_eraseall /dev/mtd/X
```

3. Create a raw file containing a `jffs2` filesystem

```
 $ mkfs.jffs2 -n -d /etc -o etc.jffs2 -e <erase_size> -s <page_size>
```

4. Write the Jffs2 image on to the NAND raw device

```
$ nandwrite -p /dev/mtd/X etc.jffs2
```

5. Mount the NAND partition and read the files

```
$ mkdir -p /tmp/mtdX
$ mount -t jffs2 /dev/mtdblockX /tmp/mtdX
$ cd /tmp/mtdX
$ find .    # list all the files
$ find . -type f -exec cat {} \; >/dev/null  # read all the files into /dev/null
$ ....#other file operations like copy from/to a NAND partition, remove..etc
```

In addition this module is tested by booting Linux with jffs2 root filesystem. In addition, for read/write test, mount a JFFS2 file system on a valid MTD partition and then do some file operations, such as copy and remove. Those operations should be successful. The files copied into the JFFS2 mount directory exist even after shutting down and restarting the platform.

NOTE

For booting Linux with jffs2 as the root filesystem the command line argument should be changed. If the jffs2 root filesystem is present on mtdblock 2 then the following command line should be used

`exec -c "noinitrd console=ttymxc0,115200 root=/dev/mtdblock2 rw rootfstype=jffs2 ip=none".`

End of Test FSL-UT-NAND-MTD-0010

40.22 Keypad Tests

To Run the Keypad Test Code:

After enabling the keypad test through "`make menuconfig`" and then doing "`make`" as described in Section 40.1, "Enabling the Unit Tests", the keypad test script is located in `<ltib_dir>/rpm/BUILD/misc/test/mxc_keyb_test` directory. It can be downloaded using the methods described in Section 41.1, "Downloading a File"".

Assuming that the test application is "`mxc_keyb_test.out`", the following are the steps to run it after being downloaded onto the platform:

1. `# chmod 755 *`

2. # ./mxc_keyb_test.out 2 2 8
3. When it says "Please press keys on the keypad:", press any 5 keys on the keypad within 5 seconds.

   You should see:

```
Key Press is LEFT
Key Release is LEFT
```

40.23 FEC Tests

The test cases in Section 40.23, “FEC Tests“ are available for i.MX27 and i.MX35 platforms.

Test ID: FSL-UT-FEC-0010

The following items are unit test cases required to test for proper functionality of the Ethernet driver for the Linux kernel. Following tests can be carried out using test application that is, client server programs to send or receive test packet.

- ping OK
- ftp OK (busybox 1.01)
- dhcp OK
- telnet OK
- telnetd OK
- nfs OK

For more information about the material covered in this chapter, see related Ethernet Transceiver data sheet. Also refer the <ltib_dir>/rpm/BUILD/linux_2.6.22/drivers/net/fec.c source file.

End of Test FSL-UT-FEC-0010

40.24 SMSC LAN9217 Ethernet Tests

Test ID: FSL-UT-ETHERNET-0010

The following items are unit test cases required to test for proper functionality of Ethernet driver for the Linux kernel. Following tests can be carried out using test application; that is, client server programs to send or receive test packet.

- ping OK
- ftp OK (busybox 1.01)
- dhcp OK
- telnet OK
- telnetd OK
- nfs OK

For more information about the material covered in this chapter, see SMSC LAN9217 – Ethernet Controller Data Sheet.

End of Test FSL-UT-ETHERNET-0010

40.25 I²C Test

Use the I²C device tests, such as “Camera” to verify the functionality of the MXC I²C driver.

40.26 CSPI Tests

The source code for the unit test applications (only for loop back mode) is available at the following location:

```
<ltib_dir>/rpm/BUILD/misc/test/mxc_spi_test/
```

The built unit test application for SPI tests is

```
/unit_tests/mxc_spi_test1.out
```

The built unit test modules for SPI test is the following:

```
/unit_tests/modules/mxc_spi_testmod.ko
```

This test send up to 32 bytes to a specific SPI device. SPI writes data received data from the user into Tx FIFO and waits for the data in the Rx FIFO. Once the data is ready in the Rx FIFO, it is read and sent to user. Test is considered successful if data received matches with the data sent.

NOTE

Because this test is intended to test the SPI device, it is always configured in loop back mode, since it is dangerous to send random data to SPI slave devices. This requires the kernel image to be rebuilt with `CONFIG_SPI_MXC_TEST_LOOPBACK` (Refer Section 29.5, “Configuration“”) enabled in the `menuconfig`. Also the test module `mxc_spi_testmod.ko` must be inserted before testing.

To run the SPI test:

```
Options: ./mxc_spi_test1.out <spi_no> <nb_bytes> <value>
<spi_no> - CSPI Module number in [0, 1, 2]
<nb_bytes> - No. of bytes: [1-32]
<value> - Actual value to be sent
```

Example:

```
./mxc_spi_test/mxc_spi_test1.out 0 9 FreeScale
Execute data transfer test: 0 9 FreeScale
Data sent : FreeScale
Data received : FreeScale
Test PASSED.
```

40.27 MMC/SD/SDIO Test

Before performing tests, load the module below:

For SDHC, load these modules if CONFIG_MMC_MXC is configured as “M” before tests:

```
insmod mxc_mmc
```

Test ID: FSL-UT-MMC-0010

From the `/unit_tests/` directory execute the auto test.

```
./autorun-mmc_sh
```

The result should be "Exiting PASS".

End of Test FSL-UT_MMC-0010

- Test for MMC card insertion event detection.
- Test for MMC card removal event detection.

Block Read/Write Test:

'`dd`' is the command for reading/writing blocks. Note this command can corrupt the partitions and filesystem on MMC card.

Examples:

Test ID: FSL-UT-MMC-0020

To clear the first 5 KB of the card:

```
$dd if=/dev/zero of=/dev/mmcblk0 bs=1024 count=5
```

The response should be

- 5+0 records in
- 5+0 records out

End of Test FSL-UT-MMC-0020

Test ID: FSL-UT-MMC-0030

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

```
$dd if=file_name of=/dev/mmcblk0
```

End of Test FSL-UT-MMC-0030

Test ID: FSL-UT-MMC-0040

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

```
$dd if=/dev/mmcblk0 of=output_file bs=1024 count=1
```

End of Test FSL-UT-MMC-0040

Test ID: FSL-UT-MMC-0050

File System Test:

For the filesystem tests select appropriate filesystem types and partition types when configuring the kernel.

Insert an MMC or SD card. For an SD card, the following type of information is displayed:

```
mmcblk0: mmc0:cffc SD512 495488KiB
mmcblk0: p1
```

For an MMC card, the following type of information will be displayed:

```
mmcblk0: mmc:001 000000 1003520KiB
mmcblk0:p1
```

The following command can be used to find out the capacities of the card.

If the card is pre-formatted, this command shows the size of the card, partitions and filesystem type:

```
$fdisk -l /dev/mmcblk0
```

If the card is not formatted:

— Create the partitions on the card using the following command:

```
$fdisk /dev/mmcblk0
```

— After the partition the created files resemble the following:

```
/dev/mmcblk0p[1-4]
```

End of Test FSL-UT-MMC-0050

Test ID: FSL-UT-MMC-0060

Format the card using mkfs.minix, mkfs.ext2 or mkfs.ext3 depending on the filesystem:

```
$mkfs.ext2 /dev/mmcblk0p1
$mkfs.ext3 /dev/mmcblk0p1
$mkfs.minix /dev/mmcblk0p1 no_blocks
```

End of Test FSL-UT-MMC-0060

Test ID: FSL-UT-MMC-0070

Mount the file system

```
$mkdir /mnt/mmc_part1
$mount -t ext2 /dev/mmcblk0p1 /mnt/mmc_part1 or
$mount -t ext3 /dev/mmcblk0p1 /mnt/mmc_part1
```

After mounting, file/directory operations can be performed in `/mnt/mmc_part1`

Unmount the filesystem before ejecting the card (or at least use “sync” command).

```
$umount /mnt/mmc_part1
```

End of Test FSL-UT-MMC-0070

40.28 MXC UART Tests

40.28.1 Basic UART Tests

NOTE

Select from the available test cases as needed for your IC and hardware design.

The first four steps are the unit tests performed on development hardware boards. Subsequent steps are for IrDA unit test.

1. Redirect the output of a shell command to another UART. Ensure both ends are using the same baud rate and other port settings.

   Test ID: FSL-UT-UART-MXC-0010

   ```
   stty -F /dev/ttymxc1 115200 -- set the baud rate of UART 2 to 115200
   ls >/dev/ttymxc1 -- the output should be printed on UART2's window
   ```

   End of Test FSL-UT-UART-MXC-0010

2. Start a shell in the background and redirect its standard input, standard output and standard error to a different UART port

   Test ID: FSL-UT-UART-MXC-0020

   ```
   sh </dev/ttymxc2 >/dev/ttymxc2 2>/dev/ttymxc2
   ```

   End of Test FSL-UT-UART-MXC-0020

3. A user space test application is written to test the loop-back IOCTL.

   Test ID: FSL-UT-UART-MXC-0040

   ```
   #test all internal MXC UART Ports (ttymxc/0..ttymxc/XX)
   mxc_uart_test.out /dev/ttymxcXX
   ```

   FSL-UT-UART-MXC-0040

4. Type the following command on the Linux console

   Test ID: FSL-UT-UART-MXC-0030

   ```
   cat /dev/ttymxc0
   ```

   On the board, connect UART 1 to a serial port on your PC. Open a Hyperterminal window that is connected to this serial port. Type characters in the Hyperterminal window. These characters should be printed by the cat application in the console window. Text files should also be transferred (Transfer->Send Text File). The contents of this file should be received by the cat received through UART 1. Note that this test will not pass for serial ports which have the UART DMA Enable/Disable (`UARTx_DMA_ENABLE`) configuration option set to 1 as the port will be operating in a raw mode where the character input is not being parsed. This is expected behavior.

   End of Test FSL-UT-UART-MXC-0030

5. Test Serial Infrared on the boards.

   This test example requires two boards. Before performing the IrDA, align the IrPorts facing each other, about an inch apart.

   The following steps tests Serial IrDA using IrLAN on the board:

   Test ID: FSL-UT-FIRI-0020

   On one board:

   ```
   stty -F /dev/ttymxc0 -echo
   modprobe irtty-sir
   insmod /lib/modules/2.6.22-pdk27_rel1/kernel/net/irda/irlan/irlan.ko access=2
   ifconfig irlan0 10.0.0.1 netmask 255.255.255.0 broadcast 10.0.0.255
   irattach /dev/ttymxc0 -s
   ping 10.0.0.2
   ```

   On another board:

   ```
   stty -F /dev/ttymxc0 -echo
   modprobe irtty-sir
   ```

```
insmod /lib/modules/2.6.22-pdk27_rel1/kernel/net/irda/irlan/irlan.ko access=2
ifconfig irlan0 10.0.0.2 netmask 255.255.255.0 broadcast 10.0.0.255
irattach /dev/ttymxc0 -s
telnet 10.0.0.1
```

End of Test FSL-UT-FIRI-0020

40.28.2 Early UART Support Tests

This testing is done by inspection. After booting the kernel with early UART command line, the kernel boot log messages should follow immediately.

Test Case ID: FSL-UT-EUART-MXC-0010

Use following command line examples for respective boards to test early UART functionality:

- i.MX27:

```
exec -c "noinitrd console=mxcuart,0x1000a000,115200n8 root=/dev/nfs
nfsroot=10.232.184.242:<NFS root file system directory> rw init=/linuxrc ip=dhcp"
```

End of Test Case ID: FSL-UT-EUART-MXC-0010

40.29 FM Driver Tests

The test cases in Section 40.29, “FM Driver Tests“ are available for i.MX27 and i.MX31 3-Stack platforms.

To enable audio output, run amixer command to open loopback mixer:

```
$ amixer cset name='Loopback Line-in' 1
```

If FM driver is configured as “M”, load module before tests:

```
$ insmod mxc_si4702.ko
```

Test ID: FSL-UT-FM-0010

This test starts the FM chip.

Reset FM chip by type below command in the shell

```
$ echo reset > /sys/class/i2c-adapter/i2c-0/0-0010/si4702_ctl
```

Type the following for FM to start work:

```
$ echo start > /sys/class/i2c-adapter/i2c-0/0-0010/si4702_ctl
```

End of Test FSL_UT_FM_0010

Test ID: FSL-UT-FM-0020

This test auto searches the stations:

For seek channels, enter command:

```
$ echo seek > /sys/class/i2c-adapter/i2c-0/0-0010/si4702_ctl
```

End of Test FSL_UT_FM_0020

Test ID: FSL-UT-FM-0030

This test searches for a specific station:

Enter the station number, such as 101, 103, 105, 107, currently only these channel are valid in the command line:

```
$ echo 101 > /sys/class/i2c-adapter/i2c-0/0-0010/si4702_ctl
```

End of Test FSL_UT_FM_0030

Test ID: FSL-UT-FM-0040

This test adjusts the volume.

To adjust the volume, enter vu/vd as follows:

```
$ echo vu > /sys/class/i2c-adapter/i2c-0/0-0010/si4702_ctl
```

End of Test FSL_UT_FM_0040

Test ID: FSL-UT-FM-0050

This test power off the fm chip.

Enter command:

```
$ echo halt > /sys/class/i2c-adapter/i2c-0/0-0010/si4702_ctl
```

End of Test FSL_UT_FM_0050

40.30 MMA7450L Accelerometer Driver Tests

The usage of this driver is heavily based on how the algorithm is implemented, so the MMA7450L driver unit test can only test part of this driver's functions using the command line in sysfs. The tests in Section 40.30, "MMA7450L Accelerometer Driver Tests" are available for i.MX27 and i.MX31 3-Stack platforms.

1. Install module mxc_mma7450.ko
   ```
   $ modprobe mxc_mma7450
   ```
2. Go to directory `/sys/class/i2c-adapter/i2c-0/0-001d`
   ```
   $ cd /sys/class/i2c-adapter/i2c-0/0-001d
   ```
3. Check the register value by type:
   ```
   $ cat mma7450_ctl
   ```

4. Set mma7450 measurement mode by type:

```
$ echo setmod 1 > mma7450_ctl
```

5. Repeat step 3 to check the register value update.

40.31 GPS Tests

Test ID: FSL-UT-GPS-0010

Note that the tests performed in this sections exercise the GPS solution as a system; that is, the hardware and software get tested together. GPS tests are available for i.MX27 Stack platform.

Before test, use command “lsmod” to make sure the GPS gpio control driver module:gps_gpiodrv has been installed, if not, add it on command line:

```
$ modprobe gps_gpiodrv
```

To make a device node on the command line:

```
$ mknod /dev/agpsgpio c 100 1
```

Enable log file to see GPS messages:

```
$ cd /usr/local/bin/
$ vi glconfig.xml
```

Change *cLogEnabled="false"* to *cLogEnabled="true"*.

Set the proper uart port for specific platform in glconfig.xml:

```
use /dev/ttymxc1
```

Run GPS process:

```
$ ./glgps_freescaleLinux glconfig.xml normal&
```

Exit GPS process:

```
$ echo `$pglirm,quit' >/var/run/glgpsctrl
```

There will be a log file generated in ./log/ with name like gl-xx.txt

40.32 RTC Tests

There is a user space test program for RTC that can be located in this release in the `imx-test-2.3.2.tar.gz` file. The source file is `rtctest.c`, which is under the `<ltib_dir>/rpm/BUILD/misc/test/mxc_rtc` directory after un-taring the tarball. This file was taken from `rtc.txt` under the kernel `Documentation` directory and modified to run on the i.MX platforms.

A file called "rtctest.out" is produced after a successful build.

Test ID: FSL-UT-RTC-0010

This test is exactly the same as executing the auto run test. To run the test, download it to the target and type /unit_tests/rtctest.out --full. You should see the following output:

```
RTC Driver Test Example.

Counting 5 update (1/sec) interrupts from reading /dev/rtc0: 1 2 3 4 5
Again, from using select(2) on /dev/rtc0: 1 2 3 4 5

Current RTC date/time is 1-1-1970, 00:12:54.
Alarm time now set to 00:12:59.
Waiting 5 seconds for alarm... okay. Alarm rang.

Periodic IRQ rate was 64Hz.
Counting 20 interrupts at:
2Hz:     1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
4Hz:     1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
8Hz:     1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
16Hz:    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
32Hz:    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
64Hz:    1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

                          *** Test complete ***
```

Typing cat /proc/interrupts shows 131 more events on IRQ 25.

End of Test FSL-UT-RTC-0010

40.33 Watchdog Tests

There is a user space test program for WDOG that can be located in this release in the imx-test-2.3.2.tar.gz file. The source file is wdt_driver_test.c which is located under the <ltib_dir>/rpm/BUILD/misc/test/wdog/ directory after you untar the imx-test-2.3.2 tarball. This file is taken from watchdog.txt under the kernel Documentation/watchdog directory with slight modifications to run on the i.MX platforms.

There should be a file called wdt_driver_test.out after a successful build. To run the test, download it to the target and type ./wdt_driver_test.out. The following text will appear on the terminal:

```
Usage: wdt_driver_test <timeout> <sleep> <test>
timeout: value in seconds to cause wdt timeout/reset
sleep: value in seconds to service the wdt
test: 0 - Service wdt with ioctl(), 1 - with write()
```

To test the WDOG timeout feature with the IOCTL, do the following:

Test ID: FSL-UT-WDOG-0070

```
./wdt_driver_test.out 1 2 0 &
```

This should generate a reset after 3 seconds (a 1 second time-out and a 2 second sleep).

End of Test FSL-UT-WDOG-0070

Test ID: FSL-UT-WDOG-0080

```
./wdt_driver_test.out 2 1 0 &
```

The system should keep running without being reset. This test requires the kernel to be executed with the "`jtag=on`" on some platforms.

End of Test FSL-UT-WDOG-0080

Test ID: FSL-UT-WDOG-0090

To test the WDOG module, the following test programs must be built:

```
mxc_wdog_tm.ko
wdog_test.out.
```

A test module `mxc_wdog_tm.c` was written to test the kernel space API provided by the watchdog module. This test module provides the IOCTL `MXCTEST_WDOG` to user space applications. Download the module `mxc_wdog_tm.ko` to the platform and install it using the following command:

```
insmod mxc_wdog_tm.ko
```

This test module is registered in the `/dev` directory as `mxc_wdog_tm`.

A user space application called `wdog_test.c` was written to use the watchdog test module IOCTLs to simulate various watchdog time-out scenarios. Build this test program and then download the binary for this test (`wdog_test.out`) to the platform and run from the shell prompt. Type `./wdog_test.out` from the shell prompt and follow the help messages to run different tests.

End of Test FSL-UT-WDOG-0090