i.MX51 3-Stack 1.6 Linux

Reference Manual

Part Number: 924-76374
Rev. 5.1.0
10/2009

freescale™
semiconductor

How to Reach Us:

Home Page:
www.freescale.com

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

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

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

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

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

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

Contents

About This Book

Chapter 1 Introduction

Chapter 2 Architecture

Chapter 3
Machine Specific Layer (MSL)

Chapter 4
Smart Direct Memory Access (SDMA) API

Chapter 5
MC13892 Regulator Driver

Chapter 6
MC13892 RTC Driver

Chapter 7
MC13892 Digitizer Driver

Chapter 8
i.MX51 Low-level Power Management (PM) Driver

Chapter 9
CPU Frequency Scaling (CPUFREQ) Driver

Chapter 10
Dynamic Voltage Frequency Scaling (DVFS) Driver

Chapter 11
Image Processing Unit (IPU) Drivers

Chapter 12
Video for Linux Two (V4L2) Driver

Chapter 13
TV Encoder (TVE) Driver

Chapter 14
Video Processing Unit (VPU) Driver

Chapter 15
Graphics Processing Unit (GPU)

Chapter 16
ISL29003 Light Sensor Driver

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

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

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

Chapter 20
NAND Flash Memory Technology Device (MTD) Driver

Chapter 21
Low-Level Keypad Driver

Chapter 22
SMSC LAN9217 Ethernet Driver

Chapter 23
Fast Ethernet Controller (FEC) Driver

Chapter 24
WLAN Driver

Chapter 25
Security Drivers

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

Chapter 27
Inter-IC (I2C) Driver

Chapter 28
Configurable Serial Peripheral Interface (CSPI) Driver

Chapter 29
1-Wire Driver

Chapter 30
MMC/SD/SDIO Host Driver

Chapter 31
Universal Asynchronous Receiver/Transmitter (UART) Driver

Chapter 32
Bluetooth Driver

Chapter 33
ATA Driver

Chapter 34
ARC USB Driver

Chapter 35
Secure Real Time Clock (SRTC) Driver

Chapter 36
Watchdog (WDOG) Driver

Chapter 37
Pulse-Width Modulator (PWM) Driver

Chapter 38
FM Driver

Chapter 39
Global Positioning System (GPS) Driver

Chapter 40
SIM Driver

Chapter 41
OProfile

Chapter 42
Frequently Asked Questions

Tables

Figures

About This Book

The Linux board support package (BSP) represents a porting of the Linux operating system (OS) to the i.MX processors and to their associated reference boards. The BSP supports many of the hardware features on the platforms, as well as most of the Linux OS features not dependent on any specific hardware feature.

Audience

This document is targeted to individuals who will port the i.MX Linux BSP to customer-specific products. The audience is expected to have a working understanding of the Linux 2.6 kernel internals and driver models. An understanding of the i.MX processors is also required.

Conventions

This document uses the following notational conventions:

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

Definitions, Acronyms, and Abbreviations

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

Definitions and Acronyms

Term	Definition
ADC	Asynchronous Display Controller
address translation	Address conversion from virtual domain to physical domain
API	Application Programming Interface
ARM®	Advanced RISC Machines processor architecture
AUDMUX	Digital audio MUX—provides a programmable interconnection for voice, audio, and synchronous data routing between host serial interfaces and peripheral serial interfaces
BCD	Binary Coded Decimal
bus	A path between several devices through data lines
bus load	The percentage of time a bus is busy
CODEC	Coder/decoder or compression/decompression algorithm—used to encode and decode (or compress and decompress) various types of data

Definitions and Acronyms (continued)

Term	Definition
CPU	Central Processing Unit—generic term used to describe a processing core
CRC	Cyclic Redundancy Check—Bit error protection method for data communication
CSI	Camera Sensor Interface
DFS	Dynamic Frequency Scaling
DMA	Direct Memory Access—an independent block that can initiate memory-to-memory data transfers
DPM	Dynamic Power Management
DRAM	Dynamic Random Access Memory
DVFS	Dynamic Voltage Frequency Scaling
EMI	External Memory Interface—controls all IC external memory accesses (read/write/erase/program) from all the masters in the system
Endian	Refers to byte ordering of data in memory.: little endian means that the least significant byte of the data is stored in a lower address than the most significant byte, in big endian, the order of the bytes is reversed
EPIT	Enhanced Periodic Interrupt Timer—a 32-bit set and forget timer capable of providing precise interrupts at regular intervals with minimal processor intervention
FCS	Frame Checker Sequence
FIFO	First In First Out
FIPS	Federal Information Processing Standards—United States Government technical standards published by the National Institute of Standards and Technology (NIST). NIST develops FIPS when there are compelling Federal government requirements such as for security and interoperability but no acceptable industry standards
FIPS-140	Security requirements for cryptographic modules—Federal Information Processing Standard 140-2(FIPS 140-2) is a standard that describes US Federal government requirements that IT products should meet for Sensitive, But Unclassified (SBU) use
Flash	A non-volatile storage device similar to EEPROM, where erasing can only be done in blocks or the entire chip.
Flash path	Path within ROM bootstrap pointing to an executable Flash application
Flush	Procedure to reach cache coherency. Refers to removing a data line from cache. This process includes cleaning the line, invalidating its VBR and resetting the tag valid indicator. The flush is triggered by a software command
GPIO	General Purpose Input/Output
hash	Hash values are produced to access secure data. A hash value (or simply hash), also called a message digest, is a number generated from a string of text. The hash is substantially smaller than the text itself, and is generated by a formula in such a way that it is extremely unlikely that some other text produces the same hash value.
I/O	Input/Output
ICE	In-Circuit Emulation
IP	Intellectual Property
IPU	Image Processing Unit —supports video and graphics processing functions and provides an interface to video/still image sensors and displays
IrDA	Infrared Data Association—a nonprofit organization whose goal is to develop globally adopted specifications for infrared wireless communication

Definitions and Acronyms (continued)

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

Definitions and Acronyms (continued)

Term	Definition
ROM bootstrap	Internal boot code encompassing the main boot flow as well as exception vectors
RTIC	Real-time integrity checker—a security hardware module
SCC	SeCurity Controller—a security hardware module
SDMA	Smart Direct Memory Access
SDRAM	Synchronous Dynamic Random Access Memory
SoC	System on a Chip
SPBA	Shared Peripheral Bus Arbiter—a three-to-one IP-Bus arbiter, with a resource-locking mechanism
SPI	Serial Peripheral Interface—a full-duplex synchronous serial interface for connecting low-/medium-bandwidth external devices using four wires. SPI devices communicate using a master/slave relationship over two data lines and two control lines: *Also see SS, SCLK, MISO, and MOSI*
SRAM	Static Random Access Memory
SSI	Synchronous-Serial Interface—standardized interface for serial data transfer
TBD	To Be Determined
UART	Universal Asynchronous Receiver/Transmitter—asynchronous serial communication to external devices
UID	Unique ID–a field in the processor and CSF identifying a device or group of devices
USB	Universal Serial Bus—an external bus standard that supports high speed data transfers. The USB 1.1 specification supports data transfer rates of up to 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.MX51 PDK Linux Quick Start Guide*
- *BSP API Document (BSP Doxygen Code Documentation)*
- *i.MX51 PDK Linux User's Guide*
- *i.MX51 PDK Hardware User's Guide*
- *MCIMX51 Multimedia Applications Processor Reference Manual* (MCIMX51RM)
- [KERN] *Linux kernel coding style*. 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 processor:

- i.MX51 Applications Processor

NOTE

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

The purpose of this software package is to support Linux on the i.MX51 family of integrated circuits (ICs) and their associated platforms (3-Stack boardEVK). It provides the software necessary to interface the standard open-source Linux kernel to the i.MX hardware. The goal is to enable Freescale customers to rapidly build products based on i.MX devices that use the Linux OS.

The BSP is not a platform or product reference implementation. It does not contain all of the product-specific drivers, hardware-independent software stacks, 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 2.6.28 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. Linux BSP Supported Features

Feature	Description	Chapter Source	Applicable Platform
Machine Specific Layer			
MSL	MSL (Machine Specific Layer) supports interrupts, Timer, Memory Map, GPIO/IOMUX, SPBA, SDMA. • Interrupts (AITC/AVIC): The Linux kernel contains common ARM code for handling interrupts. The MSL contains platform-specific implementations of functions for interfacing the Linux kernel to the Cortex-A8 interrupt controller. • Timer (GPT): The General Purpose Timer (GPT) is set up to generate an interrupt as programmed to provide OS ticks. Linux facilitates timer use through various functions for timing delays, measurement, events, alarms, high resolution timer features, and so on. Linux defines the MSL timer API required for the OS-tick timer and does not expose it beyond the kernel tick implementation. • GPIO/EDIO/IOMUX: The GPIO and EDIO components in the MSL provide an abstraction layer between the various drivers and the configuration and utilization of the system, including GPIO, IOMUX, and external board I/O. The IO software module is board-specific, and resides in the MSL layer as a self-contained set of files. I/O configuration changes are centralized in the GPIO module so that changes are not required in the various drivers. • SPBA: The Shared Peripheral Bus Arbiter (SPBA) provides an arbitration mechanism among multiple masters to allow access to the shared peripherals. The SPBA implementation under MSL defines the API to allow different masters to take or release ownership of a shared peripheral.	Chapter 3, “Machine Specific Layer (MSL)”	All
SDMA API	The Smart Direct Memory Access (SDMA) API driver controls the SDMA hardware. It provides an API to other drivers for transferring data between MCU, DSP and peripherals. The SDMA controller is responsible for transferring data between the MCU memory space, peripherals, and the DSP memory space. The SDMA API allows other drivers to initialize the scripts, pass parameters and control their execution. SDMA is based on a microRISC engine that runs channel-specific scripts.	Chapter 4, “Smart Direct Memory Access (SDMA) API”	i.MX51
Power Management IC (PMIC) Drivers			
MC13892 Regulator	MC13892 regulator driver provides the low-level control of the power supply regulators, setting voltage level and enable/disable regulators.	Chapter 5, “MC13892 Regulator Driver”	i.MX51

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

Feature	Description	Chapter Source	Applicable Platform
MC13892 RTC	MC13892 RTC driver for Linux provides the access to PMIC RTC control circuits	Chapter 6, "MC13892 RTC Driver"	i.MX51
MC13892 Digitizer Driver	MC13892 digitizer driver for Linux that provides low-level access to the PMIC analog-to-digital converters	Chapter 7, "MC13892 Digitizer Driver"	i.MX51
Power Management Drivers			
Low-level PM Drivers	The low-level power management driver is responsible for implementing hardware-specific operations to meet power requirements and also to conserve power on the development platforms. Driver implementations are often different for different platforms. It is used by the DPM layer.	Chapter 8, "i.MX51 Low-level Power Management (PM) Driver"	i.MX51
CPU Frequency Scaling	The CPU frequency scaling device driver allows the clock speed of the CPUs to be changed on the fly.	Chapter 9, "CPU Frequency Scaling (CPUFREQ) Driver"	i.MX51
DVFS	The Dynamic Voltage Frequency Scaling (DVFS) device driver allows simple dynamic voltage frequency scaling. The frequency of the core clock domain and the voltage of the core power domain can be changed on the fly with all modules continuing their normal operations.	Chapter 10, "Dynamic Voltage Frequency Scaling (DVFS) Driver"	i.MX51
Multimedia Drivers			
TV-OUT	TV-OUT is an integrated television encoder that encodes video signals and generates synchronization signals for a given television standard.	Chapter 13, "TV Encoder (TVE) Driver"	i.MX51
IPU	The Image Processing Unit (IPU) is designed to support video and graphics processing functions and to interface with video/still image sensors and displays. The IPU driver is a self-contained driver module in the Linux kernel. It contains a custom kernel-level API to manipulate logical channels. A logical channel represents a complete IPU processing flow. The IPU driver includes a frame buffer driver, a V4L2 device driver, and low-level IPU drivers.	Chapter 11, "Image Processing Unit (IPU) Drivers"	i.MX51
V4L2 Output	The Video for Linux 2 (V4L2) output driver uses the IPU post-processing functions for video output. The driver implements the standard V4L2 API for output devices.	Chapter 12, "Video for Linux Two (V4L2) Driver"	i.MX51
V4L2 Capture	The Video for Linux 2 (V4L2) capture device includes two interfaces: the capture interface and the overlay interface. The capture interface records the video stream. The overlay interface displays the preview video.	Chapter 12, "Video for Linux Two (V4L2) Driver"	i.MX51
VPU	The Video Processing Unit (VPU) is a multi-standard video decoder and encoder that can perform decoding and encoding of various video formats.	Chapter 14, "Video Processing Unit (VPU) Driver"	i.MX51

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

Feature	Description	Chapter Source	Applicable Platform
AMD GPU	The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D/3D graphics applications.	Chapter 15, “Graphics Processing Unit (GPU)”	i.MX51
Light sensor	The ISL29003 driver provides the interfaces to ambient light sensing, backlight control	Chapter 16, “ISL29003 Light Sensor Driver	i.MX51
Sound Drivers			
ALSA Sound	The Advanced Linux Sound Architecture (ALSA) is a sound driver that provides ALSA and OSS compatible applications with the means to perform audio playback and recording functions. ALSA has a user-space component called ALSAlib that can extend the features of audio hardware by emulating the same in software (user space), such as resampling, software mixing, snooping, and so on. The ASoC Sound driver supports stereo codec playback and capture through SSI.	Chapter 17, “Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver”	i.MX51
S/PDIF	The S/PDIF driver is designed under the Linux ALSA subsystem. It implements one playback device for Tx and one capture device for Rx. MX51only supports S/PDIF transmitter.	Chapter 18, “The Sony/Philips Digital Interface (S/PDIF) Tx Driver”	i.MX51
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 20, “NAND Flash Memory Technology Device (MTD) Driver”	i.MX51
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 8×8 external key pad matrix of single poll switches.	Chapter 21, “Low-Level Keypad Driver”	i.MX51
Networking Drivers			
LAN9217 Ethernet	The SMSC LAN9217 Ethernet driver interfaces SMSC LAN9217-specific functions with the standard Linux kernel network module.	Chapter 22, “SMSC LAN9217 Ethernet Driver”	i.MX51

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

Feature	Description	Chapter Source	Applicable Platform
FEC	The FEC Driver performs the full set of IEEE 802.3/Ethernet CSMA/CD media access control and channel interface functions. The FEC requires an external interface adaptor and transceiver function to complete the interface to the Ethernet media. It supports half or full-duplex operation on 10 Mbps- or 100 Mbps-related Ethernet networks.	Chapter 23, “Fast Ethernet Controller (FEC) Driver”	i.MX51
WLAN	The WLAN driver is used to drive the APM6628 module to implement Wi-Fi functionality.	Chapter 24, “WLAN Driver”	i.MX51
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.	Chapter 25, “Security Drivers”	i.MX51
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	Chapter 26, “Symmetric/Asymmetric Hashing and Random Accelerator (Sahara) Drivers”	i.MX51
Bus Drivers			
I^2C	The I^2C bus driver is a low-level interface that is used to interface with the I^2C bus. This driver is invoked by the I^2C chip driver; it is not exposed to the user space. The standard Linux kernel contains a core I^2C module that is used by the chip driver to access the bus driver to transfer data over the I^2C bus. This bus driver supports: • Compatibility with the I^2C bus standard • Bit rates up to 400 Kbps • Standard I^2C master mode • Power management features by suspending and resuming I^2C.	Chapter 27, “Inter-IC (I2C) Driver”	i.MX51

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

Feature	Description	Chapter Source	Applicable Platform
1-Wire	This is an integrated 1-Wire interface. The driver is implemented as a character driver and provides a custom user space API. This driver supports: • A single 1-Wire memory device connected to the 1-Wire peripheral for read/write bit and read/write byte operations • 1-Wire peripheral in the product for single device detection and selection • Interface to the 1-Wire peripheral at the read/write block and read/write page level.	Chapter 29, “1-Wire Driver”	i.MX51
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 28, “Configurable Serial Peripheral Interface (CSPI) Driver”	i.MX51
MMC/SD/SDIO - eSDHC	The MMC/SD/SDIO Host driver implements the standard Linux driver interface to eSDHC.	Chapter 30, “MMC/SD/SDIO Host Driver”	i.MX51
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 31, “Universal Asynchronous Receiver/Transmitter (UART) Driver”	i.MX51
General Drivers			
USB	The USB driver implements a standard Linux driver interface to the ARC USB-OTG controller.	Chapter 34, “ARC USB Driver”	i.MX51
Bluetooth	The Bluetooth driver provides synchronous and asynchronous wireless connection among multiple devices.	Chapter 32, “Bluetooth Driver”	i.MX51
ATA	The ATA module is an AT attachment host interface. Its main use is to interface with hard disk devices. The ATA driver is compliant with the ATA-6 standard, and supports the following protocols: • PIO mode 0, 1, 2, 3, and 4 • Multi-word DMA mode 0, 1, and 2 • Ultra DMA mode 0, 1, 2, 3, and 4and 3 with bus clocks of 50MHz or higher • Ultra DMA mode 5 with bus clock of 80MHz or higher. It supports the IDE and LibATA interfaces.	Chapter 33, “ATA Driver”	i.MX51
SRTC	The SRTC driver is designed to support MXC Secure RTC module to keep the time and date	Chapter 35, “Secure Real Time Clock (SRTC) Driver”	i.MX51

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

Feature	Description	Chapter Source	Applicable Platform
WatchDog	The Watchdog Timer module protects against system failures by providing an escape from unexpected hang or infinite loop situations or programming errors. This WDOG implements the following features. • Generates a reset signal if it is enabled but not serviced within a predefined time-out value • Does not generate a reset signal if it is serviced within a predefined time-out value	Chapter 36, “Watchdog (WDOG) Driver”	i.MX51
MXC PWM driver	The MXC PWM driver provides the interfaces to access MXC PWM signals	Chapter 37, “Pulse-Width Modulator (PWM) Driver”	i.MX51
FM (Si4702)	The FM (Si4702) driver provides the interfaces to control Silicon Laboratories Si4702 FM tuner integrated circuit.	Chapter 38, “FM Driver”	i.MX51
GPS	The communications driver provides a serial interface to the core driver and communicates with the external GPS chip set.	Chapter 39, “Global Positioning System (GPS) Driver	i.MX51
SIM	The SIM driver implements a Linux driver interface to the Subscriber Identification Module (SIM).		i.MX51
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.	See the document in Redboot release package	i.MX51
uBoot	uBoot is an open source boot loader.	See uBoot User guide	i.MX51
GUI			
gnome	gnome is a Network Object Model Environment supported by the GUN.	See Gnome mobile release note	i.MX51
Tools			
OProfile	OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.	Chapter 41, “OProfile”	i.MX51

Chapter 2
Architecture

This chapter describes the overall architecture of the Linux port to the i.MX processor. 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.

2.1 Linux BSP Block Diagram

Figure 2-1 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, as well as hardware-specific drivers and functions provided by Freescale for the i.MX processors.

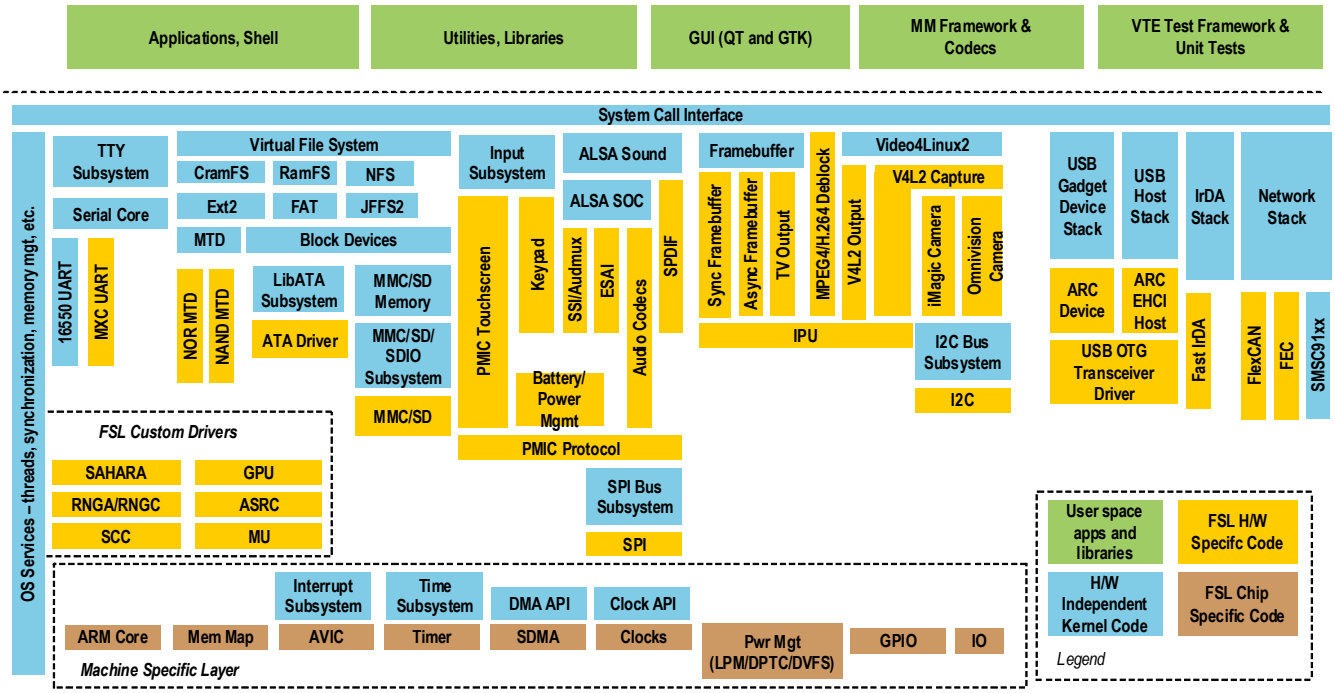


Figure 2-1. BSP Block Diagram

2.2 Kernel

The i.MX Linux port is based on the standard Linux kernel. The kernel supports many of the features found in most modern embedded OSs such as:

- Process and thread management
- Memory management (memory mapping, allocation/deallocation, MMU, and L1/L2 cache control)
- Resource management (interrupts)
- Power management
- File systems (VFS, cramfs, ext2, ramfs, NFS, devfs, JFFS2, FAT)
- Linux Device 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.

2.2.1 Kernel Configuration

For this BSP release, kernel configuration is done through the Linux Target Image Builder (LTIB). See the LTIB documentation for details. The following are some of the configuration settings available on some platforms, that are different from the standard features:

- Embedded mode
- Module loading/unloading
- Cortex-A8
- 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
- Frame buffer
- Kernel debugging
- Automatic kernel module loading
- Power management
- Memory Technology Device (MTD) support
- USB Host/device multiplexing
- Unsorted block images (UBI) support
- Flash translation layer (FTL)
- CPU frequency scaling

2.2.2 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 2-1.

Table 2-1. MSL Directories

Platform	Directory
i.MX51 3-Stack	<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51

For more information, see Chapter 3, "Machine Specific Layer (MSL)."

2.2.2.1 Memory Map

Before the kernel starts running in the virtual space, the physical-to-virtual address mapping for the I/O 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 starting address of virtual addresses, starting address of physical addresses, and the size of the memory region and the type of the region.

2.2.2.2 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 Cortex-A8 Interrupt Controller.

Together, they support the following capabilities:

- AVIC initialization
- ARM Interrupt Controller (AITC) initialization
- Interrupt enable/disable control
- ISR binding
- ISR dispatch
- Interrupt chaining
- Standard Linux API for accessing interrupt functions

2.2.2.3 General Purpose Timer (GPT)

The GPT is configured to generate an interrupt every 10 ms 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 in low-power modes other than idle.

2.2.2.4 Smart Direct Memory Access (SDMA) API

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

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

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

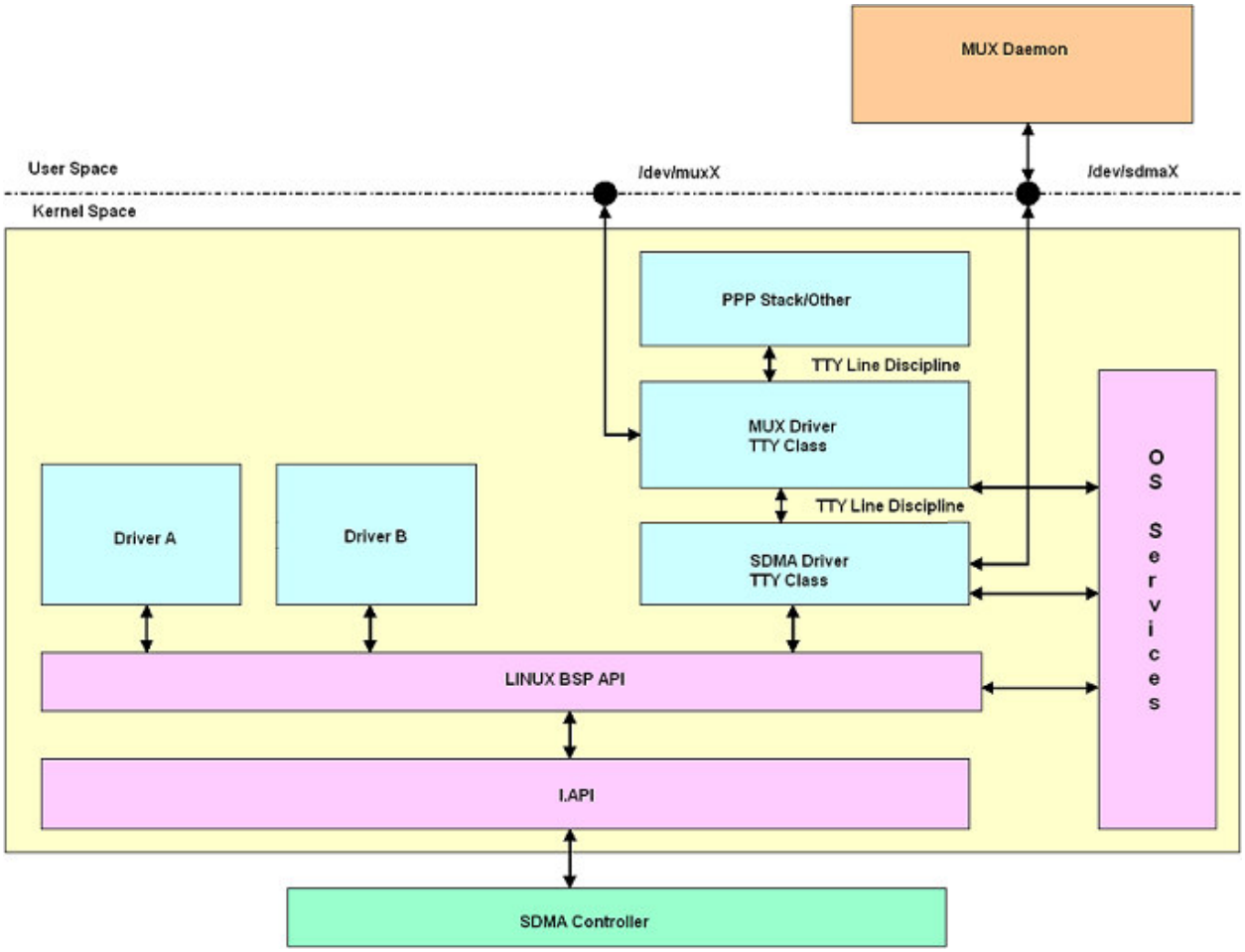


Figure 2-2. SDMA Block Diagram

2.2.2.5 Input/Output (I/O)

The Input/Output (I/O) 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 I/O 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. The additions to the module are included in every new release of the BSP.

2.2.2.6 Shared Peripheral Bus Arbiter (SPBA)

The SPBA provides an arbitration mechanism to allow 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.

2.3 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 a `modprobe.conf` file that contain the dependency information for the modules.

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

2.3.1 Universal Asynchronous Receiver/Transmitter (UART) Driver

The i.MX family of processors support a Universal Asynchronous Receiver/Transmitter (UART) driver.

2.3.1.1 UART Driver

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

- Interrupt-driven and SDMA-driven transmit/receive of characters
- Standard Linux baud rates up to 1.5 Mbps
- Transmitting and receiving characters with 7-bit and 8-bit character lengths
- Transmitting one or two 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 sets 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 which is needed to bring up the command prompt through one of the UART ports

A kernel configuration parameter gives the user the ability to choose the UART driver, and also to choose whether the UART should be used as the system console.

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

2.3.2 Real-Time Clock (RTC) Driver

The 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.

2.3.3 Watchdog Timer (WDOG) Driver

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

The WDOG software implementation provides routines to service the WDOG timer, so that the timeout does not occur. The WDOG is serviced (at the same time for the platforms with two WDOGs) if it is already enabled before the Linux kernel boots (enabled by boot loader or ROM) with a configurable service interval. 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 is present (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.

2.3.4 SDMA API Driver

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

Functions of the SDMA API include:

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

2.3.5 Image Processing Unit (IPU) Driver

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

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

- Synchronous Display Controller (SDC)
- Asynchronous Display Controller (ADC)
- Display Interface (DI)
- Image DMA Controller (IDMAC)

- CMOS Sensor Interface (CSI)
- Image Converter (IC)

2.3.6 Video for Linux 2 (V4L2) Driver

The Video for Linux Two (V4L2) drivers are plug-ins to the V4L2 framework that enable support for camera and preprocessing functions, and video and post-processing functions. The V4L2 Linux standard API specification is available at http://v4l2spec.bytesex.org/spec/.

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

2.3.8 Keypad Driver

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

It supports the following features:

- Supports up to 8×8 external key pad matrix of single poll switches. The keypad matrix can be configured
- Any pins not used for keypad are available as general purpose I/O through an IOCTL call
- The keypad driver supports two modes of interface to the upper layer. The modes are raw mode and map mode (xlate mode)
- The keypad driver is implemented as a single driver
- When configured in raw mode the scan code of keys pressed and released are sent to the upper layer
- When configured in map mode the map code (key mapping) of keys pressed and released are sent to the upper layer. scan codes are converted into map codes from the keymap lookup table
- Dynamic configuration of keymap translation table (static default) through an IOCTL call
- The keypad mode can be set using `KDSKBMODE` IOCTL call
- Supports multiple key presses (with required keypad design)
- A long key press can be configured to generate multiple key press events
- Supports key press detection in standby mode

- Follows the standard Linux Keyboard API
- Key chording handled by users of this driver (GUI)

2.3.9 Memory Technology Device (MTD) Driver

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.

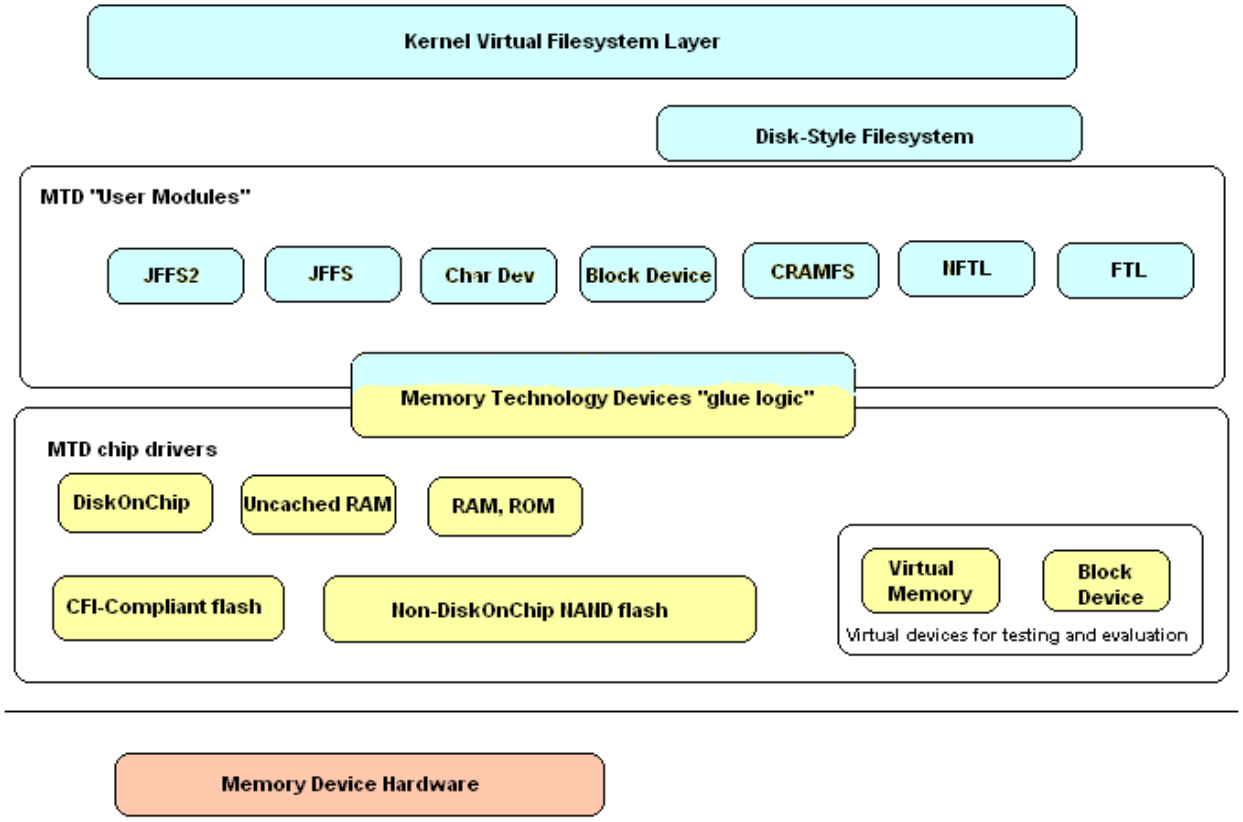


Figure 2-3. MTD Architecture

Figure 2-3 is excerpted from *Building Embedded Linux Systems*, 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.

2.3.9.1 NOR MTD Driver

The NOR MTD driver is board-specific as it depends on the actual NOR Flash chip (Common Flash Interface or CFI-compliant) on the board and can have file systems, such as CRAMFS and JFFS2 on top of it. The driver implementation supports the lowest level operations on the Flash chip, such as read, write and erase. The NOR MTD supports XIP on Flash devices which support writing some Flash banks while executing from other banks (as long as proper bank partitioning is done). The BSP by default creates static MTD partitions in the driver source code to support either the Intel Strata Flash or the Spansion Flash on the board. It also allows the RedBoot partitions to be used (and have higher priority over the static partitions) if these partitions exist in the RedBoot.

2.3.9.2 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.

2.3.10 Networking Drivers

The networking drivers are described in the next sections.

2.3.10.1 SMSC LAN9217 Ethernet Driver

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

The SMSC LAN9217 Ethernet Driver has the following features:

- Efficient PacketPage Architecture that can operate in I/O and memory space, and as a DMA slave
- Full duplex operation
- On-chip RAM buffers for transmission and reception of frames
- Programmable transmit features like automatic retransmission on collision and automatic CRC generation
- EEPROM support for configuration
- MAC address setting
- 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.

2.3.11 USB Driver

The Linux kernel supports two main types of USB drivers: drivers on a host system and drivers on a device. A common USB host is a desktop computer. The USB drivers for a host system control the USB devices that are plugged into it. The USB drivers in a device, control how that single device looks to the host computer as a USB device. Because the term “USB device drivers” is very confusing, the USB developers have created the term “USB gadget drivers” to describe the drivers that control a USB device that connects to a computer.

2.3.11.1 USB Host-Side API Model

Within the Linux kernel, host-side drivers for USB devices talk to the usbcore APIs. There are two types of public usbcore APIs, targeted at two different layers of USB driver:

- General purpose drivers, exposed through driver frameworks such as block, character, or network devices
- Drivers that are part of the core, which are involved in managing a USB bus.

Such core drivers include the hub driver, which manages trees of USB devices, and several different kinds of host controller drivers (HCDs), which control individual buses. For more information, see Chapter 2 of http://www.kernel.org/doc/htmldocs/usb.html.

The device model seen by USB drivers is relatively complex:

- USB supports four kinds of data transfer (control, bulk, interrupt, and isochronous). Two transfer types use bandwidth as it is available (control and bulk), while the other two types of transfer (interrupt and isochronous) are scheduled to provide guaranteed bandwidth.
- The device description model includes one or more configurations per device, only one of which is active at a time. Devices that are capable of high speed operation must also support full speed configurations, along with a way to ask about the other speed configurations that might be used.
- Configurations have one or more interfaces. Interfaces may be standardized by USB Class specifications, or may be specific to a vendor or device.
- Interfaces have one or more endpoints, each of which supports one type and direction of data transfer such as bulk out or interrupt in.
- The only host-side drivers that actually touch hardware (reading/writing registers, handling IRQs, and so on) are the HCDs.

2.3.11.2 USB Device-Side Gadget Framework

The Linux Gadget API can be used by peripherals, which act in the USB device (slave) role.

Components of the Gadget Framework (see http://www.linux-usb.org/gadget/) are as follows:

- Peripheral Controller Drivers—implement the Gadget API, and are the only layers that talk directly to the hardware. Different controller hardware needs different drivers, which may also need board-specific customization. These provide a software gadget device, visible in sysfs. This device can be thought of as being the virtual hardware to which the higher-level drivers are written.

- Gadget Drivers—use the Gadget API, and can often be written to be hardware-neutral. A gadget driver implements one or more functions, each providing a different capability to the USB host, such as a network link or speakers.
- Upper Layers, such as the network, file system, or block I/O subsystems—generate and consume the data that the gadget driver transfers to the host through the controller driver.

2.3.11.3 USB OTG Framework

Systems need specialized hardware support to implement OTG, including a special Mini-AB jack and associated transceiver to support Dual-Role operation. They can act either as a host, using the standard Linux-USB host side driver stack, or as a peripheral, using the Gadget framework. To do that, the system software relies on small additions to those programming interfaces, and on a new internal component (here called an OTG Controller) affecting which driver stack connects to the OTG port. In each role, the system can re-use the existing pool of hardware-neutral drivers, layered on top of the controller driver interfaces (usb_bus or usb_gadget). Such drivers need at most minor changes, and most of the calls added to support OTG can also benefit non-OTG products.

- Gadget drivers test the is_otg flag, and use it to determine whether or not to include an OTG descriptor in each of their configurations.
- Gadget drivers may need changes to support the two new OTG protocols, exposed in new gadget attributes such as b_hnp_enable flag. HNP support should be reported through a user interface (two LEDs could suffice), and is triggered in some cases when the host suspends the peripheral. SRP support can be user-initiated just like remote wakeup, probably by pressing the same button.
- On the host side, USB device drivers need to be taught to trigger HNP at appropriate moments, using usb_suspend_device(). That also conserves battery power, which is useful even for non-OTG configurations.
- Also on the host side, a driver must support the OTG Targeted Peripheral List, a whitelist used to reject peripherals not supported with a given Linux OTG host. This whitelist is product-specific—each product must modify `otg_whitelist.h` to match its interoperability specification.

Non-OTG Linux hosts, such as PCs and workstations, normally have some solution for adding drivers, so that peripherals that are not recognized can eventually be supported. That approach is unreasonable for consumer products that may never have their firmware upgraded, and where it is usually unrealistic to expect traditional PC/workstation/server kinds of support model to work. For example, it is often impractical to change device firmware once the product has been distributed, so driver bugs cannot normally be fixed if they are found after shipment.

Additional changes are needed below those hardware-neutral usb_bus and usb_gadget driver interfaces but those are not discussed here. Those affect the hardware-specific code for each USB Host or Peripheral controller, and how the HCD initializes (since OTG can be active only on a single port). They also involve what may be called an OTG Controller Driver, managing the OTG transceiver and the OTG state machine logic as well as much of the root hub behavior for the OTG port. The OTG controller driver needs to activate and deactivate USB controllers depending on the relevant device role. Some related changes were needed inside usbcore, so that it can identify OTG-capable devices and respond appropriately to HNP or SRP protocols.

2.3.12 Security Drivers

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

2.3.12.1 Security Controller (SCC) Module 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 SCC is a part of the Freescale platform independent security architecture (PISA). It supports the following features:

- 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
- Autonomous hardware security state controller with debug inputs that are tied to all platform test access detection signals to trigger shutdown
- Self-clearing (zeroing) 2 Kbyte 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
- Security Timer which is an independent security watchdog timer whose time-out triggers a security violation
- Algorithm Sequence Checker (ASC) which can be used by software to force software synchronization to the ASCs internal linear feedback shift register (LFSR) as a software assurance check
- 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
- 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

2.3.12.2 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
- Operates on 512 bits at a time, and in the end pads 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 SHA-1 hash over a number of large, potentially non-contiguous segments of memory
- Power management

2.3.12.3 Run-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 peripheral memory contents and assist with boot authentication. The RTIC has the ability to verify the memory contents during system boot and 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 two segments per block)
- Works with High Assurance Boot process
- Support for up to four independent memory blocks
- Programmable DMA bus duty cycle timer and watchdog timer

2.3.13 General Drivers

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

- Multimedia Card (MMC)/Secure Digital (SD) driver
- I^2C Client and Bus drivers
- Dynamic Power Management (DPM) driver

2.3.13.1 MMC/SD Host Driver

The MMC/SD card driver implements a standard Linux MMC host driver SSP interface configured to work in MMC/SD mode. The driver is an underlying layer for the Linux MMC block driver that follows standard Linux driver API. The driver has the following features:

- MMC/SD cards
- Standard MMC/SD commands

- 1-bit or 4-bit operation
- Card insertion and removal events
- Write protection signal

2.3.13.2 MMC/SD Slot Driver

The 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. This driver supports the following features:

- 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 32×16 bit data buffer built in
- Plug and play support
- 100 Mbps Maximum hardware data rate in 4-bit mode
- 1-bit or 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

2.3.13.3 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
- Bit rates up to 400 Kbps
- Start and stop signal generation/detection
- Acknowledge bit generation/detection
- Interrupt-driven, byte-by-byte data transfer
- Standard I^2C master mode
- Power management features by suspending and resuming I^2C

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

2.3.13.4 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.

2.3.13.5 Dynamic Power Management (DPM) Driver

DPM refers to power management schemes implemented while programs are 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

2.3.13.5.1 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.

2.3.13.5.2 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.

2.3.13.5.3 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.

2.3.13.5.4 Policy Managers

A policy maps each operating state to a congruent 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 2-4 shows the high level design for DPM.

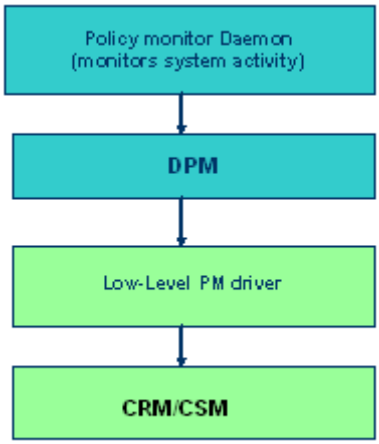


Figure 2-4. DPM High Level Design

Figure 2-5 shows the DPM architecture block diagram.

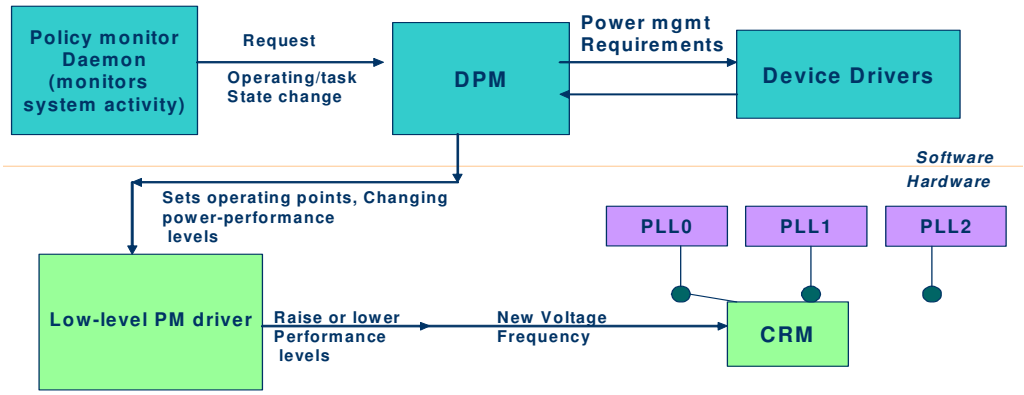


Figure 2-5. DPM Architecture Block Diagram

2.3.13.6 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 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 managed by reducing the voltage/frequency and the severity of clock gating.

2.3.13.7 Dynamic Voltage and Frequency Scaling (DVFS) Driver

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

2.3.13.8 Dynamic Process and Temperature Compensation (DPTC) Driver

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

2.4 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 several purposes:

- Sets up the system, such as:
  - AHB Lite IP Interface (AIPS)
  - Multi Layer Cross Bar Switch (MAX)
  - Memory
  - Different clocks
- Loads Linux kernel image to SDRAM
- Obtains proper information for the Linux kernel
- Passes control to the Linux kernel

NOTE

Not all boot loaders are supported on all boards.

2.4.1 Functions of Boot Loaders

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

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

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

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

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

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

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

The following boot loaders are provided in the BSP:

- RedBoot

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

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

2.4.2 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.

Chapter 3
Machine Specific Layer (MSL)

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

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

These modules are normally available in the following directory:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51
```

The header files are implemented under the following directory:

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

The MSL layer contains not only the modules common to all the boards using the same processor, such as the interrupts and timer, but it also contains modules specific to each board, such as the memory map. The following sections describe the basic hardware and software operation and the software interfaces for MSL modules. First, the common modules, such as Interrupts and Timer are discussed. Next, the board-specific modules, such as Memory Map and general purpose input/output (GPIO) (including IOMUX on some platforms) are detailed. Because of the complexity of the SDMA module, its design is explained in Chapter 4, “Smart Direct Memory Access (SDMA) API.”

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

3.1 Interrupts

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

3.1.1 Interrupt Hardware Operation

The Interrupt Controller controls and prioritizes a maximum of 64 internal and external interrupt sources. Each source can be enabled or disabled by configuring the Interrupt Enable Register or using the Interrupt Enable/Disable Number Registers. When an interrupt source is enabled and the corresponding interrupt source is asserted, the Interrupt Controller asserts a normal or a fast interrupt request depending on the associated Interrupt Type Register setting.

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

3.1.2 Interrupt Software Operation

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

The following file has a description of the ARM interrupt architecture.

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

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

3.1.3 Interrupt Features

The interrupt implementation supports the following features:

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

3.1.4 Interrupt Source Code Structure

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

`tzic.c (If CONFIG_MXC_TZIC is selected)`

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

`hardware.h`
`irqs.h`

Table 3-1 lists the source files for interrupts.

Table 3-1. Interrupt Files

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

3.1.5 Interrupt Programming Interface

The machine-specific interrupt implementation exports a single function. This function initializes the Interrupt Controller hardware and registers functions for interrupt enable and disable from each interrupt source. This is done with the global structure `irq_desc` of type `struct 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 (on some platforms) EDIO interrupts. This allows drivers to use the standard interrupt interface supported by ARM Linux, such as the `request_irq()` and `free_irq()` functions.

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

3.2.1 Timer Hardware Operation

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

3.2.2 Timer Software Operation

The timer software implementation provides an initialization function that initializes the GPT with the proper clock source, interrupt mode and interrupt interval. The timer then registers its interrupt service routine and starts timing. The interrupt service routine is required to service the OS for the purposes mentioned in Section 3.2, “Timer.” Another function provides the time elapsed as the last timer interrupt.

3.2.3 Timer Features

The timer implementation supports the following features:

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

3.2.4 Timer Source Code Structure

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

3.3 Memory Map

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

3.3.1 Memory Map Hardware Operation

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

3.3.2 Memory Map Software Operation

A table mapping the virtual memory to physical memory is implemented for i.MX platforms as defined in the `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51/mm.c` file.

3.3.3 Memory Map Features

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

3.3.4 Memory Map Source Code Structure

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

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/include/mach
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-imx
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51
```

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

Table 3-2. Memory Map Files

File	Description
mx51.h	Header files for the IO module physical addresses
hardware.h	Macro header file
mm.c	Memory map definition file

3.3.5 Memory Map Programming Interface

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

3.4 IOMUX

The limited number of pins of highly integrated processors can have multiple purposes. The IOMUX module controls a pin usage so that the same pin can be configured for different purposes and can be used by different modules. This is a common way to reduce the pin count while meeting the requirements from various customers. Platforms that do not have the IOMUX hardware module can do pin muxing through the GPIO module.

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

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

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose and cannot be changed by software. Otherwise, the IOMUX module needs to be configured to serve a particular purpose that is dictated by the system (board) design. If the pin is connected to an external UART transceiver and therefore to be used as the UART data transmit signal, it should be configured as the primary function. If the pin is connected to an external Ethernet controller for interrupting the ARM core, then it should be configured as GPIO input pin with interrupt enabled. Again, be aware that the software does not have control over what function a pin should have. The software only configures pin usage according to the system design.

3.4.1 IOMUX Hardware Operation

The following discussion applies only to those processors that have an IOMUX hardware module. The IOMUX controller registers are briefly described here. For detailed information, refer to the pin multiplexing section of the IC Reference Manual.

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

3.4.2 IOMUX Software Operation

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

3.4.3 IOMUX Features

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

3.4.4 IOMUX Source Code Structure

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

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51/`

Table 3-3. IOMUX Files

File	Description
iomux.c	IOMUX function implementation
mx51_pins.h	Pin definitions in the iomux_pins enum

3.4.5 IOMUX Programming Interface

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

3.4.6 IOMUX Control Through GPIO Module

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

For a multi-purpose pin, the GPIO controller provides the multiplexing control so that each pin may be configured either as a functional pin (which can be subdivided into either major function or one alternate function) whose operation is controlled by a specific hardware module, or it can be configured as a GPIO pin, in which case, the pin is controlled by the user through software with further configuration through the GPIO module. In addition, there are some special configurations for a GPIO pin (such as output based A_IN, B_IN, C_IN or DATA register, but input based A_OUT or B_OUT).

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose which can not be changed by software. Otherwise, the GPIO module needs to be configured properly to serve a particular purpose that is dictated with the system (board) design. If this pin is connected to an external UART transceiver, it should be configured as the primary function or if this pin is connected to an external Ethernet controller for interrupting the core, then it should be configured as GPIO input pin with interrupt enabled. The software does not have control over what function a pin should have. The software only configures a pin for that usage according to the system design.

3.4.6.1 GPIO Hardware Operation

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

3.4.6.1.1 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.

3.4.6.1.2 PULLUP Control

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

3.4.6.2 GPIO Software Operation

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

3.4.6.3 GPIO Features

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

3.4.6.4 GPIO Source Code Structure

The GPIO 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-51/
```

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

Table 3-4. IOMUX Through GPIO Files

File	Description
iomux.c	IOMUX function implementation
dummy_gpio.c	Dummy GPIO interfaces. GPIO configuration is initialized in earlier phase and all GPIO activate/deactivate functions used in the drivers are dummied.
mx51_3stack_gpio.c	GPIO setup functions
mx51_pins.h	Pin name definitions

3.4.6.5 GPIO Programming Interface

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

3.5 General Purpose Input/Output (GPIO)

The GPIO module provides general-purpose pins that can be configured as either inputs or outputs. When configured as an output, the pin state (high or low) can be controlled by writing to an internal register. When configured as an input, the pin input state can be read from an internal register.

3.5.1 GPIO Software Operation

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

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

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

3.5.1.1 API for GPIO

The GPIO implementation supports the following features:

- An API for registering an interrupt service routine to a GPIO interrupt. This is made possible as the number of interrupts defined by `NR_IRQS` is expanded to accommodate all the possible GPIO pins that are capable of generating interrupts.
- Functions to request and free an IOMUX pin. If a pin is used as GPIO, another set of request/free function calls are provided. The user should check the return value of the request calls to see if the pin has already been reserved before modifying the pin state. The free function calls should be made when the pin is not needed. See the API document for more details.

- Aligned parameter passing for both IOMUX and GPIO function calls. In this implementation the same enumeration for `iomux_pins` is used for both IOMUX and GPIO calls and the user does not have to figure out in which bit position a pin is located in the GPIO module.
- Minimal changes required for the public drivers such as Ethernet and UART drivers as no special GPIO function call is needed for registering an interrupt.

3.5.2 GPIO Features

This GPIO implementation supports the following features:

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

3.5.3 GPIO Source Code Structure

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

Table 3-5. GPIO Files

File	Description
mx51_pins.h	GPIO private header file
gpio.h	GPIO public header file
gpio.c	Function implementation

3.5.4 GPIO Programming Interface

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

3.6 EDIO

Not all platforms have the EDIO hardware module. This section applies only to those that do. The EDIO module provides external interrupt capability to the processors.

3.6.1 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 interrupt controller. This module can handle eight such interrupts simultaneously with selectable configurations for each incoming signal reaching EDIO.

3.6.2 EDIO Software Operation

The EDIO interrupt has been integrated into the generic platform level interrupt implementation as in `irq.c` in the `<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc` directory. For drivers that need to set up the

interrupt attributes, such as interrupt edges or levels, the `set_irq_type()` can be called. The interrupt clearing that is needed for the EDIO interrupts is hidden from the driver.

3.6.3 EDIO Features

The EDIO module controls the EDIO interrupt attributes provided by the hardware.

3.6.4 EDIO Source Code Structure

All of the EDIO module source code is in the files below in the `<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/include/mach` and `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-imx` directories.

Table 3-6. EDIO Files

File	Description
mx51.h	Header files for the IO module physical addresses
irq.c	Common functions for various boards

3.6.5 EDIO Programming Interface

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

3.7 SPBA Bus Arbiter

Not all platforms have the SPBA hardware module. Therefore, this section only applies to the platforms with SPBA module in them. The SPBA bus arbiter provides arbitration mechanism among multiple masters to have access to the shared peripherals.

3.7.1 SPBA Hardware Operation

The SPBA is a three-to-one 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 Kbytes 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)

3.7.2 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.

3.7.3 SPBA Features

This SPBA implementation supports the following features:

- Provides an API to allow different masters to take/release ownership of a shared peripheral

3.7.4 SPBA Source Code Structure

All of the SPBA module source code is in the MSL layer. The following files are available in the `<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/include/mach` and `<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc` directories:

Table 3-7. SPBA Files

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

3.7.5 SPBA Programming Interface

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

Chapter 4
Smart Direct Memory Access (SDMA) API

4.1 Overview

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

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

4.2 Hardware Operation

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

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

4.3 Software Operation

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

Complete support for SDMA is provided in three layers (see Figure 4-1):

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

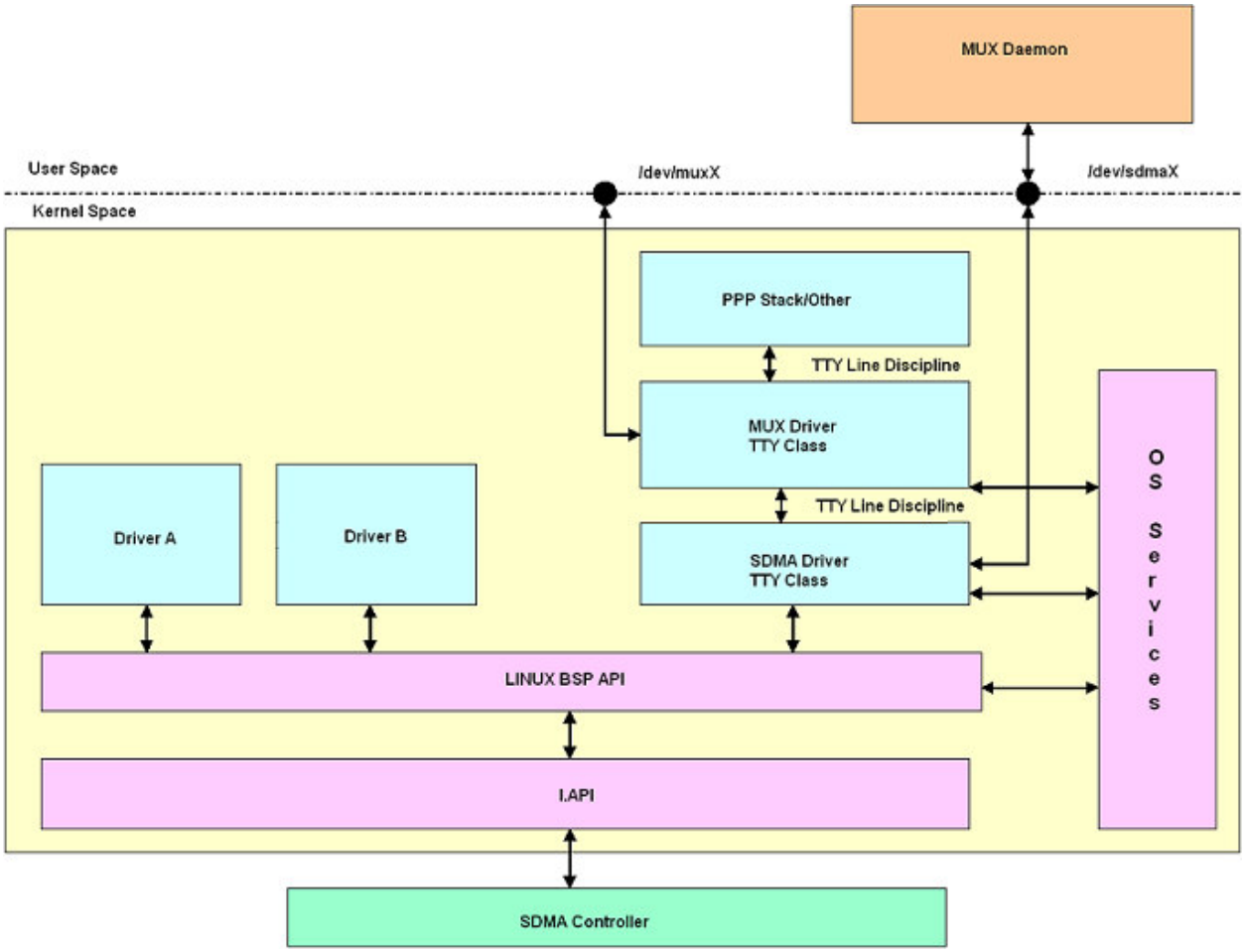


Figure 4-1. SDMA Block Diagram

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

Table 4-1 provides a list of drivers that use SDMA and the number of SDMA physical channels used by each driver. A driver can specify the SDMA channel number that it wishes to use (static channel allocation) or can have the SDMA driver provide a free SDMA channel for the driver to use (dynamic channel

allocation). For dynamic channel allocation, the list of SDMA channels is scanned from channel 32 to channel 1. On finding a free channel, that channel is allocated for the requested DMA transfers.

Table 4-1. SDMA Channel Usage

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

4.4 Source Code Structure

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

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

Table 4-2. SDMA API Source Files

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

Table 4-3 shows the header files available in the directory, `/<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51/.`

Table 4-3. SDMA Script Files

File	Description
sdma_script_code.h	SDMA RAM scripts for i.MX51

4.5 Menu Configuration Options

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

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

  System type > Freescale MXC implementations > MX51 Options > Use SDMA API.

  By default, this option is Y.

4.6 Programming Interface

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

4.7 Usage Example

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

Chapter 5
MC13892 Regulator Driver

The MC13892 regulator driver provides the low-level control of the power supply regulators, selection of voltage levels, and enabling/disabling of regulators. This device driver makes use of the PMIC protocol driver to access the PMIC hardware control registers.

5.1 Hardware Operation

The MC13892 provides reference and supply voltages for the application processor as well as peripheral devices. Four buck (step down) converters and two boost (step up) converters are included. The buck converters provide the power supply to processor cores and to other low voltage circuits such as I/O and memory. Dynamic voltage scaling is provided to allow controlled supply rail adjustments for the processor cores and/or other circuitry. Two DVS control pins are provided for pin controlled DVS on the buck switchers targeted for processor core supplies.

Linear regulators are directly supplied from the battery or from the switchers and include supplies for I/O and peripherals, audio, camera, BT, WLAN, and so on. Naming conventions are suggestive of typical or possible use case applications, but the switchers and regulators may be utilized for other system power requirements within the guidelines of specified capabilities. General Purpose Outputs (GPO) can be used for enabling external functions or supplies, thermistor biasing, and/or a muxed ADC input.

5.2 Driver Features

The MC13892 PMIC regulator driver is based on the PMIC protocol driver and regulator core driver. It provides the following services for regulator control of the PMIC component:

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

5.3 Software Operation

The PMIC power management driver and the MC13892 PMIC regulator client driver perform operations by reconfiguring the PMIC hardware control registers. This is done by calling protocol driver APIs with the required register settings.

Some of the PMIC power management operations depend on the system design and configuration. For example, if the system is powered by a power source other than the PMIC, then turning off or adjusting the PMIC voltage regulators has no effect. Conversely, if the system is powered by the PMIC, then any

changes that use the power management driver and the regulator client driver can affect the operation or stability of the entire system.

5.4 Regulator APIs

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

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

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

- regulator_get—lookup and obtain a reference to a regulator

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

- regulator_put—free the regulator source

  ```
  void regulator_put(struct regulator *regulator, struct device *dev);
  ```

- regulator_enable—enable regulator output

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

- regulator_disable—disable regulator output

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

- regulator_is_enabled—is the regulator output enabled

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

- regulator_set_voltage—set regulator output voltage

  ```
  int regulator_set_voltage(struct regulator *regulator, int uV);
  ```

- regulator_get_voltage—get regulator output voltage

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

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

5.5 Driver Architecture

Figure 5-1 shows the basic architecture of the MC13892 regulator driver.

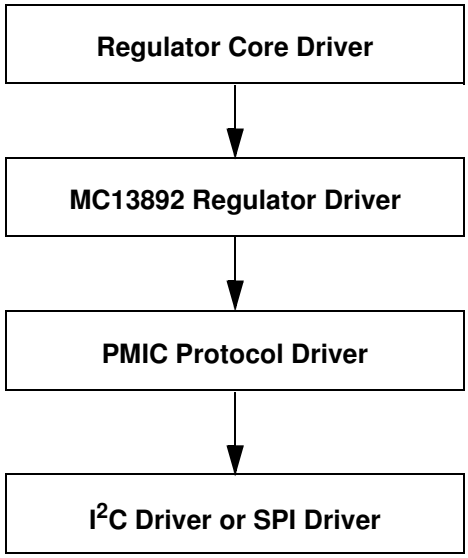


Figure 5-1. MC13892 Regulator Driver Architecture

5.6 Driver Interface Details

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

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

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

5.7 Source Code Structure

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

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

Table 5-1. MC13892 Power Management Driver Files

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

5.8 Menu Configuration Options

The following Linux kernel configurations are provided for the MC13892 Regulator driver. To get to the PMIC power configuration, use the command `./ltib -c` when located in the `<ltib dir>`. On the configuration screen select **Configure Kernel**, exit, and when the next screen appears, choose.

- Device Drivers > Voltage and Current regulator support > MC13892 Regulator Support.

Chapter 6
MC13892 RTC Driver

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

6.1 Driver Features

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

6.2 Software Operation

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

6.3 Driver Implementation Details

Configuring the MC13892 RTC driver includes the following parameters:

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

6.3.1 Driver Access and Control

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

6.4 Source Code Structure

Table 6-1 lists the source files for MC13892 RTC driver that are available in the `<ltib_dir>/rpm/BUILD/linux/drivers/rtc` directory.

Table 6-1. MC9S08DZ60 RTC Driver Files

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

6.5 Menu Configuration Options

The following Linux kernel configurations are provided for this module. To get to the MC13892 RTC configuration, use the command ./ltib -c when located in the `<ltib dir>`. In the screen, select **Configure Kernel**, exit, and a new screen appears.

- Device Drivers > Realtime Clock > Freescale MC13892 Real Time Clock.

Chapter 7
MC13892 Digitizer Driver

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

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

- Sampling the voltages on the touch panel interfaces to obtain the (X,Y) position and pressure measurements
- Battery voltage level monitoring
- Measurement of the voltage on the USB ID line to differentiate between mini-A and mini-B plugs

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

The PMIC ADC has a 10-bit resolution and supports either a single channel conversion or automatic conversion of all input channels in succession. The conversion can also be triggered by issuing a command or by detecting the rising edge on a special signal line.

A hardware interrupt can be generated following the completion of an ADC conversion. A hardware interrupt can also be generated if the ADC conversion results are outside of previously defined high and low level thresholds. Some PMIC chips also provide a pulse generator that is synchronized with the ADC conversion. The pulse generator can enable or drive external circuits in support of the ADC conversion process.

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

7.1 Driver Features

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

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

- Selects either a single ADC input channel or an entire group of input channels to be converted
- Specifies high and low level thresholds for each ADC conversion
- Starts an ADC conversion by issuing the appropriate start conversion command
- Starts an ADC conversion immediately following the rising edge of the ADTRIG input line or after a predefined delay following the rising edge
- Enable/disables hardware interrupts for all ADC-related event notifications
- Provides an interrupt handler routine that receives and properly handles all ADC end-of-conversion or exceeded high/low level threshold event notifications
- Other device drivers register/deregister additional callback functions to provide custom handling of all ADC-related event notifications
- Provides a read-only device interface for passing touchpanel (X,Y) coordinates and pressure measurements to applications
- Provides the ability to read out one or more ADC conversion results
- Implements the appropriate input scaling equations so that the ADC results are correct
- Specifies the delay between successive ADC conversion operations, if supported by the PMIC. For PMIC chips that do not support this feature, the device driver returns a NOT_SUPPORTED status
- Provides support for a pulse generator that is synchronized with the ADC conversion. For PMIC chips that do not support this feature, returns a NOT_SUPPORTED status
- Provides a complete IOCTL interface to initiate an ADC conversion operation and to return the conversion results
- Provides support for a polling method to detect when the ADC conversion has been completed

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

7.2 Software Operation

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

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

The PMIC ADC conversion can take a significant amount of time. The delay between a start of conversion request and a conversion completed event may even be open ended, if the conversion is not started until the appropriate external trigger signal is received. Therefore, all ADC conversion requests must be placed

in a wait queue until the conversion is complete. Once the ADC conversion has completed, the calling thread can be removed from the wait queue and reawakened.

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

7.3 Source Code Structure

Table 7-1 lists the source files for the MC13892-specific version of this driver. These are contained in the following directories:

```
<ltib_dir>/rpm/BUILD/linux/drivers/mxc/pmic/MC13892/pmic_adc.c
<ltib_dir>/rpm/BUILD/linux/include/linux/pmic_adc.h
<ltib_dir>/rpm/BUILD/linux/drivers/input/touchscreen/mxc_ts.c
```

Table 7-1. MC13892 Digitizer Driver Files

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

7.4 Menu Configuration Options

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

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

  Device Drivers > MXC Support Drivers > MXC PMIC Support > MC13892 ADC support
- Driver for the MXC touch screen. In menuconfig, this option is available under:

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

Chapter 8
i.MX51 Low-level Power Management (PM) Driver

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

8.1 Hardware Operation

The i.MX51 supports four low power modes: RUN, WAIT, STOP, and LPSR (low power screen).

Table 8-1 lists the detailed clock information for the different low power modes.

Table 8-1. Low Power Modes

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

For the detailed information about lower power modes, see the *MCIMX51 Multimedia Applications Processor Reference Manual* (MCIMX51RM).

8.2 Software Operation

The i.MX51 PM driver maps the low-power modes to the kernel power management states as listed below:

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

The i.MX51 PM driver performs the following steps to enter and exit low power mode:

1. Enable the gpc_dvfs_clk
2. Allow the Coretex-A8 platform to issue a deep sleep mode request
3. If STOP mode:
   a) Program CCM CLPCR register to request standby voltage change and power down on-chip oscillator
   b) Request switching off ARM/NENO power when pdn_req is asserted

c) Request switching off embedded memory peripheral power when pdn_req is asserted
d) Program TZIC wakeup register to set wakeup interrupts

4. Call `cpu_do_idle` to execute WFI pending instructions for wait mode
5. If STOP mode, call `cpu_do_suspend_workaround` to execute WFI pending instructions for stop mode
6. Generate a wakeup interrupt and exit low power mode
7. Disable gpc_dvfs_clk

8.3 Source Code Structure

In STOP mode, the i.MX51 can assert the VSTBY signal to the PMIC and request a voltage change. The Machine Specific Layer (MSL) usually sets the standby voltage in STOP mode according to i.MX51 data sheet.

Table 8-2 shows the PM driver source files. These files are available in `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51/`

Table 8-2. PM Driver Files

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

8.4 Menu Configuration Options

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

- CONFIG_PM—Build support for power management. In menuconfig, this option is available under

  Power management options > Power Management support

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

  Power management options > Suspend to RAM and standby

8.5 Programming Interface

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

Chapter 9
CPU Frequency Scaling (CPUFREQ) Driver

The CPU frequency scaling device driver allows the clock speed of the CPU to be changed on the fly. Once the CPU frequency is changed, the VDDGP voltage is changed to the voltage value defined in `cpu_wp_auto`. This method can reduce power consumption (thus saving battery power), because the CPU uses less power as the clock speed is reduced.

9.1 Software Operation

The CPUFREQ device driver is designed to change the CPU frequency and voltage on the fly. If the frequency is not defined in `cpu_wp_auto`, the CPUFREQ driver changes the CPU frequency to the nearest frequency in the array. The frequencies are manipulated using the clock framework API, while the voltage is set using the regulators API. The CPU frequencies in the array are based on the boot CPU frequency which can be changed by using the clock command in RedBoot.

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

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

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

To change the CPU frequency to a value that is given by using the command above (for example, to 800 MHz) use this command:

```
echo 800000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

The frequency 800000 is in KHz, which is 800 MHz.

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

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

9.2 Source Code Structure

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

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

Table 9-1. CPUFREQ Driver Files

File	Description
cpufreq.c	CPUFREQ functions

9.3 Menu Configuration Options

The following Linux kernel configuration is provided for this module:

- CONFIG_CPU__FREQ—In menuconfig, this option is located under
  CPU Power Management > CPU Frequency scaling
  The following options can be selected:
  - CPU Frequency scaling
  - CPU frequency translation statistics
  - Default CPU frequency governor (userspace)
  - Performance governor
  - Powersave governor
  - Userspace governor for userspace frequency scaling
  - On-demand CPU frequency policy governor
  - Conservative CPU frequency governor
  - CPU frequency driver for i.MX CPUs
  - CPU idle PM support

9.3.1 Board Configuration Options

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

Chapter 10
Dynamic Voltage Frequency Scaling (DVFS) Driver

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

10.1 Hardware Operation

The DVFS core module is a power management module. The purpose of the DVFS module is to detect the appropriate operation frequency for the IC. DVFS core is operated under control of the GPC (General Power Controller) block. The hardware DVFS core is served by GPC IRQ. The DVFS core domain performance update procedure includes both voltage and frequency changes in appropriate order by the GPC controller (hardware). The GPC controller performs updates of the supply voltage and the domain frequency in a fully automated way.

10.2 Software Operation

The DVFS device driver allows the frequency of the core clock domain and the voltage of the core power domain to be changed on the fly. The frequency of the core clock domain and the voltage of the core power domain are changed by switching between defined freq-voltage operating points. The frequencies are manipulated using the clock framework API, while the voltage is set using the regulators API. The CPU frequencies in the array are based on the boot CPU frequency which can be changed using the clock command in RedBoot.

To Enable the DVFS core use this command:

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

To Disable The DVFS core use this command:

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

10.3 Source Code Structure

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

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

Table 10-1. DVFS Driver Files

File	Description
dvfs_core.c	Linux DVFS functions

10.4 Menu Configuration Options

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

10.4.1 Board Configuration Options

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

Chapter 11
Image Processing Unit (IPU) Drivers

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

Typical logical channels include:

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

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

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

— Set sensor clock
— Set capture size

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

11.1 Hardware Operation

The detailed hardware operation of the IPU is discussed in the *MCIMX51 Multimedia Applications Processor Reference Manual (MCIMX51RM)*. Figure 11-2 shows the IPU hardware modules.

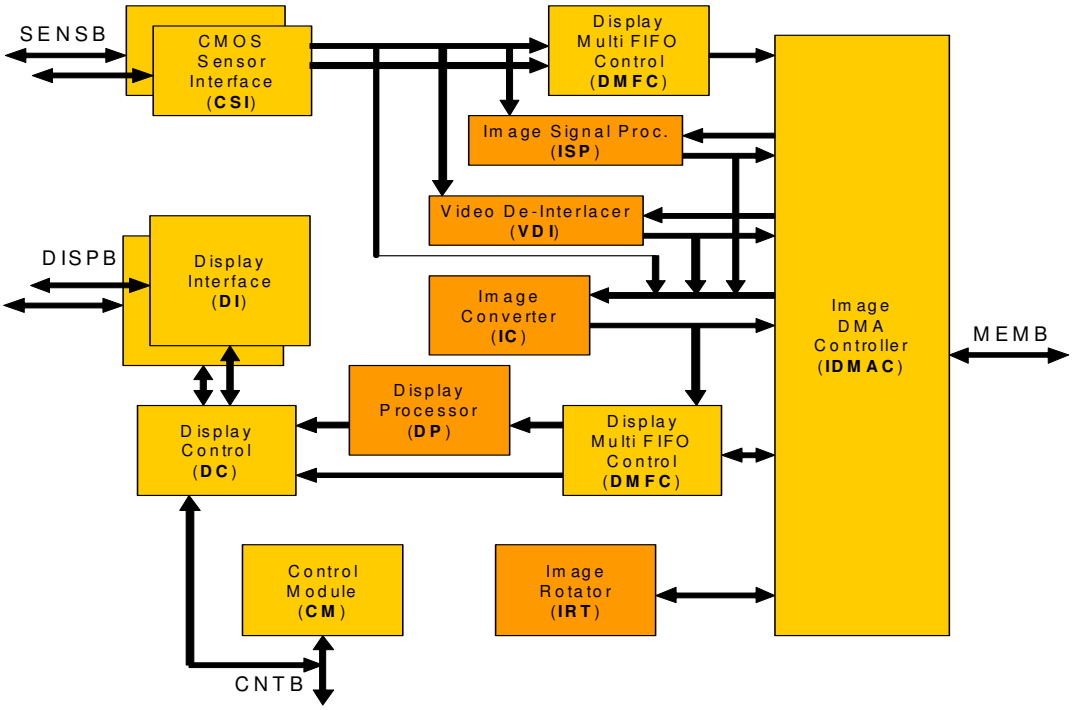


Figure 11-1. IPU Module Overview

11.2 Software Operation

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

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

Figure 11-2 shows the interaction between the different graphics/video drivers and the IPU.

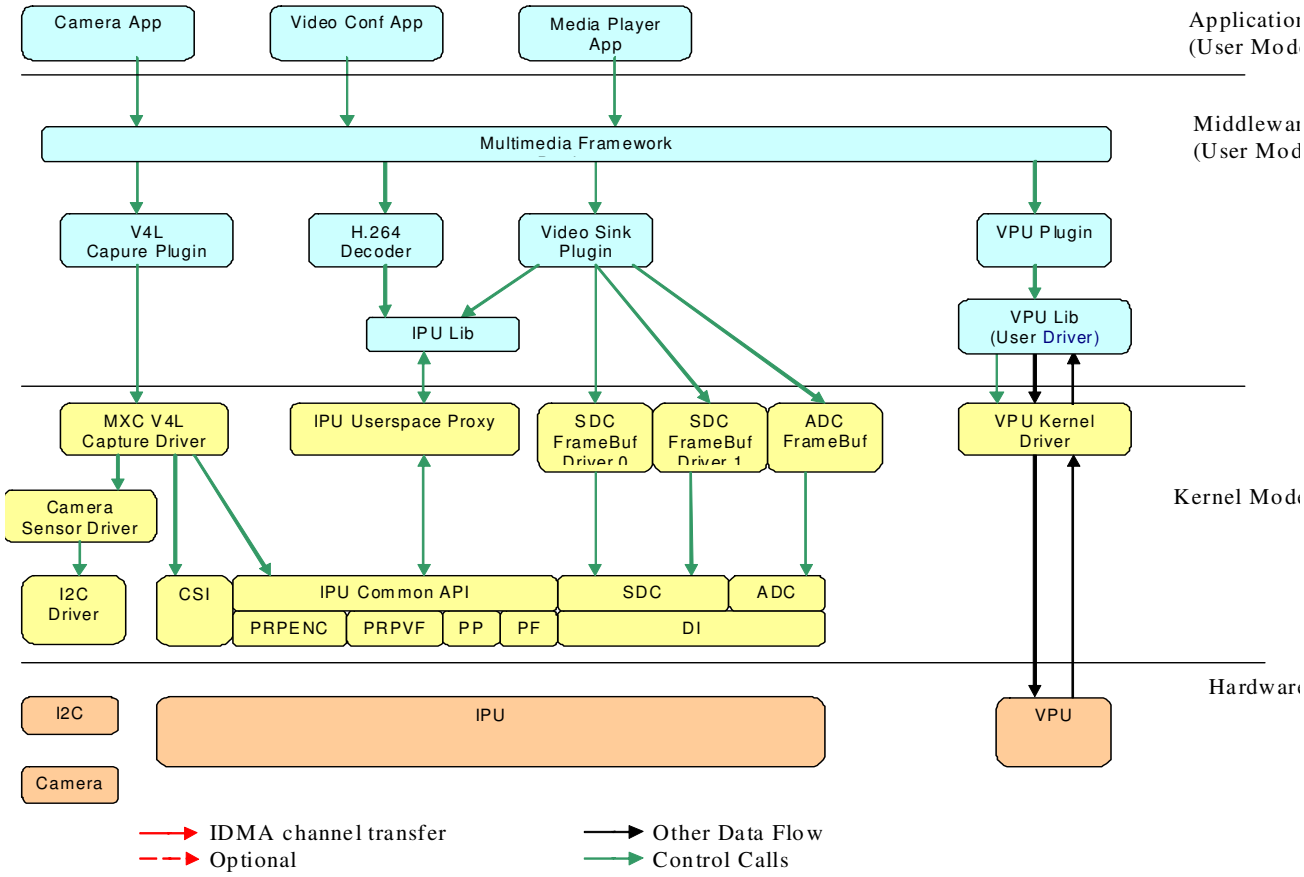


Figure 11-2. Graphics/Video Drivers Software Interaction

The IPU drivers are sub-divided as follows:

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

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

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

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

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

  `<ltib_dir>/rpm/BUILD/linux/drivers/mxc/ipu3`

  directory of the Linux kernel.

11.2.1 IPU Frame Buffer Drivers Overview

The frame buffer device provides an abstraction for the graphics hardware. It represents the frame buffer video hardware, and allows application software to access the graphics hardware through a well-defined interface, so that the software is not required to know anything about the low-level hardware registers.

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

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

11.2.1.1 IPU Frame Buffer Hardware Operation

The frame buffer interacts with the IPU hardware driver module.

11.2.1.2 IPU Frame Buffer Software Operation

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

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

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

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

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

11.2.1.3 Synchronous Frame Buffer Driver

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

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

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

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

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

The following features are supported:

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

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

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

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

11.3 Source Code Structure

Table 11-1 lists the source files associated with the IPU, Sensor, V4L2 and Panel drivers. These files are available in the following directories:

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

Table 11-1. IPU Driver Files

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

Table 11-2 lists the global header files associated with the IPU and Panel drivers. These files are available in the following directories:

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

Table 11-2. IPU Global Header Files

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

11.4 Menu Configuration Options

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

- CONFIG_MXC_IPU—Includes support for the Image Processing Unit. In menuconfig, this option is available under:

Device Drivers > MXC support drivers > Image Processing Unit Driver

By default, this option is Y for all architectures.

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

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

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

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

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

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

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

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

  Use case driver for dumb sensor or

  CSI > IC(PRP VF) > MEM

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

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

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

  Use case driver for dumb sensors

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

  or for smart sensors

  CSI > IC(PRP ENC) > MEM.

  In menuconfig, this option is available under:

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

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

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

  VIDEO_DEV && MXC_IPU && MXC_IPU_PRP_VF_SDC && MXC_IPU_PRP_ENC

  In menuconfig, this option is available under:

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

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

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

  By default, this option is Y for all.
- CONFIG_FB—This is the configuration option to include frame buffer support in the Linux kernel. In menuconfig, this option is available under:

  Device Drivers > Graphics support > Support for frame buffer devices

  By default, this option is Y for all architectures.
- CONFIG_FB_MXC—This is the configuration option for the MXC Frame buffer driver. This option is dependent on the CONFIG_FB option. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support

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

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

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

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

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > CLAA WVGA Panel.
- CONFIG_FB_MXC_TVOUT_TVE —This is the configuration option that chooses the MXC TVOUT Encoder. This option is dependent on CONFIG_FB_MXC_SYNC_PANEL and CONFIG_MXC_IPU_V3 options. In menuconfig, this option is available under:

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

- CONFIG_FB_MXC_TVOUT_CH7024 —This configuration option selects the CH7024 TVOUT encoder. This option is dependent on the CONFIG_FB_MXC_SYNC_PANEL option. In menuconfig, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > CH7024 TV Out Encoder
- CONFIG_FB_MXC_TVOUT —This configuration option selects the FS453 TVOUT encoder. This option is dependent on CONFIG_FB_MXC_SYNC_PANEL option. In menuconfig, this option is available under:

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

11.5 Programming Interface

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

Chapter 12
Video for Linux Two (V4L2) Driver

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

The features supported by the V4L2 driver are as follows:

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

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

12.1 V4L2 Capture Device

The V4L2 capture device includes two interfaces:

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

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

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

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

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

12.1.1 V4L2 Capture IOCTLs

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

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

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

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

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

Figure 12-1 shows a block diagram of V4L2 Capture API interaction.

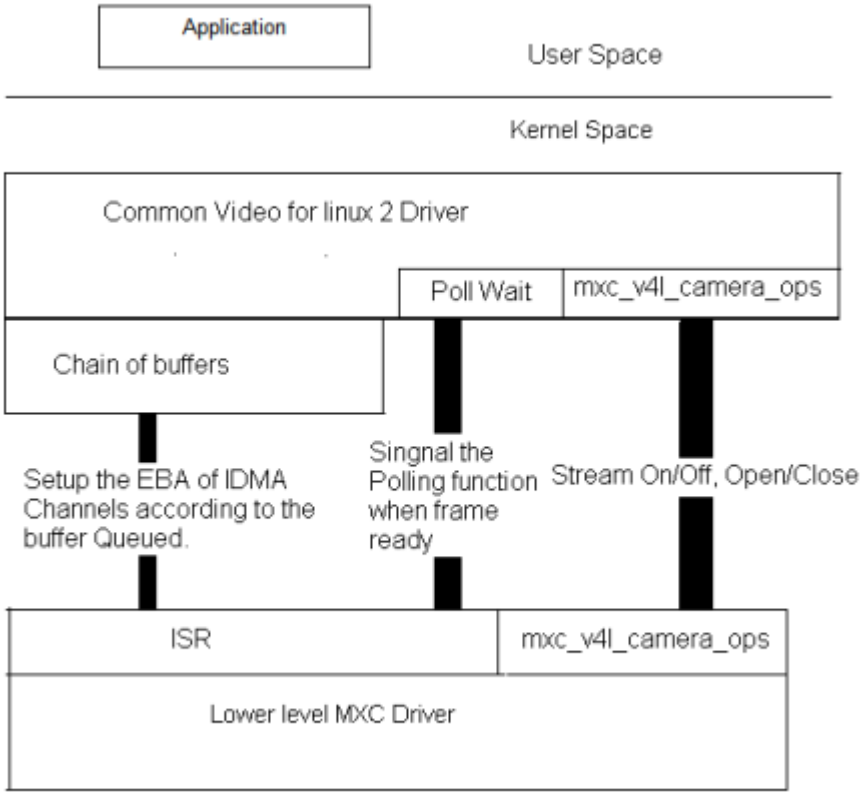


Figure 12-1. Video4Linux2 Capture API Interaction

12.1.2 Use of the V4L2 Capture APIs

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

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

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

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

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

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

12.2 V4L2 Output Device

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

12.2.1 V4L2 Output IOCTLs

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

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

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

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

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

12.2.2 Use of the V4L2 Output APIs

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

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

12.3 Source Code Structure

Table 12-1 lists the source and header files associated with the V4L2 drivers. These files are available in the following directory:

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

Table 12-1. V2L2 Driver Files

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

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

12.4 Menu Configuration Options

The Linux kernel configuration options are provided in the chapter on the IPU module. See Section 11.4, "Menu Configuration Options."

12.5 V4L2 Programming Interface

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

Chapter 13
TV Encoder (TVE) Driver

The TV Encoder (TVE) is designed to provide a direct connection between an Application Processor (AP) and a TV set by analog interfaces. The TV Encoder supports different Standard Definition (SD) and High Definition (HD) television standards. The module is based on mixed (digital and analog) signal processing. It includes three main components:

- TV Signal Processor (TVSP)
- Triple Video Digital-to-Analog Converter (TVDAC)
- Cable Detection Circuit (CDC)

13.1 TVE Driver Overview

The TVE takes in digital graphics input, which is the output of one of the two Image Processing Unit (IPU) Display Interfaces (DI), and converts it to TV output. The TV-out driver implements a frame buffer driver in the Linux kernel. The frame buffer driver implements IPU DI configurations and the digital graphics input to TVE. The driver is enabled by selecting the TV-out option under the graphics parameters in the kernel configuration. The TVE driver is a Frame buffer driver.

13.2 Driver Features

The TV Encoder supports different Standard Definition (SD) and High Definition (HD) television standards. The driver supports:

- SD mode
  - — TV standards
    - – NTSC
    - – PAL B,D,G,H, I/M/N
  - — Output formats
    - – Composite video (CVBS)
    - – S-video (Y/C), composite video and S-video signals may be output simultaneously
    - – YPrPb component
    - – RGB component
  - — Programmable notch/pre-comb filter for CVBS
  - — Switchable pedestal
  - — Copy Generation Management System (CGMS) support according to
    - – EIA-608b
    - – IEC 61880-1

  - EIA-J CPR-1204-1
- Output oversampling up to ×16 for elimination of external analog filters
- HD mode
  - TV standards
    - 720p 60 Hz
  - Output formats
    - YPrPb component
    - RGB component
  - Output oversampling up to ×4 for elimination of external analog filters

13.3 Hardware Operation

The TVE receives a 30-bit video data stream from the IPU. In functional mode, 24-bits are used to transfer the video data in the YCbCr 4:2:2 or YCbCr 4:4:4 formats. Each of the YCrCb components is represented by an 8-bit word. For some TVE configurations, the TVE performs inter-field vertical filtering for de-flickering, fine sharpening and high-frequency noise reduction.

An output voltage of each TVE channel is monitored by the Cable Detection (CD) system. The CD is able to distinguish between the following cases:

- TVE directly drives a 75 Ω load
- Cable is disconnected
- Double terminated cable
- Output is shorted to the ground

13.4 Software Operation

The driver implements the TVE the following output format configurations:

- NTSC
- PAL
- 720p 60

Setting the TV can be done by writing the frame buffer mode (for example: U:720x480i-60 > /sys/class/graphics/fb1/mode). The driver implements cable detect on standby mode and on the fly detection to allow quality improvement when possible.

13.5 Source Code Structure

Table 13-1 describes the source files associated with the TVE driver, which are available in the directory `<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc/.`

Table 13-1. TV-Out Driver Files

File	Description
tve.c	Source file for TVE TV-Out driver

Table 13-2 describes the source files associated with the frame buffer drivers which use TVE driver are available in the directory `<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc/`.

Table 13-2. Frame Buffer Driver Files

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

13.6 Menu Configuration Options

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

- Device Drivers > Graphics support > MXC TVE TV Out Encoder

Chapter 14
Video Processing Unit (VPU) Driver

The Video Processing Unit (VPU) is a high performance multi-standard video processing unit which can perform H.264 BP/MP/HP, VC-1 SP/MP/AP, MPEG-4 SP/ASP, MPEG-2 MP, MJPEG BP decoding and H.264 BP, MPEG-4 SP and MJPEG BP encoding as a single IP. It supports a full duplex video codec with approximately 30 fps at D1 image resolution, multi-party call, and integrates multiple video processing standards together.

The VPU driver supports the following multimedia video stream processing features:

- Multi-standard video codec
  - — Decoding
    - – MPEG-4 simple and advanced simple profiles except GMC
    - – H.264 baseline, main and high profiles
    - – H.263 profile 3
    - – VC-1 simple, main and advanced profiles
    - – MPEG-2 main profile at high level
    - – MJPEG Baseline
  - — Encoding
    - – MPEG-4 simple profile
    - – H.264 baseline profile
    - – H.263 profile 3
    - – MJPEG Baseline
  - — Multi-format/multi-instance operation
    - – Decode up to four streams simultaneously
- Decoding tools
  - — H.264
    - – Fully compatible with ITU-T recommendation H.264 specification in BP/MP/HP
    - – Supports CABAC/CAVLC
    - – Variable block size (16×16, 16×8, 8×16, 8×8, 8×4, 4×8 and 4×4)
    - – Error detection, concealment and error resilience tools such as FMO and ASO
  - — VC-1
    - – Supports all VC-1 main profile features (SMPTE Proposed SMPTE Standard for Television: VC-1 Compressed Video Bitstream format and Decoding Process)
    - – Supports Simple/Main/Advanced Profile
    - – Multi-resolution (Dynamic resolution) is not processed inside of VPU

- MPEG-4
  - Supports simple/advanced simple profile except GMC
  - Supports H.263 baseline profile
  - Supports Xvid
- MPEG-2
  - Fully compatible with ISO/IEC 13182-2 MPEG2 specification in main Profile
  - Support I, P and B frame
  - Support field coded picture (interlaced) and frame coded picture
- MJPEG
  - Baseline ISO/IEC 10918-1 JPEG compliance
  - Supports JFIF 1.02 input format with up to three components
  - 8-bit samples for each component
  - Support up to 4:4:4 decoding
- Minimum decoding size is 16×16 pixels

- Encoding tools
  - [±32, ±16] 1/2 and 1/4-pel accuracy motion estimation
  - 16×16, 8×8, 4×4 block sizes are supported
  - Available block sizes can be configurable
  - The encoder uses only one reference frame for motion estimation
  - Unrestricted motion vector
  - MPEG-4 AC/DC prediction and H.264 Intra prediction
  - H.263 Annex J, K(RS = 0 and ASO = 0), and T
  - Error resilience tools
    - MPEG-4 re-synchronize marker and data-partitioning with RVLC (Fixed number of bits/macroblocks between macroblocks)
    - CIR (Cyclic Intra Refresh)/AIR (Adaptive Intra Refresh)
    - Bit-rate control (CBR & VBR)
  - MJPEG supports up to 4:2:2 format
  - Minimum encoding size is 32 pixels in horizontal and 16 pixels in vertical

14.1 Hardware Operation

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

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

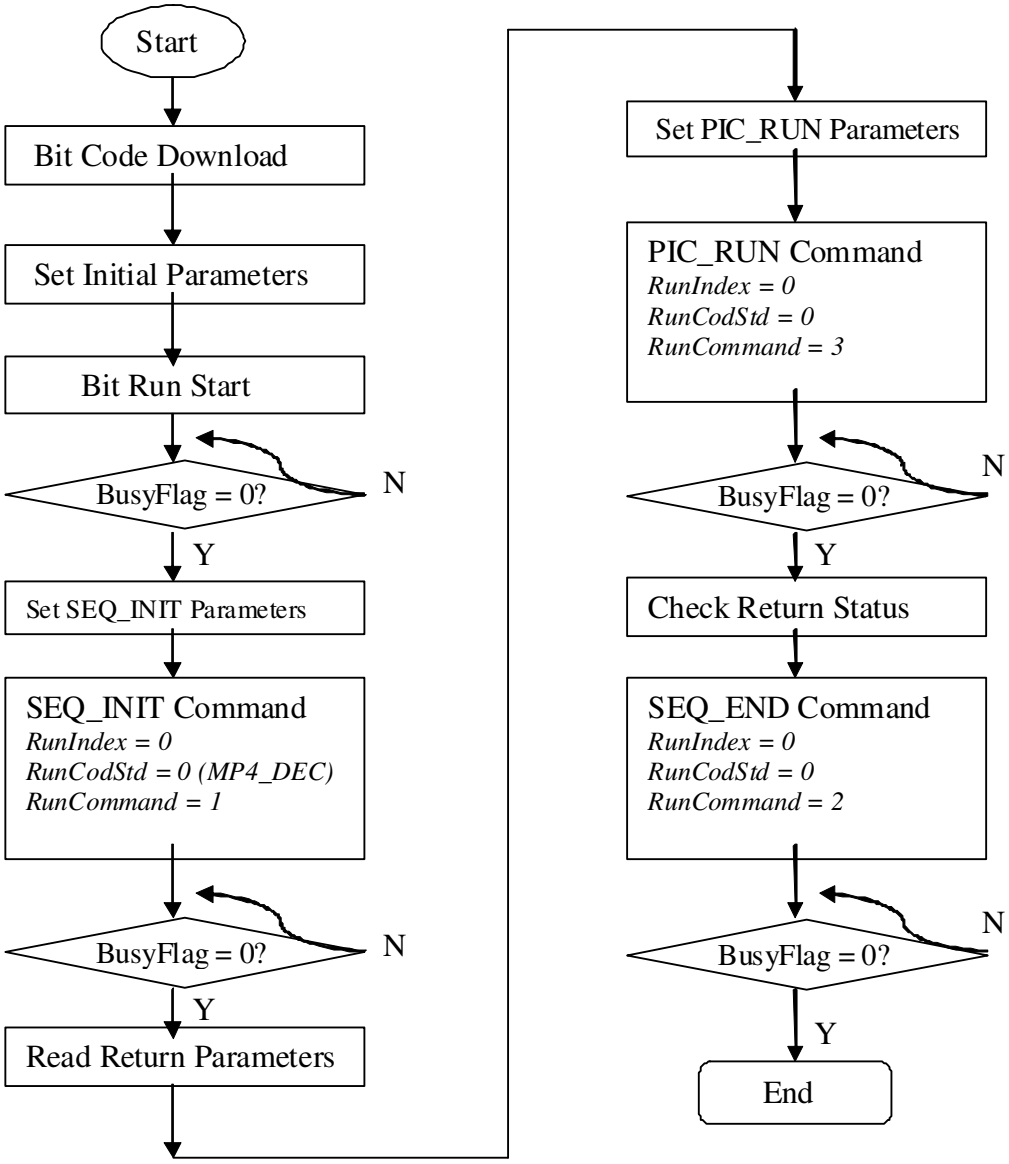


Figure 14-1. VPU Hardware Data Flow

14.2 Software Operation

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

The VPU kernel driver include the following functions:

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

The VPU user space driver has the following functions:

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

14.3 Source Code Structure

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

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

Table 14-1. VPU Driver Files

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

Table 14-2 lists the user-space library source files available in the `<ltib_dir>/rpm/BUILD/imx-lib-5.1.0/vpu` directory:

Table 14-2. VPU Library Files

File	Description
vpu_io.c	Interfaces with the kernel driver for opening the VPU device and allocating memory
vpu_io.h	Header file for IOCTLs
vpu_lib.c	Core codec implementation in user space
vpu_lib.h	Header file of the codec
vpu_reg.h	Register definition of VPU
vpu_util.c	File implementing common utilities used by the codec
vpu_util.h	Header file
vpu_fw/vpu_fw_imx51.bin	Binary code running on the DSP inside the VPU

NOTE

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

14.4 Menu Configuration Options

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

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

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

14.5 Programming Interface

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

The codec library APIs are listed below:

```
RetCode vpu_EncOpen(EncHandle* pHandle, EncOpenParam* pop);
RetCode vpu_EncClose(EncHandle encHandle);
RetCode vpu_EncGetInitialInfo(EncHandle encHandle, EncInitialInfo* initialInfo);
RetCode vpu_EncRegisterFrameBuffer(EncHandle encHandle, FrameBuffer* pBuffer, int num,
                                   int stride);
RetCode vpu_EncGetBitstreamBuffer(EncHandle handle, PhysicalAddress* prdPrt,
                                  PhysicalAddress* pwrPtr, Uint32* size);
RetCode vpu_EncUpdateBitstreamBuffer(EncHandle handle, Uint32 size);
RetCode vpu_EncStartOneFrame(EncHandle encHandle, EncParam* pParam);
RetCode vpu_EncGetOutputInfo(EncHandle encHandle, EncOutputInfo* info);
RetCode vpu_EncGiveCommand (EncHandle pHandle, CodecCommand cmd, void* pParam);
```

```
RetCode vpu_DecOpen(DecHandle* pHandle, DecOpenParam* pop);
RetCode vpu_DecClose(DecHandle decHandle);
RetCode vpu_DecGetBitstreamBuffer(DecHandle pHandle, PhysicalAddress* pRdptr,
                                  PhysicalAddress* pWrptr, Uint32* size);
RetCode vpu_DecUpdateBitstreamBuffer(DecHandle decHandle, Uint32 size);
RetCode vpu_DecSetEscSeqInit(DecHandle pHandle, int escape);
RetCode vpu_DecGetInitialInfo(DecHandle decHandle, DecInitialInfo* info);
RetCode vpu_DecRegisterFrameBuffer(DecHandle decHandle, FrameBuffer* pBuffer, int num,
                                  int stride, DecBufInfo* pBufInfo);
RetCode vpu_DecStartOneFrame(DecHandle handle, DecParam* param);
RetCode vpu_DecGetOutputInfo(DecHandle decHandle, DecOutputInfo* info);
RetCode vpu_DecBitBufferFlush(DecHandle handle);
RetCode vpu_DecClrDispFlag(DecHandle handle, int index);
RetCode vpu_DecGiveCommand(DecHandle pHandle, CodecCommand cmd, void* pParam);
```

System I/O operations are listed below:

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

14.6 Defining an Application

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

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

Chapter 15
Graphics Processing Unit (GPU)

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

15.1 Driver Features

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

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

15.2 Hardware Operation

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

15.3 Software Operation

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

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

15.4 Source Code Structure

Table 15-1 lists the modules and libraries associated with GPU.

Table 15-1. GPU Driver Files

File	Description	Location
libc2d.so libegl13.so libgles20.so libOpenVG.so libres.so libbb2d.so libcsi.so libgles11.so libgsl.so libpanel2.so	GPU related libraries that are part of the Linux BSP file system	/usr/lib/
gpu_z430.ko	Kernel level modules for the GPU driver and the display driver	/lib/modules/2.6.28-515-g4eec389/extra/

15.5 API References

Refer to the following web sites for detailed specifications:

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

15.6 Menu Configuration Options

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

- Package list > amd-gpu-bin-mx51

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

Chapter 16
ISL29003 Light Sensor Driver

The ISL29003 is an integrated light sensor with a 16-bit integrating type ADC, I^2C user programmable lux range select for optimized counts per lux, and I^2C multi-function control and monitoring capabilities. The internal ADC provides 16-bit resolution while rejecting 50Hz and 60Hz flicker caused by artificial light sources. It can be used in aspects such as ambient light sensing, backlight control and more.

16.1 ISL29003 Features

- Range select via I^2C
  - Range 1 = 0 lux to 1000 lux
  - Range 2 = 0 lux to 4000 lux
  - Range 3 = 0 lux to 16,000 lux
  - Range 4 = 0 lux to 64,000 lux
- Human eye response (540 nm peak sensitivity)
- Temperature compensated
- 16-bit resolution
- Adjustable resolution: up to 65 counts per lux
- User-programmable upper and lower threshold interrupt
- Simple output code, directly proportional to lux
- IR + UV rejection
- 50 Hz/60 Hz rejection
- 2.5 V to 3.3 V supply

16.2 Requirements

The ISL29003 driver is based on an I^2C driver and makes use of a hardware monitor system. The user must enable this support in the Linux kernel.

16.3 Software Architecture

Figure 16-1 shows the software architecture. At the driver initial phase, an I^2C client is registered to the I^2C system and is used during the process of ISL29003 operations. The ISL29003 registers itself to the hardware monitor system that provides user access facilities.

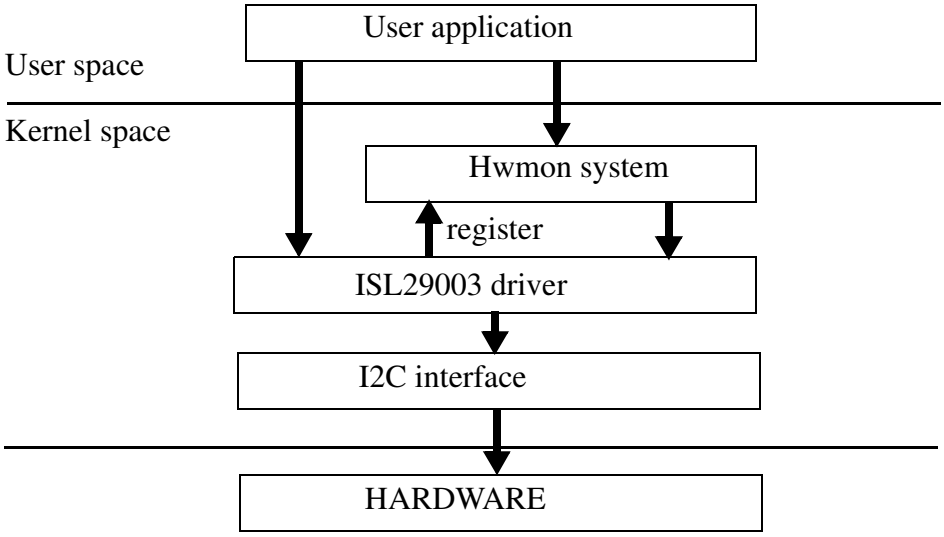


Figure 16-1. Driver Architecture

16.4 Source Code Structure

The driver source code structure contains only one file located in the directory:

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

Table 16-1. Driver Source Code Structure File

File	Description
isl29003.c	Implementation of the isl29003 light sensor driver.

16.5 Linux Menu Configuration

To enable the ISL29003 driver use the command `./ltib -c` when located in the <`ltib dir`>. On the screen displayed, select **Configure the kernel** and exit. When the next screen appears select the following option:

- Device Drivers > Hardware Monitoring support > ISL29003 Light Sensor

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

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

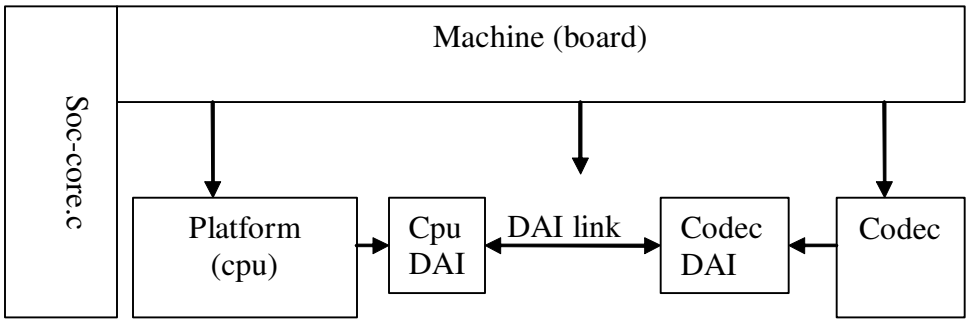


Figure 17-1. ALSA SoC Software Architecture

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

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

17.1 SoC Sound Card

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

NOTE

Only the Stereo Codec is supported on the i.MX51 platform.

17.1.1 Stereo Codec Features

The stereo codec supports the following features:

- Sample rates for playback and capture are 32 KHz, 44.1 KHz, 48 KHz, and 96 KHz
- Channels:
  - Playback: supports two channels. (stereo)
  - Capture: supports two channels. (Only one channel has valid voice data due to hardware connection)
- Audio formats:
  - Playback:
    - SNDRV_PCM_FMTBIT_S16_LE
    - SNDRV_PCM_FMTBIT_S20_3LE
    - SNDRV_PCM_FMTBIT_S24_LE
  - Capture:
    - SNDRV_PCM_FMTBIT_S16_LE
    - SNDRV_PCM_FMTBIT_S20_3LE
    - SNDRV_PCM_FMTBIT_S24_LE

17.1.2 Sound Card Information

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

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

17.2 ASoC Driver Source Architecture

As illustrated in Figure 17-2, `imx-pcm.c` is shared by the stereo ALSA SoC driver, the 5.1 ALSA SoC driver and the Bluetooth codec driver. This file is responsible for pre-allocating DMA buffers and managing DMA channels.

The stereo codec is connected to the CPU through the SSI interface. `imx-ssi.c` registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip SSI interface. `sgtl5000.c` registers the stereo codec and hifi DAI drivers. The direct hardware operations on the stereo codec are in `sgtl5000.c`.

`imx-3stack-sgtl5000.c` is the machine layer code which creates the driver device and registers the stereo sound card.

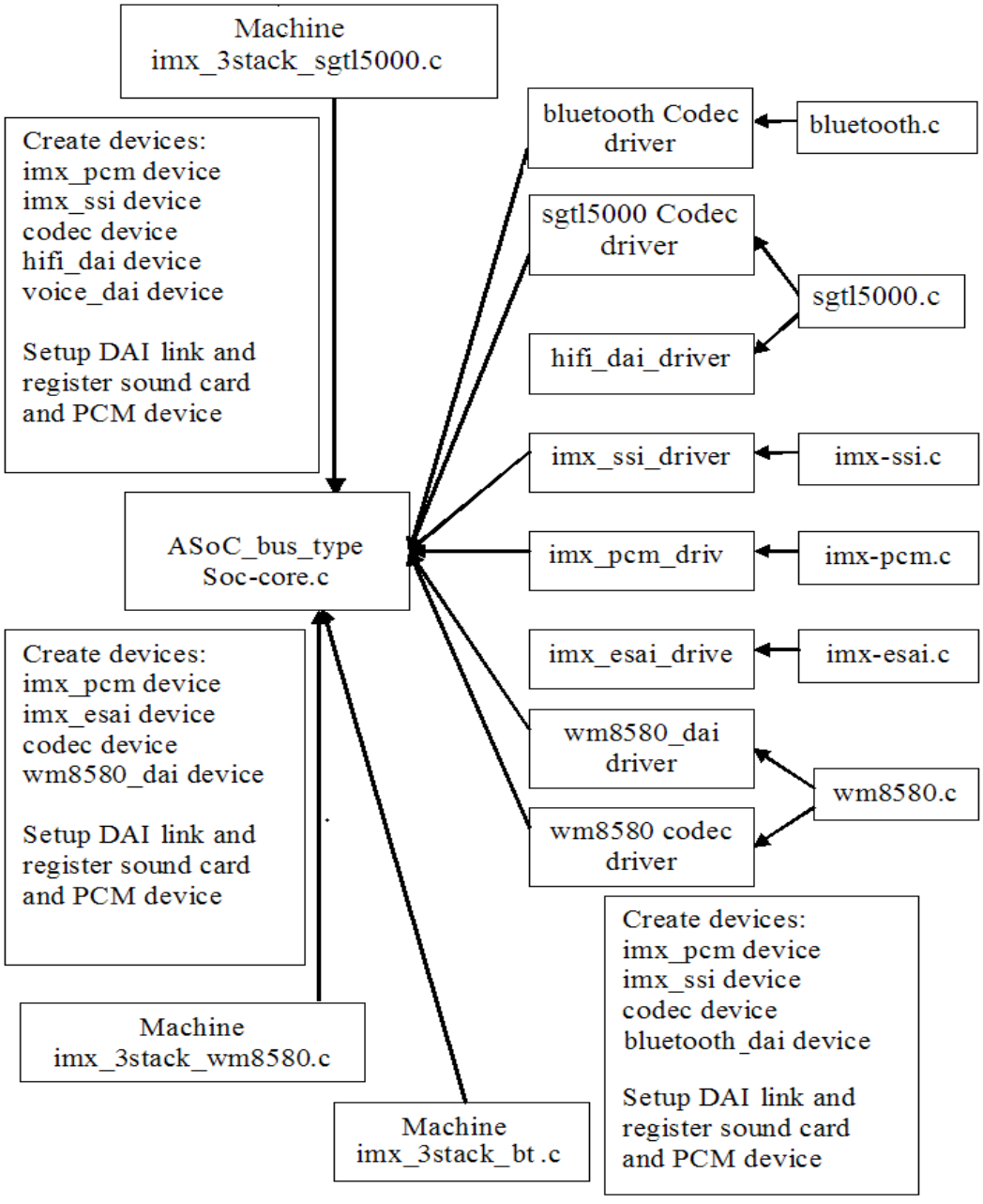


Figure 17-2. ALSA SoC Source File Relationship

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

Table 17-1. Stereo Codec SoC Driver Files

File	Description
imx/imx-3stack-sgtl5000.c	Machine layer for stereo codec ALSA SoC
imx/imx-pcm.c	Platform layer for stereo codec ALSA SoC
imx/imx-pcm.h	Header file for PCM driver and AUDMUX register definitions
imx/imx-ssi.c	Platform DAI link for stereo codec ALSA SoC
imx/imx-ssi.h	Header file for platform DAI link and SSI register definitions
codecs/sgtl5000.c	Codec layer for stereo codec ALSA SoC
codecs/sgtl5000.h	Header file for stereo codec driver

17.3 Menu Configuration Options

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

- SoC Audio support for i.MX SGTL5000. In menuconfig, this option is available under
  Device drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio for the Freescale i.MX CPU

17.4 Hardware Operation

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

17.4.1 Stereo Audio Codec

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

The SGTL5000 ASoC codec driver exports the audio record/playback/mixer APIs according to the ASoC architecture. Additionally, this driver provides audio-loop back function for the FM driver to enable stereo FM output. The ALSA related audio function and the FM loopback function cannot be performed simultaneously.

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

- Codec DAI and PCM configuration

- Codec control I/O—using I^2C
- Mixers and audio controls
- Codec audio operations
- DAC Digital mute control

The SGTL5000 codec is registered as an I^2C client when the module initializes. The APIs are exported to the upper layer by the structure `snd_soc_dai_ops`. The `io_probe` routine initializes the codec hardware to the desired state.

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

17.5 Software Operation

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

17.5.1 Sound Card Registration

The codecs have the same registration sequence:

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

17.5.2 Device Open

The ALSA driver:

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

After triggering for the first time, the subsequent DMA reads and writes are configured by the DMA callback.

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

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

18.1 S/PDIF Overview

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

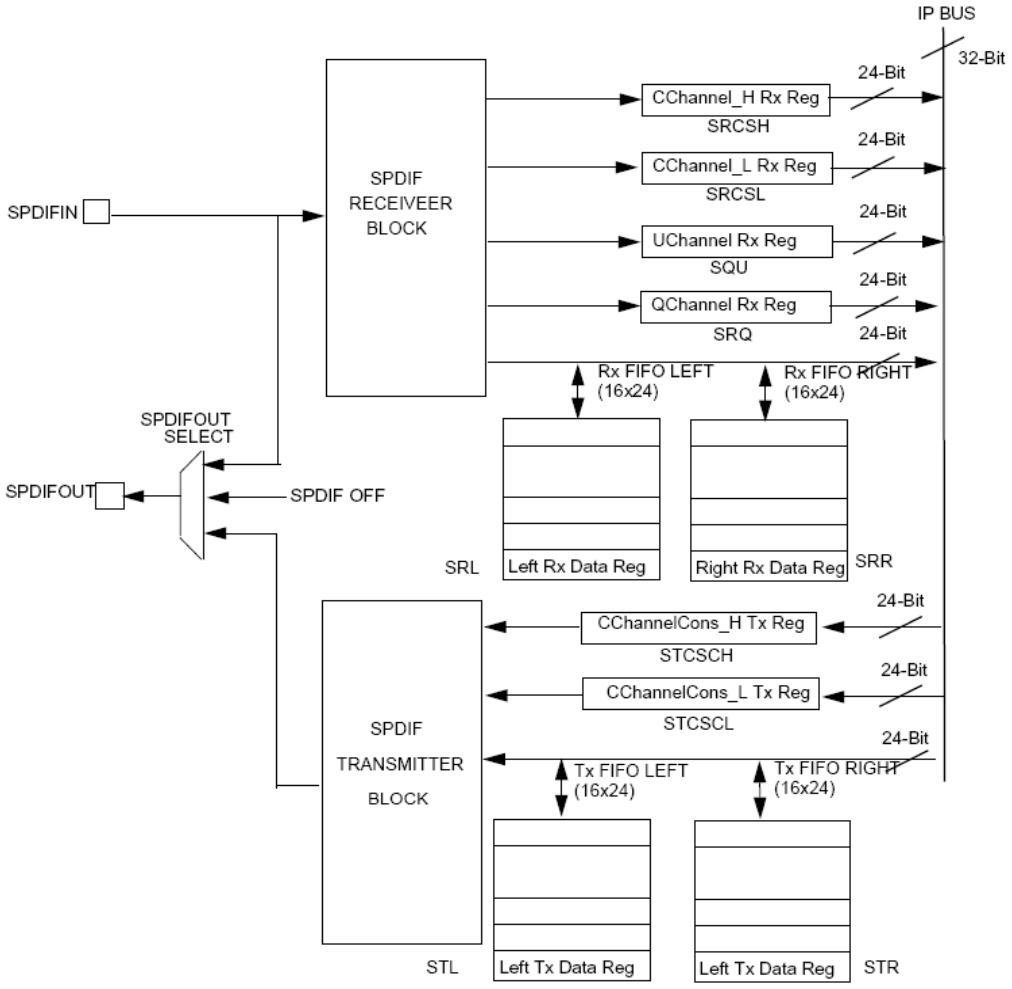


Figure 18-1. S/PDIF Transceiver Data Interface Block Diagram

18.1.1 Hardware Overview

The S/PDIF is composed of two parts:

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

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

18.1.2 Software Overview

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

18.2 S/PDIF Tx Driver

The S/PDIF Tx driver supports the following features:

- 32, 44.1 and 48 KHz sample rates
- Signed 16 and 24-bit little Endian sample format. Due to S/PDIF SDMA feature, the 24-bit output sample file must have 32-bits in one channel per frame, and only the 24 LSBs are valid

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

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

  ```
  #modprobe snd-spdif
  ```

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

  — Get card ID and name

  For example:

  ```
  #cat /proc/asound/cards
  0 [imx3stack      ]: SGTL5000 - imx-3stack
  ```

```
                imx_3stack (SGTL5000)
          1 [TXRX           ]: MXC_SPDIF - MXC SPDIF TX/RX
                MXC Freescale with SPDIF
```

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

— Get Playback PCM device info

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

- Software operation

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

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

NOTE

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

18.2.1 Driver Design

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

18.2.2 Provided User Interface

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

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

18.3 S/PDIF Rx Driver

The S/PDIF Rx driver supports the following features:

- 16, 32, 44.1, 48, 64 and 96 KHz receiving sample rate

- Signed 24-bit little endian sample format. Due to S/PDIF SDMA feature, each channel bit length in PCM recorded frame is 32 bits, and only the 24 LSBs are valid

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

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

  ```
  #modprobe snd-spdif
  ```

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

  — Get Card ID and name

  For example:

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

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

  — Get capture PCM device info

  ```
  #cat /proc/asound/TXRX/pcm[card id]/info
  ```
- Software operation

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

  ```
  #arecord -D "hw:2,0" -t wav -c 2 -r 48000 -f S24_LE record.wav
  ```

NOTE

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

18.3.1 Driver Design

Before the driver can read a data frame from the S/PDIF receiver FIFO, it must wait for the internal DPLL to be locked. Using the high speed system clock, the internal DPLL can extract the bit clock (advanced pulse) from the input bit stream. When this internal DPLL is locked, the LOCK bit of PhaseConfig Register is set and the driver configures the interrupt, clock and SDMA channel. After that, the driver can receive audio data, channel status, user bits and valid bits concurrently.

For channel status reception, a total of 48 channel status bits are received in two registers. The driver reads them out when a user application makes a request.

For user bits reception, there are two modes for User Channel reception: CD and non-CD. The mode is determined by the USyncMode (bit 1 of CDText_Control register). User can call the sound control interface to set the mode (see Figure 18-1), but no matter what the mode is, the driver handles the user bits in the same way. For the S/PDIF Rx, the hardware block copies the Q bits from the user bits to the QChannel registers and puts the user bits in UChannel registers. The driver allocates two queue buffers for both U bits and Q bits. The U bits queue buffer is 96×2 bytes in size, the Q bits queue buffer is 12×2 bytes

in size, and queue buffers are filled in the U/Q Full, Err and Sync interrupt handlers. This means that the user can get the previous ready U/Q bits while S/PDIF driver is reading new U/Q bits.

For valid bit reception, S/PDIF Rx hardware block triggers an interrupt and set interrupt status upon reception. A sound control interface is provided for the user to get the status of this valid bit.

18.3.2 Provided User Interface

The S/PDIF Rx driver provides interfaces for user application as shown in Table 18-1.

Table 18-1. S/PDIF Rx Driver Interfaces

Interface	Type	Mode[a]	Parameter	Comment
Common PCM	PCM	—	—	PCM open/close prepare/trigger hw_params/sw_params
Rx Sample Rate	Sound Control[b]	r	Integer Range: [16000, 96000]	Get sample rate. It is not accurate due to DPLL frequency measure module. So the user application must do a correction to the get value.
USyncMode	Sound Control	rw	Boolean Value: 0 or 1	Set 1 for CD mode Set 0 for non-CD mode
Channel Status	Sound Control	r	struct snd_aes_iec958 Only status [6] array member is used	—
User bit	Sound Control	r	Byte array 96 bytes for U bits 12 bytes for Q bits	—
No good V bit	Sound Control	r	Boolean Value: 0 or 1	An interrupt is associated with the valid flag. (interrupt 16 - SPDIFValNoGood). This interrupt is set every time a frame is seen on the SPDIF interface with the valid bit set to invalid.

[a] The mode column shows the interface attribute: r (read) or w (write)

[b] The sound control type of interface is called by the snd_ctl_xxx() alsa-lib function

The user application can follow the program flow from Figure 18-2 to use the S/PDIF Rx driver. First the application opens the S/PDIF Rx PCM device, waits for the DPLL to lock the input bit stream, and gets the input sample rate. If the USyncMode needs to be set, set it before reading the U/Q bits. Next, set the hardware parameters, including channel number, format and capture sample rate which is obtained from the driver. Then call prepare and trigger to startup S/PDIF Rx stream read. Finally call the read function

to get the data. During the reading process, applications can read the U/Q bits and channel status from the driver and valid the no good bit.

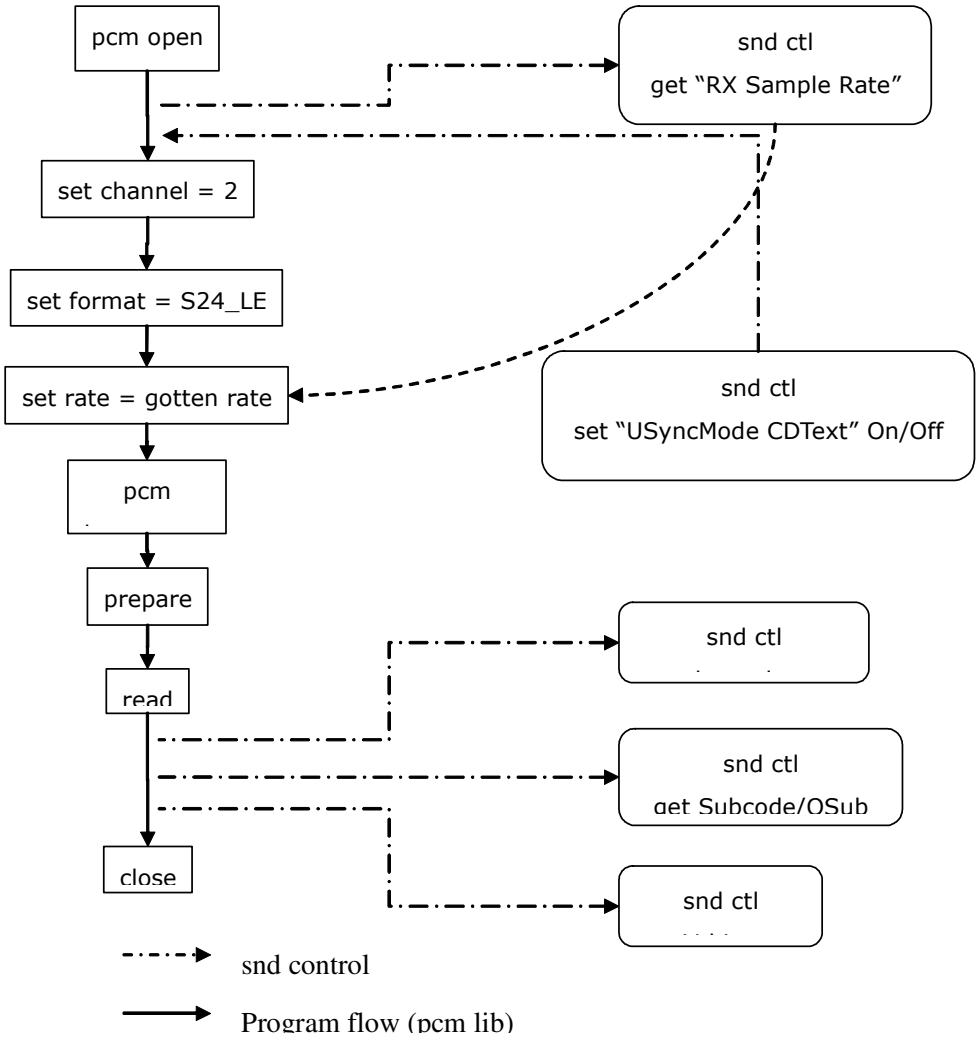


Figure 18-2. S/PDIF Rx Application Program Flow

18.4 Interrupts and Exceptions

S/PDIF Tx/Rx hardware block has many interrupts to indicate the success, exception and event. The driver handles the following interrupts:

- DPLL Lock and Loss Lock—Saves the DPLL lock status; this is used when getting the Rx sample rate
- U/Q Channel Full and overrun/underrun—Puts the U/Q channel register data into queue buffer, and update the queue buffer write pointer
- U/Q Channel Sync—Saves the ID of the buffer whose U/Q data is ready for read out
- U/Q Channel Error—Resets the U/Q queue buffer

18.5 Source Code Structure

Table 18-2 lists the source file that is available in the directory:

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

Table 18-2. S/PDIF Driver Files

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

18.6 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

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

  Device Drivers > Sound card support > Advanced Linux Sound Architecture

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

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

  By default, this option is M.

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

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

19.1 Hardware Operation

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

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

19.2 Software Operation

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

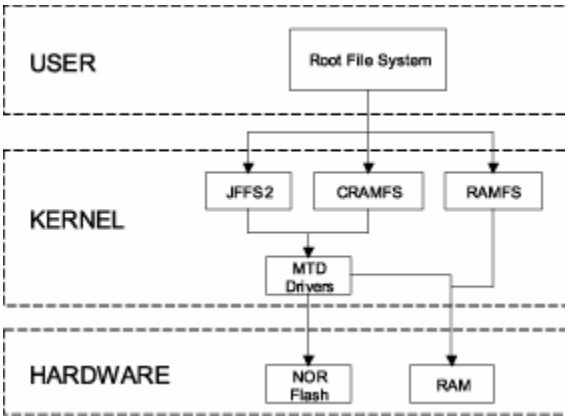


Figure 19-1. Components of a Flash-Based File System

The MTD subsystem for Linux is a generic interface to memory devices, such as Flash and RAM, providing simple read, write and erase access to physical memory devices. Devices called `mtdblock`

devices can be mounted by JFFS, JFFS2 and CRAMFS file systems. The SPI NOR MTD driver is based on the MTD data Flash driver in the kernel by adding SPI access. In the initialization phase, the SPI NOR MTD driver detects a data Flash by reading the JEDEC ID. Then the driver adds the MTD device. The SPI NOR MTD driver also provides the interfaces to read, write, erase NOR Flash.

19.3 Driver Features

This NOR MTD implementation supports the following features:

- Provides necessary information for the upper layer MTD driver

19.4 Source Code Structure

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

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

Table 19-1 shows the driver files:

Table 19-1. SPI NOR MTD Driver Files

File	Description
mxc_dataflash.c	Source file

19.5 Menu Configuration Options

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

- Device Drivers > Memory Technology Device (MTD) support

Chapter 20
NAND Flash Memory Technology Device (MTD) Driver

20.1 Overview

The NAND Flash MTD driver is for the NAND Flash Controller (NFC) on the i.MX series processor. For the NAND MTD driver to work, only the hardware specific layer has to be implemented. The rest of the functionality, such as Flash read/write/erase, is automatically handled by the generic layer provided by the Linux MTD subsystem for NAND devices.

20.1.1 Hardware Operation

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

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

20.1.2 Software Operation

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

There are three layers of NAND MTD drivers:

- MTD driver
- Generic NAND driver
- Hardware specific driver

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

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

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

20.2 Requirements

This NAND Flash MTD driver implementation meets the following requirements:

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

20.3 Source Code Structure

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

20.4 Linux Menu Configuration Options

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

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

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

20.5 Programming Interface

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

Chapter 21
Low-Level Keypad Driver

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

The keypad driver supports the following features:

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

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

21.1 Hardware Operation

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

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

21.2 Software Operation

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

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

Every keypress or release follows the debounce state machine shown in Figure 21-1. The `mxc_kpp_scan_matrix` function is called for every keypress and release interrupt.

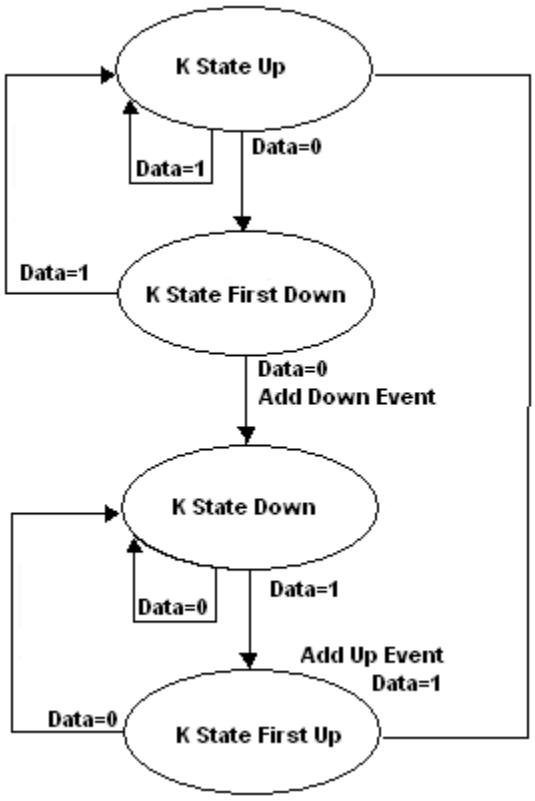


Figure 21-1. Keypad Driver State Machine

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

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

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

The structure of `input_event` is as follows:

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

where:

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

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

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

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

Refer to Table 21-3 for map codes and scan codes.

21.3 Reassigning Keycodes

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

21.4 Driver Features

The keypad driver supports the following features:

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

21.5 Source Code Structure

Table 21-1 shows the keypad driver source files that are available in the following directories:

```
<ltib_dir>/rpm/BUILD/linux/drivers/input/keyboard
<ltib_dir>/rpm/BUILD/linux/include/linux
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51
```

Table 21-1. Keypad Driver Files

File	Description
mxc_keyb.c	Low-level driver implementation
mxc_keyb.h	Driver structures, control register address definitions
nput.h	Generic Linux keycode definitions
mx51_3stack.c	Contains the platform-specific keymapping keycode array

21.6 Menu Configuration Options

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

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

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

  Device Drivers > Input device support > Event interface.

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

- Matrix config—The keypad matrix can be configured for up to eight rows and eight columns. The keypad matrix configuration can be done by changing the `rowmax` and `colmax` members in the `keypad_plat_data` structure in the platform specific file (see Table 21-1).
- Debounce delay—The user can configure the debounce delay by changing the variable `KScanRate` defined in `mxc_keyb.c`

21.7 Programming Interface

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

21.8 Interrupt Requirements

Table 21-2 lists the keypad interrupt timer requirements.

Table 21-2. Keypad Interrupt Timer Requirements

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

21.9 Device-Specific Information

Table 21-3 shows key connections, key scan codes, and key map codes of the keys on the keypad for a specific platform.

Table 21-3. Key Connections for Keypad

Key	Row	Column	Scancode	Linux Key Code	Platform
1	0	0	0	KEY_1	i.MX51
2	0	1	1	KEY_2	i.MX51
3	0	2	2	KEY_3	i.MX51

Table 21-3. Key Connections for Keypad (continued)

Key	Row	Column	Scancode	Linux Key Code	Platform
FNC1	0	3	3	KEY_F1	i.MX51
UP	0	4	4	KEY_UP	i.MX51
FNC2	0	5	5	KEY_F2	i.MX51
4	1	0	6	KEY_4	i.MX51
5	1	1	7	KEY_5	i.MX51
6	1	2	8	KEY_6	i.MX51
LEFT	1	3	9	KEY_LEFT	i.MX51
SELECT	1	4	10	KEY_SELECT	i.MX51
RIGHT	1	5	11	KEY_RIGHT	i.MX51
7	2	0	12	KEY_7	i.MX51
8	2	1	13	KEY_8	i.MX51
9	2	2	14	KEY_9	i.MX51
GP1	2	3	15	KEY_F3	i.MX51
DOWN	2	4	16	KEY_DOWN	i.MX51
GP2	2	5	17	KEY_F4	i.MX51
0	3	0	18	KEY_0	i.MX51
OK	3	1	19	KEY_OK	i.MX51
ESC	3	2	20	KEY_ESC	i.MX51
ON	3	3	21	KEY_ENTER	i.MX51
MENU	3	4	22	KEY_MENU	i.MX51
BACK	3	5	23	KEY_BACK	i.MX51

Chapter 22
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 802.3 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 (normally eth0; however, in the case of a FEC driver enabled it is eth1). The probe function of this driver is declared in `<ltib_dir>/rpm/BUILD/linux/drivers/net/Space.c` to probe for the device and to initialize it during boot.

22.1 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.

22.2 Software Operation

The SMSC LAN9217 Ethernet Driver has the functions:

- 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

22.3 Requirements

The Ethernet driver meets the following requirements:

- Provides all the entry points to interface with the Linux kernel 2.6 net module
- Implements the default data configuration function to set the MAC address and interface media used in case of EEPROM failure
- Follows Linux kernel coding style. This is included in Linux distributions as the file Documentation/Coding Style

22.4 Source Code Structure

Table 22-1 shows the source files available in the

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

Table 22-1. Ethernet Driver Files

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

22.5 Linux Menu Configuration Options

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

- CONFIG_SMSC911X – Provided for this module. This option is available under

  Device Drivers > Network Device Support > Ethernet (10 or 100 Mbit) > SMSC LAN911x/LAN921x families embedded ethernet support.

Chapter 23
Fast Ethernet Controller (FEC) Driver

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

The FEC driver supports the following features:

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

The network adapter can be accessed through the `ifconfig` command with interface name `eth0`. The driver auto-probes the external adaptor (PHY device).

23.1 Hardware Operation

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

A brief overview of the device functionality is provided here. For details see the FEC chapter of the *i.MX51 Multimedia Applications Processor Reference Manual*.

In MII mode, there are 18 signals defined by the IEEE 802.3 standard and supported by the EMAC. SNI mode uses a subset of the 18 signals. These signals are listed in Table 23-1.

Table 23-1. Pin Usage in MII and SNI Modes

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

Table 23-1. Pin Usage in MII and SNI Modes (continued)

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

The MII management interface consists of two pins, FEC_MDIO and FEC_MDC. These pins are configured through the GPIO settings. The FEC hardware operation can be divided in the following parts. For detailed information consult the *i.MX51 Multimedia Applications Processor Reference Manual*.

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

  Before transmitting, the controller waits for carrier sense to become inactive, then determines if carrier sense stays inactive for 60 bit times. If the transmission begins after waiting an additional 36 bit times (96 bit times after carrier sense originally became inactive). Both buffer (TXB) and frame (TXF) interrupts may be generated as determined by the settings in the EIMR.
- Reception—The FEC receiver is designed to work with almost no intervention from the host and can perform address recognition, CRC checking, short frame checking, and maximum frame length checking. When the driver enables the FEC receiver by asserting ECR[ETHER_EN], it immediately starts processing receive frames. When FEC_RX_DV asserts, the receiver checks for a valid PA/SFD header. If the PA/SFD is valid, it is stripped and the frame is processed by the receiver. If a valid PA/SFD is not found, the frame is ignored. In MII mode, the receiver checks for at least one byte matching the SFD. Zero or more PA bytes may occur, but if a 00 bit sequence is detected prior to the SFD byte, the frame is ignored.

  After the first six bytes of the frame have been received, the FEC performs address recognition on the frame. During reception, the Ethernet controller checks for various error conditions and once the entire frame is written into the FIFO, a 32-bit frame status word is written into the FIFO. This

status word contains the M, BC, MC, LG, NO, CR, OV, and TR status bits, and the frame length. Receive Buffer (RXB) and Frame Interrupts (RXF) may be generated if enabled by the EIMR register. When the receive frame is complete, the FEC sets the L bit in the RxBD, writes the other frame status bits into the RxBD, and clears the E bit. The Ethernet controller next generates a maskable interrupt (RXF bit in EIR, maskable by RXF bit in EIMR), indicating that a frame has been received and is in memory. The Ethernet controller then waits for a new frame.

- Interrupt management—When an event occurs that sets a bit in the EIR, an interrupt is generated if the corresponding bit in the interrupt mask register (EIMR) is also set. The bit in the EIR is cleared if a one is written to that bit position; writing zero has no effect. This register is cleared upon hardware reset. These interrupts can be divided into operational interrupts, transceiver/network error interrupts, and internal error interrupts. Interrupts which may occur in normal operation are GRA, TXF, TXB, RXF, RXB, 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 are visible to network management through the MIB counters. For PHY interrupt, which is interfaced through PBC (CPLD), it is optional for link status detect.

23.2 Software Operation

The FEC driver supports the following functions:

- 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

23.3 Source Code Structure

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

Table 23-2. FEC Driver Files

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

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

23.4 Menu Configuration Options

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

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

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

23.5 Programming Interface

Table 23-2 lists the source files for the FEC driver. The following section shows the modifications that were required to the original Ethernet driver source for porting it to the i.MX device.

23.5.1 Device-Specific Defines

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

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

```
/*
 *      Define the buffer descriptor structure.
 */
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 */
        unsigned long     reserved8[9];             /* Transmit descriptor ring */
        unsigned long     fec_fifo_ram[112];        /* FIFO RAM buffer */
} fec_t;
```

23.5.2 Getting a MAC Address

The following statement gets the MAC address through the IIM (IC Identification).

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

If the MAC address is not programmed, the driver sets the MAC address to “0x00:0x00:0x00:0x00:0x00:0x00:”, which is not an acceptable address. The MAC address can also be set by the REDBOOT command `fconfig`.

Due to certain pin conflicts, the LCD driver must be disabled when using the FEC driver on the i.MX51 3-Stack board.

Chapter 24
WLAN Driver

The APM6628 is a full-featured Wi-Fi 802.11b/g and Bluetooth v2.0+EDR combo module that simultaneously provides Wi-Fi and Bluetooth connections.

The WLAN driver is used to drive the APM6628 module to implement Wi-Fi functionality. The APM6628 module adopts the CSR UniFi V5 solution.

The UniFi driver implements the Wi-Fi module of the APM6628. This driver provides access to a network using an access point (AP), which is a standard Ethernet interface in Linux. The user level tools communicate with the UniFi driver using the Linux Wireless Extension. The user can use the Linux Wireless Tools (WT) to configure Wi-Fi.

24.1 Hardware Operation

The APM6628 provides the Wi-Fi functionality needed to interface the system to a LAN network.

24.1.1 Register Access

The APM6628 accesses its registers using two methods: SDIO and SPI. The i.MX boards use SDIO to access the APM6628 with the CMD52 command channel and the CMD53 data channel as follows:

- For APM6628 register access, the driver uses the CMD52 access to each of the on-chip registers and memory locations directly.
- For APM6628 data access, the driver uses the CMD53 to transfer blocks of data directly to or from the on-chip MMU buffers.

24.1.2 Transmission

The driver uses the CMD52/53 to transfer packets to the device.

24.1.3 Reception

The driver uses the CMD52/53 to receive packets from the device.

24.1.4 Encryption and Decryption

There are two phases of encryption or decryption. In the first phase, the driver uses CMD52/53 to transmit

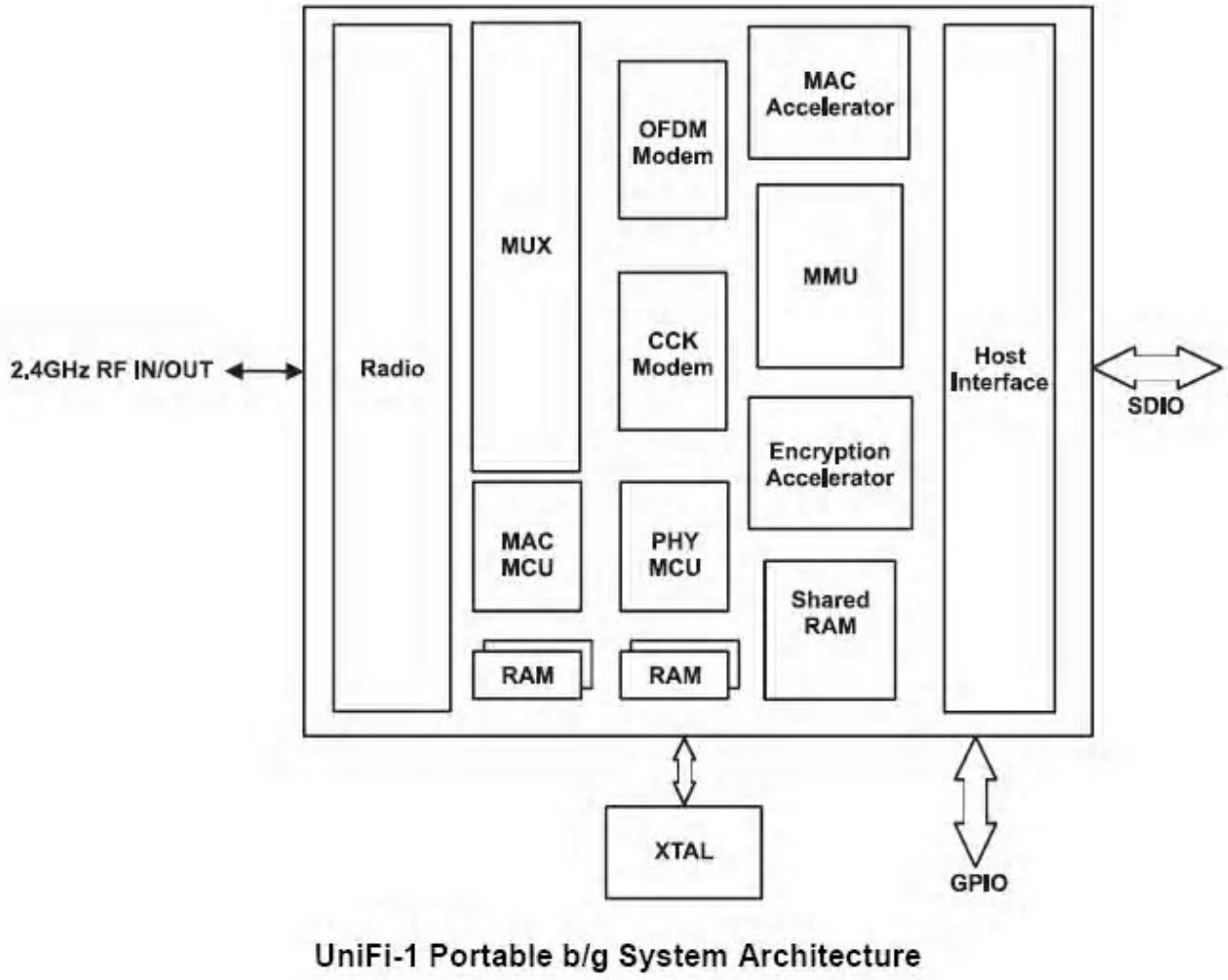
Figure 24-1.

original data to the device. In the second phase, the driver uses CMD52/53 to read back the data which has been transmitted. Full support for WEP40/64 and WEP104/128; WPA/WPA2(802.11i) and enhanced encryption modes.

Full support for WEP40/64 and WEP104/128; WPA/WPA2(802.11i) enhanced encryption modes.

Figure 24-2 illustrates system architecture.

Figure 24-2. Wi-Fi System Architecture

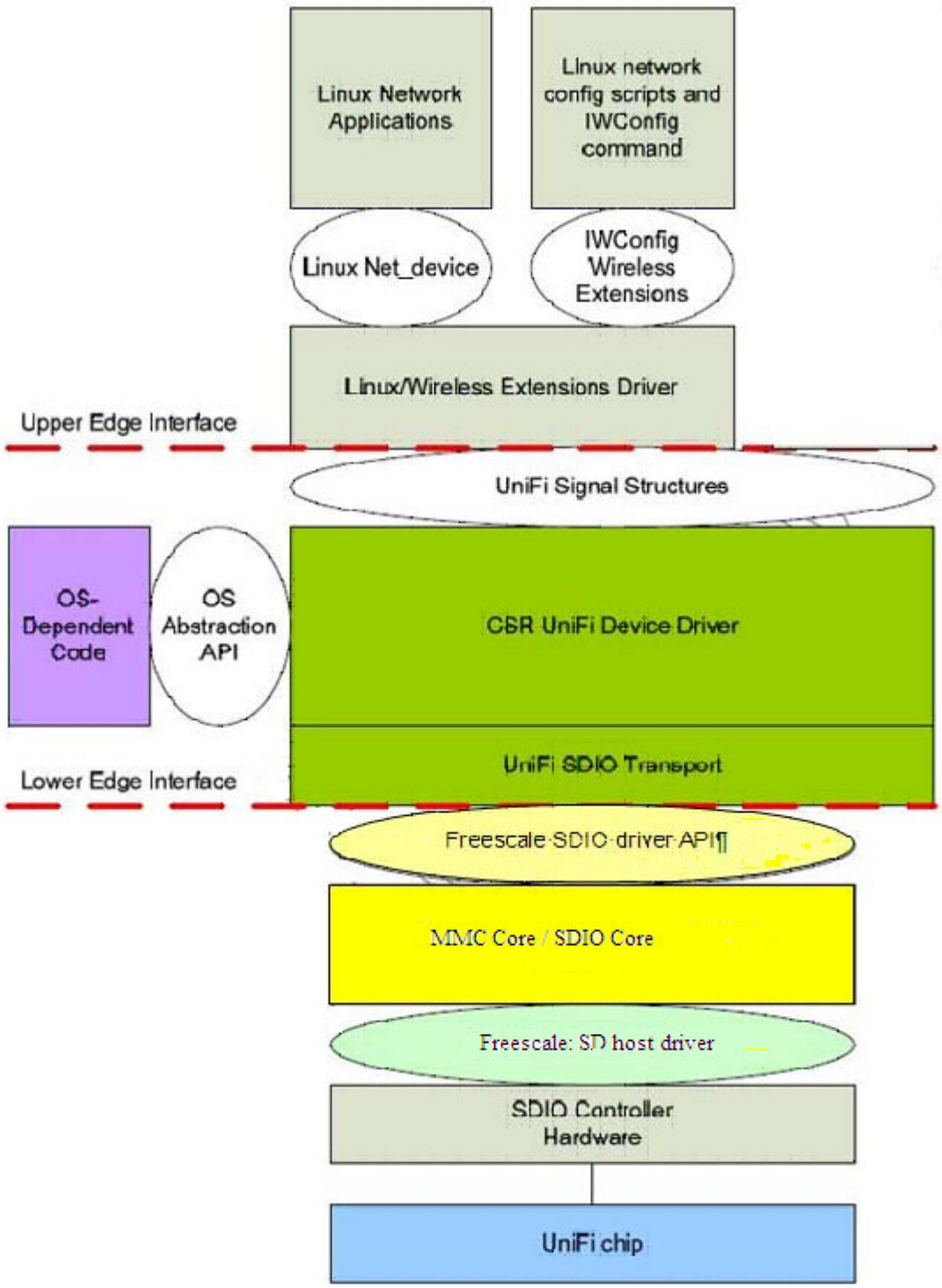


24.2 Software Operation

Figure 24-3. Software architecture

The CSR UniFi driver implements the Wi-Fi module of the APM6628. This driver provides access to a network using an access point (AP), which simulates a standard Ethernet interface in Linux. The software

operates the Wi-Fi device using Linux Wireless Extension, and configures Wi-Fi using Linux Wireless Tools (WT) or wpa_supplicant. Figure 24-3 illustrates the software architecture.



24.3 Configuration

There are two configurations: kernel configuration and WPA configuration.

24.3.1 Linux Configuration

To get to the WLAN 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 configuration options are provided for the driver:

- Networking > Wireless > Generic IEEE 802.11 Networking Stack
- Networking > Wireless > Wireless extensions

Select these options for wireless support. By default, these options are enabled.

24.3.2 WPA Configuration

The following configurations are provided for the WPA supplicant:

```
CONFIG_WIRELESS_EXTENSION=y
CONFIG_IEEE8021X_EAPOL=y
CONFIG_EAP_PSK=y
CONFIG_CTRL_IFACE=y
CONFIG_L2_PACKET=linux
```

"These configurations are located in the file in <ltib dir>/dist/lfs-5.1/wpa_supplicant/wpa_supplicant.spec. Ltib builds the WPA supplicant automatically according to the spec file.

24.4 Programming Interface

The Freescale SDIO structure exports an interface to get the `mmc_host` structure pointer to the UniFi driver. The UniFi driver uses this structure to issue the process of discovery of the SDIO card, and adds code to enable or disable the power supply.

Chapter 25
Security Drivers

The security drivers provide several APIs that facilitate access to various security features in the processor. The secure controller (SCC) consists of two modules, a secure RAM module and a secure monitor module. The SCC key encryption module (KEM) has a security feature for storing encrypted data in the on-chip RAM (Red data = unencrypted data, Black data = encrypted data), with a total size of 2 Kbytes. This module is needed in cases where data must be stored securely in external memory in encrypted form. This module can clear the secure RAM during intrusion.

The security design covers the following modules:

- Boot Security
- SCC (Secure RAM, Secure Monitor)
- Algorithm Integrity Checker
- Security Timer
- Key Encryption Module (KEM), Zeroization module

25.1 Hardware Overview

The platform has several different security blocks. The details of the individual blocks are described in the following sections.

25.1.1 Boot Security

During boot, the boot pins must be set to enable the processor to boot. The SCC module must be enabled by blowing specific fuses. By booting in this manner, the integrity of the data in the Flash (kernel image) can be assured. Any violation in the data integrity raises an alarm.

25.1.2 Secure RAM

Figure 25-1 shows the SCC-Secure RAM and its modules. Individual modules are described in the following sections.

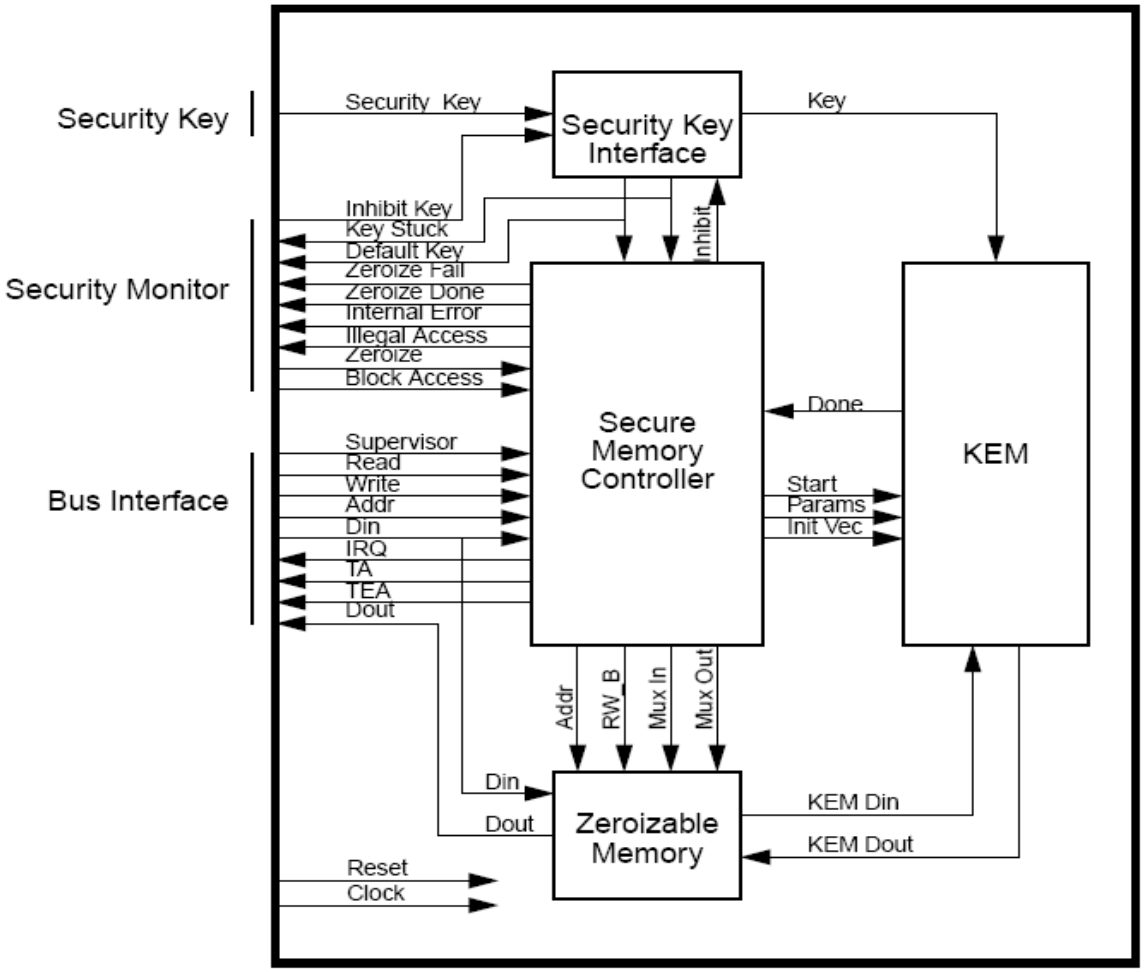


Figure 25-1. Secure RAM Block Diagram

25.1.3 KEM

The KEM uses the 3DES algorithm and a 168-bit key for encryption of data. The key is programmed during manufacture and is accessible only to the encryption module. It is not accessible on any bus external to the secure memory module. The data in the external RAM is stored in an encrypted format. The data is encrypted using 3DES algorithm so that it can be decoded only using the SCC module.

25.1.4 Zeroizable Memory

The memory module can be multiplexed in and out of the RAM to allow the memory controller to switch paths according to the Secure RAM state and the host read and write accesses. When zeroing sections of memory, only the memory controller has access. When encrypting or decrypting, only the KEM module

has access. When the Secure RAM is in the Idle state, the host can access the memory. The Zeroize Done signal is used to reset the encryption module and the memory controller. While the Zeroize Done signal is low, any attempted access by the host is ignored. When the Zeroize signal is asserted, or when the Zeroize Memory bit in the Interrupt Control register is set, not only is the Red and Black memory initialized, but most of the registers are also reset. The Red Start, Black Start, Length, Control, Error Status, Init Vector 0, and Init Vector 1 registers are cleared. The encryption engine is also reset. The Zeroization takes place whenever there is a security violation like external bus intrusion. The Red and Black memory area is usually cleared during system boot-up.

25.1.5 Security Key Interface Module

The Security Key Interface module uses a 168-bit encryption key. The physical structures for the encryption key resides elsewhere. The Secret Key Interface contains a key mux to select between the encryption key and the default key and test the logic to determine the validity of the encryption key. In the Secure state the encryption key is used. In the Non-Secure state, the default key prevents unauthorized access to SCC-encrypted data and is useful for test purposes.

25.1.6 Secure Memory Controller

The Secure Memory controller implements an internal data handler that moves data in and out of the KEM, a memory clear function, and all of the supervisor-accessible Control and Status registers.

25.1.7 Security Monitor

The Security Monitor (SMN) is a critical component of security assurance for the platform. It determines when and how Secure RAM resources are available to the system, and it also provides mechanisms for verifying software algorithm integrity. This block ensures that the system is running in such a manner as to provide protection for the sensitive data that is resident in the SCC. The Security Monitor consists of five main sub-blocks:

- Secure State Controller
- Security Policy
- Algorithm Integrity Checker (AIC)
- Security Timer
- Debug Detector

Figure 25-2 shows a block diagram of the SMN.

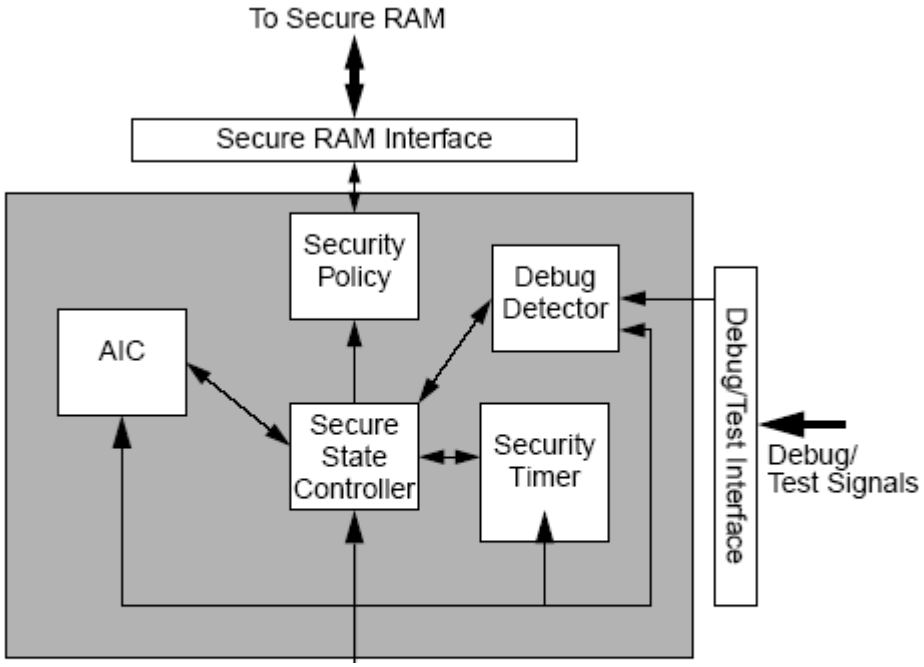


Figure 25-2. Security Monitor Block Diagram

25.1.8 Secure State Controller

The Secure State Controller, shown in Figure 25-3, is a state machine that controls the security states of the chip.

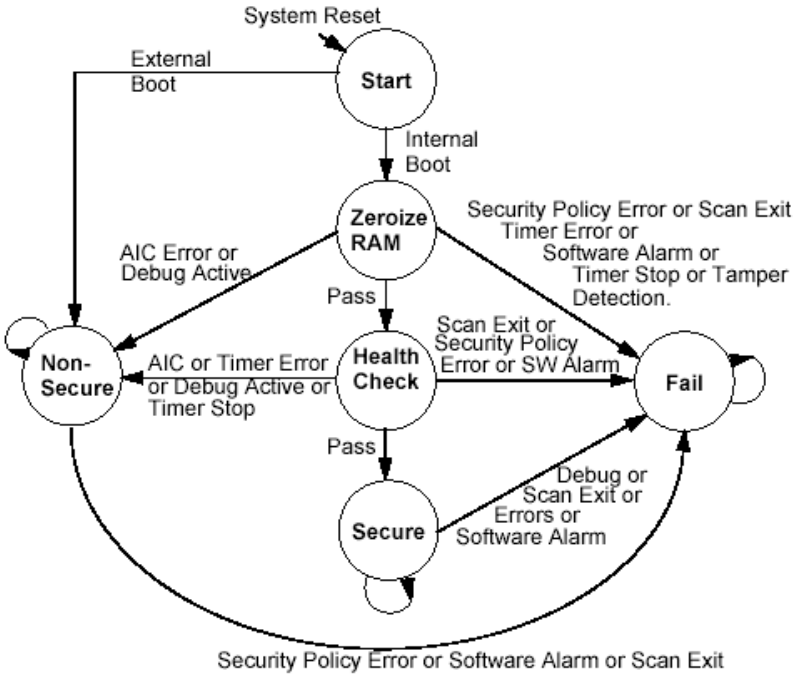


Figure 25-3. Secure State Controller State Diagram

25.1.9 Security Policy

The Security Policy block uses state information from the Secure State Controller along with inputs from the Secure RAM to determine what access to the Secure RAM is allowed based on the policy table. The policy table is available in the L3 specification document of the corresponding platform.

25.1.10 Algorithm Integrity Checker (AIC)

The Algorithm Integrity Checker (AIC) is used in conjunction with software to provide assurance that critical software (such as a software encryption algorithm) operates correctly. It is also an integral part of the power-up procedure as it must be used to achieve a secure state.

25.1.11 Secure Timer

The Secure Timer is a 32-bit programmable timer. It is used in conjunction with the Secure State Controller during power-up to ensure that the transition to the Secure state happens in the appropriate amount of time. After power-up, the timer can be used as a watchdog timer for any time-critical routines or algorithms. If the timer is allowed to expire, it generates an error.

25.1.12 Debug Detector

The debug detector monitors the various debug and test signals and informs the secure state controller of the status. The secure state controller receives an alert when debug modes, such as JTAG and scan are active. The debug detector status register can be read by the host processor to determine which debug signals are currently active. Refer to the SCC section in L3 specification document of the corresponding platform for more information on the SCC-Debug Detector.

25.2 Software Operation

Besides the hardware security modules, there is optional, specialized software that helps to deliver security.

25.2.1 SCC Common Software Operations

The SCC driver is only available to other kernel modules. That is, there is no node file in `/dev`. Thus, it is not possible for a user-mode program to access the driver, and it is not possible for a user program to access the device directly.

With the exception of `scc_monitor_security_failure()`, all routines are synchronous, which means they do not return to their caller until the requested action completes, or fails to complete. Some of these functions could take some time to perform, depending upon the request.

Routines are provided to:

- Encrypt or decrypt secrets—`scc_crypt()`
- Trigger a security-violation alarm—`scc_set_sw_alarm()`
- Get configuration and version information—`scc_get_configuration()`

- Zero areas of memory—`scc_zeroize_memories()`
- Work on wrapped and stored secret values—`scc_alloc_slot()`, `scc_dealloc_slot()`, `scc_load_slot()`, `scc_decrypt_slot()`, `scc_encrypt_slot()`, and `scc_get_slot_info()`
- Monitor the Security Failure alarm—`scc_monitor_security_failure()`
- Stop monitoring Security Failure alarm—`scc_stop_monitoring_security_failure()`
- Write registers of the SCC—`scc_write_register()`
- Read registers of the SCC—`scc_read_register()`

The driver does not allow storage of data in either the Red or Black memories. Any decrypted information is returned to the user. If the user wants to use the information at a later point, the encrypted form must again be passed to the driver, and it must be decrypted again.

The SCC encrypts and decrypts using Triple DES with an internally stored key. When the SCC is in Secure mode, it uses its secret, unique-per-chip key. When it is in Non-Secure mode, it uses a default key. This ensures that secrets stay secret if the SCC is not in Secure mode.

Not all functions are implemented, such as interfaces to the ASC/AIC components and the timer functions. These and other features must be accessed through `scc_read_register()` and `scc_write_register()`, using the `#define` values provided.

25.3 Driver Features

The SCC driver supports the following features:

- Checks whether the SCC fuse is blown or not (SCC Disabled/Enabled)
- Configures the Red and Black memory area addresses and number of blocks to be encrypted/decrypted
- Loads the data to be encrypted
- Loads the data to be decrypted
- Starts the Ciphering mechanism
- Reports back the status of the KEM module
- Zeros blocks in the Red/Black memory area
- Checks for the boot type: internal or external
- Raises a software alarm
- Reports back the status of the Zeroize module
- Configures the AIC start and end algorithm sequence number
- Checks the sequence of the algorithm
- Finds the next sequence number given the current sequence number
- Configures the Security Timer
- Reports back the status of the Security Timer module

25.4 Source Code Structure

This section contains the various files that implement the Security modules. Table 25-1 lists the headers and source files associated with the security driver.

- The C source files are available in the directory, `<ltib_dir>/rpm/BUILD/linux/drivers/mxc/security` directory.
- Header files are available in the directory, `<ltib_dir>/rpm/BUILD/linux/include/linux`.
- The RNG driver also depends on the header files in the directory, `<ltib_dir>/rpm/BUILD/linux/drivers/mxc/security/sahara2/include`.

Table 25-1. SCC Driver Files

File	Description
Makefile	Used to compile, link and generate the final binary image
mxc_scc_driver.h	Header file related to SCC module interface
mxc_scc_internals.h	Header file which contains definitions needed by the SCC driver. This is intended to be the file that contains most or all of the code or changes needed to port the driver.
scc2_driver.h	Header file related to SCCV2 module interface
scc2_internals.h	Header file with SCCV2 driver-related definitions

25.5 Menu Configuration Options

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

- CONFIG_MXC_SECURITY_SCC—Use the SCC module. In menuconfig, it is available under Device Drivers > MXC Support drivers > MXC Security Drivers > MXC_SCC_Driver.
- CONFIG_MXC_SECURITY_SCC2—Use the SCCV2 module. In menuconfig, it is available under

  Device Drivers > MXC Support drivers > MXC Security Drivers > MXC SCC2 Driver.

  By default, this option is Y
- CONFIG_MXC_SECURITY_RNG—Use the RNG module core API. In menuconfig, it is available under

  Device Drivers > MXC Support drivers > MXC Security Drivers > MXC_RNG_Driver.

  By default, this option is Y
- CONFIG_RNG_TEST_DRIVER—Debug the RNG module. In menuconfig, it is available under Device Drivers > MXC Support drivers > MXC Security Drivers > MXC RNG Driver > MXC RNG debug register

  By default, this option is N for platform. This configuration should be enabled for `rnga_read_register()` and `rnga_write_register()` functions to be defined and exported. This may affect inserting the test driver modules, which might assume the availability of these functions.

- CONFIG_MXC_SECURITY_CORE—Use the security core module API such as RNG and RTIC. In menuconfig, it is available under

  MXC Support drivers > MXC Security Drivers

  By default, this option is Y

25.5.1 Source Code Configuration Options

25.5.1.1 Board Configuration Option

To Configure the SCC, perform the following steps:

1. Install Icepick and point it to the license file
- Blow the following fuses to SCC key 0–SCC key 20. Refer to the *MCIMX51 Multimedia Applications Processor Reference Manual (MCIMX51RM)* for register details.

```
SCC Key0   = 0x77
SCC Key1   = 0xff
SCC Key2   = 0x3a
SCC Key3   = 0x76
SCC Key4   = 0x02
SCC Key5   = 0xb0
SCC Key6   = 0x0a
SCC Key7   = 0x0d
SCC Key8   = 0x90
SCC Key9   = 0x76
SCC Key10  = 0xf8
SCC Key11  = 0x07
SCC Key12  = 0x13
SCC Key13  = 0x9e
SCC Key14  = 0x36
SCC Key15  = 0xd3
SCC Key16  = 0xfa
SCC Key17  = 0x00
SCC key18  = 0x00
SCC Key19  = 0x9d
SCC Key20  = 0xfe
```

Follow the instructions below to program the SCC key using Icepick:

1. Run Icepick
2. Issue the following commands

```
openSocket <IP Address of ICE>
initZas
source util_fuse_<platform>.tcl
init_iim
blow_fuse bank  row bit
```

The final command writes the desired fuse. The parameters passed to blow_fuse are bank, row and bit. For information about parameters to be passed refer to the L3 specification for the appropriate platform.

The following example shows how to program the value 0x77 into SCC Key0:

```
blow_fuse 1 1 0
blow_fuse 1 1 1
```

```
blow_fuse 1 1 2
blow_fuse 1 1 4
blow_fuse 1 1 5
blow_fuse 1 1 6
```

3. Issue this command:

```
sense_fuse bank row bit
```

This command reads the desired fuse value.

4. Write the following ASC Sequence in the debugger script (init_sdram.txt)

```
setmem /32 0x53FAD008 =0x00005CAA
setmem /32 0x53FAD00C =0x00002E55
setmem /32 0x53FAD010 =0x00002E55
```

5. Configure the boot mode pins SW7-1 and SW7-2 to Internal Boot.

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

26.1 Overview

The Symmetric/Asymmetric Hashing and Random Accelerator (Sahara) implements block encryption algorithms, hashing algorithms, a stream cipher algorithm, public key algorithms, and pseudo-random number generation. It has a slave IP bus interface for the host to write configuration and command information, and to read status information. It also has a DMA controller, with an AHB bus interface, to reduce the burden on the host to move the required data to and from memory.

26.2 Software Operation

26.2.1 API Notes

Kernel users should not use blocking mode unless the code is operating on behalf of the kernel process which needs to sleep because blocking mode attempts to put the current process to sleep. Therefore blocking mode cannot be used from bottom half code or 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) using the stack for I/O, 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 applies to key structures, context structures, and input/output buffers.

26.2.2 Architecture

The conceptual model is shown in Figure 26-1. All of the processes in Figure 26-1 are implemented as common code, except for the following platform-centric processes:

- UM Extension
- Init/Cleanup
- Translator
- 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.

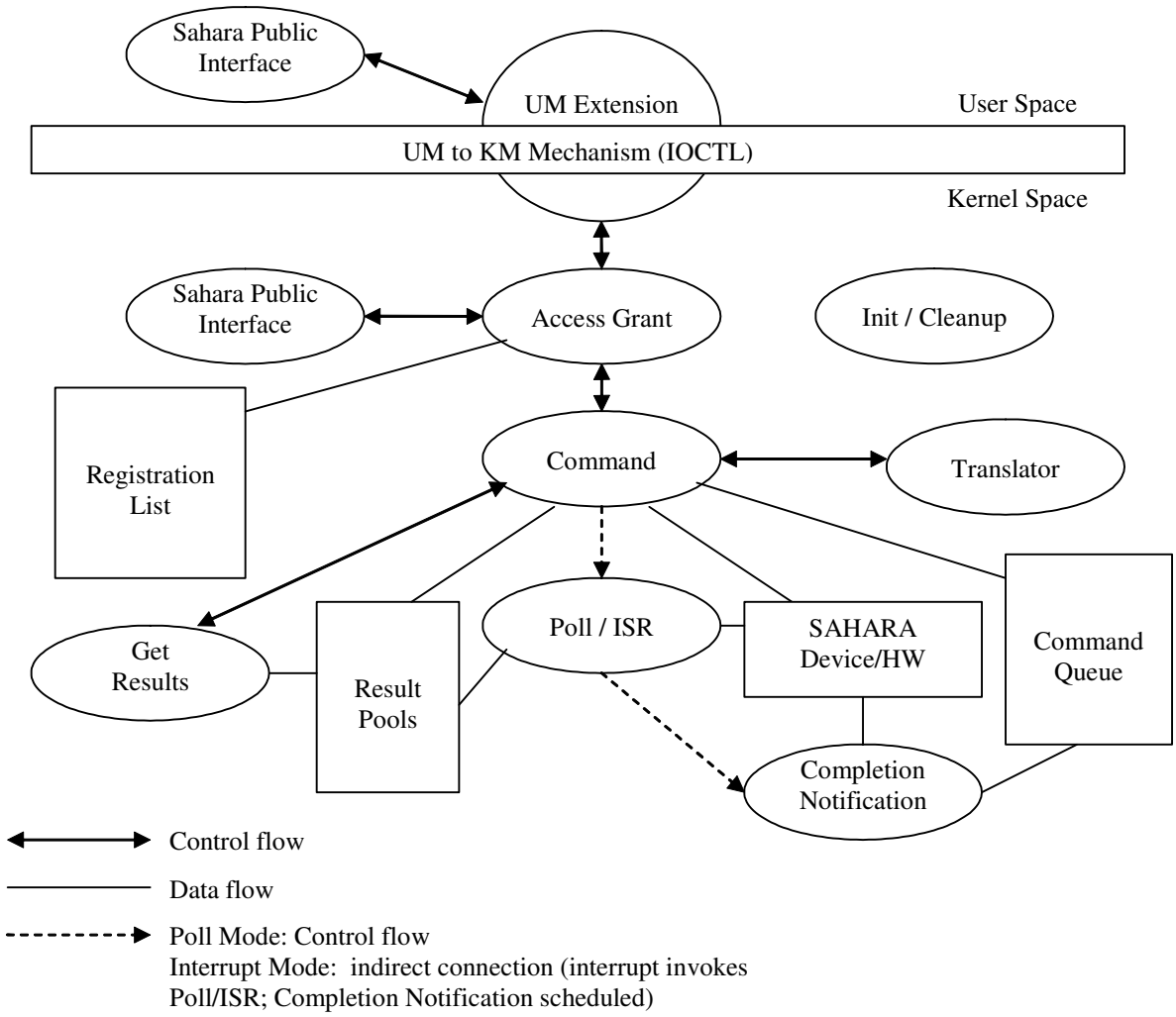


Figure 26-1. Sahara Architecture Overview

26.2.2.1 Registration List

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

26.2.2.2 Command Queue

The command queue maintains a list of commands (pointers to descriptor chains and their associated user) destined for the Sahara hardware. 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

The results pool maintains a list of completed commands. After the Sahara hardware completes, the status, along with its user association is placed in this pool.

26.2.2.4 Sahara Hardware

i.MX51 uses Sahara version4 hardware.

26.2.2.5 Initialize and Cleanup

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

- Initialize the driver when the OS wishes to start the driver
- Cleanup the driver when the OS wishes to shut down the driver

The following stops are used to initialize the driver:

1. Map Sahara registers into kernel space
2. Check that the Sahara version number is 4
3. Attach handler to Sahara interrupt line (top half) if in interrupt mode
4. Initialize Sahara interrupt
5. Seed the random number generator. The RNG Auto Reseed in the control register 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 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 (most of this can be done at compile time) as follows:
   — Sahara version (4)
   — Command queue size
   — Result pool size
   — Registration list size
   — Driver version number
   — Algorithms and modes
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

The following steps are used 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 (leave Sahara 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 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 return value
- Passes raw descriptor chains through (must be registered, that is, have a UCO). It is 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 information passed from 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 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 HW or not

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

1. Checks that there is room in the Command Queue. If there is not, returns a `queue full` status and terminates
2. Checks that there is room in the Result Pool. If there is 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, return `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 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)

Table 26-1. Blocking/Non-Blocking Definitions (continued)

User Non-blocking with callback	Driver never gives up processor, therefore User is blocked until request completion (the callback is invoked prior to request completion)	Driver queues request and returns. Upon request completion, the callback is invoked
User Non-blocking without 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 poll mode, transfer control to Poll/ISR process (if in interrupt mode, the Poll/ISR process is invoked through the interrupt mechanism)

Requests that do not involve Sahara hardware, are processed and immediately returned 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
- Ensures that blocks of data that have become fragmented due to page discontinuity are handled with links in the descriptor chain
- Locks pages so addresses remain stable
- Clears processor cache

26.2.2.11 Polling and Interrupts

Polling has the following features:

1. Continuously checks if operation is done (poll the State field in the Status Register) to determine when the Sahara hardware has completed
2. Moves the content of the in-progress element from the Command Queue to the Result Pool
3. Copies Sahara Status and Error Status registers into the result pool
4. Writes `Clr_Error` in Command register and flag as FAILED if State field in Status register is 010 or 110 (otherwise set to PASSED)
5. Loads the next command into Sahara, if one exists (to 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 runs as a tasklet or is invoked as a function as follows:

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

26.3 Driver Features

The SAHARA driver supports the following features:

- Hashing with MD5, SHA-1, SHA-224 and SHA-256 algorithms
- HMAC with the same algorithms as for hashing
- 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 for AES
- Wrapped keys (hiding keys in the SCC), using the SCC key to encrypt or decrypt, HMAC functions, or CCM
- Generation of an arbitrary number of bytes of random data
- User mode and kernel mode; callbacks and non-callback non-blocking

26.4 Source Code Structure

Table 26-2 lists the source files associated with the SAHARA driver that are available in the directory `<ltib_dir>/rpm/BUILD/linux/drivers/mxc/security/sahara2`.

Table 26-2. Sahara Source Files

File	Description
sah_driver_interface.c	Sahara low level driver
sah_hardware_interface.c	Provides an interface to the SAHARA hardware registers
sah_interrupt_handler.c	Sahara 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. 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 26-3 lists the header files associated with the SAHARA driver are found in the directory `<ltib_dir>/rpm/BUILD/linux/drivers/mxc/security/sahara2/include`.

Table 26-3. Sahara Header Files

File	Description
fsl_shw.h	Sahara 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

Table 26-3. Sahara Header Files (continued)

File	Description
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 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_MXC_SAHARA —Configuration option for Sahara Hardware support. In the menuconfig this option is found under

  MXC support drivers > MXC Security Drivers > SAHARA2 Security Hardware Support
- 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 the menuconfig this option is found under

  MXC support drivers > MXC Security Drivers > SAHARA2 Security Hardware Support
- 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

  MXC support drivers > MXC Security Drivers > SAHARA2 Security Hardware Support
- 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 causing the Sahara to be reset. This time-out period is configurable. When poll mode is selected, the value for CONFIG_MXC_SAHARA_POLL_MODE_TIMEOUT can be modified. Poll 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).

26.6 Programming Interface

This driver implements all the methods that are required by the Linux serial API to interface with the Sahara driver. It implements and provides a set of control methods to the core Sahara 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

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

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

27.1 I^2C Bus Driver Overview

The I^2C bus driver is invoked only by the I^2C chip driver and is not exposed to the user space. The standard Linux kernel contains a core I^2C module that is used by the chip driver to access the I^2C bus driver to transfer data over the I^2C bus. The chip driver uses a standard kernel space API that is provided in the Linux kernel to access the core I^2C module. The standard I^2C kernel functions are documented in the files available under `Documentation/i2c` in the kernel source tree. This bus driver supports the following features:

- Compatible with the I^2C bus standard
- Bit rates up to 400 Kbps
- Starts and stops signal generation/detection
- Acknowledge bit generation/detection
- Interrupt-driven, byte-by-byte data transfer
- Standard I^2C master mode

27.2 I^2C Device Driver Overview

The I^2C device driver implements all the Linux I^2C data structures that are required to communicate with the I^2C bus driver. It exposes a custom kernel space API to the other device drivers to transfer data to the device that is connected to the I^2C bus. Internally these API functions use the standard I^2C kernel space API to call the I^2C core module. The I^2C core module looks up the I^2C bus driver and calls the appropriate function in the I^2C bus driver to do the data transfer. This driver provides the following functions to other device drivers:

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

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

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

27.4 Software Operation

The 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`. This field is a pointer to the `i2c_algorithm` structure that describes how data is transferred over the I^2C bus. The algorithm structure contains a pointer to a function that is called whenever the I^2C chip driver wants to communicate with an I^2C device.

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

After transmitting each packet, the I^2C bus driver waits for an interrupt indicating the end of a data transmission before transmitting the next byte. It times out and returns an error if the transfer complete signal is not received. Because the I^2C bus driver uses wait queues for its operation, other device drivers should be careful not to call the I^2C API methods from an interrupt mode.

27.4.2 I^2C Device Driver Software Operation

The I^2C driver controls an individual I^2C device on the I^2C bus. A structure, `i2c_driver`, describes the I^2C chip driver. The fields of interest in this structure are `flags` and `attach_adapter`. The `flags` field is set to a value `I2C_DF_NOTIFY` so that the chip driver can be notified of any new I^2C devices, after the driver is loaded. The `attach_adapter` callback function is called whenever a new I^2C bus driver is loaded in the system. When the I^2C bus driver is loaded, this driver stores the `i2c_adapter` structure associated with this bus driver so that it can use the appropriate methods to transfer data.

27.5 Driver Features

The I^2C driver supports the following features:

- I^2C communication protocol
- I^2C master mode of operation
- Does not support the I^2C slave mode of operation

27.6 Source Code Structure

Table 27-1 shows the I^2C bus driver source files available in the directory:

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

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

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

27.7 Menu Configuration Options

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

- Device Drivers > I2C support > I2C Hardware Bus support > MXC I2C support.

27.8 Programming Interface

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

27.9 Interrupt Requirements

The I^2C module generates many kinds of interrupts. The highest interrupt rate is associated with the transfer complete interrupt as shown in Table 27-2.

Table 27-2. I^2C Interrupt Requirements

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

The typical value of the transfer bit-rate is 200 Kbps. The best case values are based on a baud rate of 400 Kbps (the maximum supported by the I^2C interface).

Chapter 28
Configurable Serial Peripheral Interface (CSPI) Driver

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

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

28.1 Hardware Operation

CSPI is used for fast data communication with fewer software interrupts than conventional serial communications. Each CSPI is equipped with a data FIFO and is a master/slave configurable serial peripheral interface module, allowing the processor to interface with external SPI master or slave devices.

The primary features of the CSPI includes:

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

28.2 Software Operation

The following sections describe the CSPI software operation.

28.2.1 SPI Sub-System in Linux

The CSPI driver layer is located between the client layer (PMIC and SPI Flash are examples of clients) and the hardware access layer. Figure 28-1 shows the block diagram for SPI subsystem in Linux.

The SPI requests go into I/O queues. Requests for a given SPI device are executed in FIFO order, and complete asynchronously through completion callbacks. There are also some simple synchronous

wrappers for those calls, including ones for common transaction types like writing a command and then reading its response.

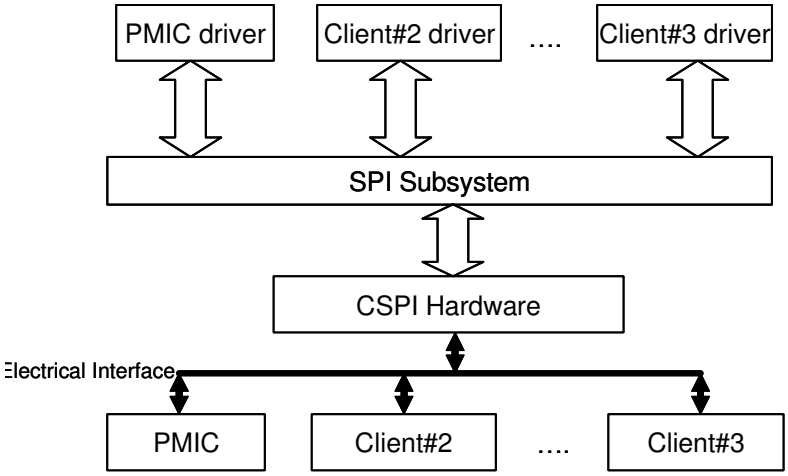


Figure 28-1. SPI Subsystem

All SPI clients must have a protocol driver associated with them and they must all be sharing the same controller driver. Only the controller driver can interact with the underlying SPI hardware module. Figure 28-2 shows how the different SPI drivers are layered in the SPI subsystem.

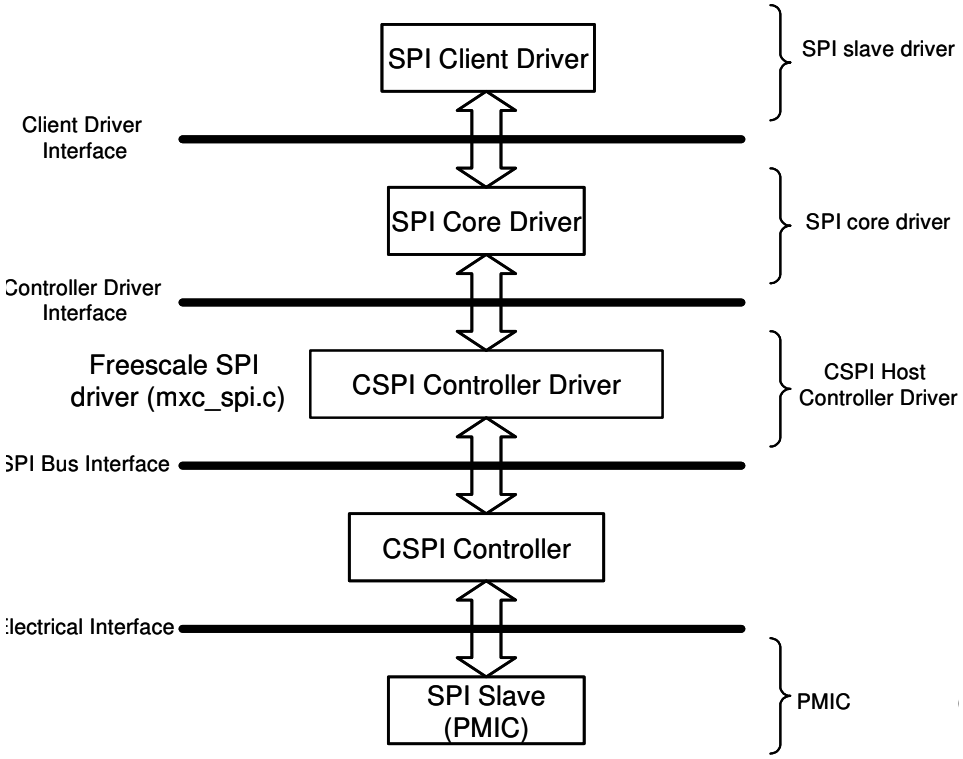


Figure 28-2. Layering of SPI Drivers in SPI Subsystem

28.2.2 Software Limitations

The CSPI driver limitations are as follows:

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

28.2.3 Standard Operations

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

- Init function `mxc_spi_init()`—Registers the `device_driver` structure.
- Probe function `mxc_spi_probe()`—Performs initialization and registration of the SPI device specific structure with SPI core driver. The driver probes for memory and IRQ resources. Configures the IOMUX to enable CSPI I/O pins, requests for IRQ and resets the hardware.
- Chip select function `mxc_spi_chipselect()`—Configures the hardware CSPI for the current SPI device. Sets the word size, transfer mode, data rate for this device.
- SPI transfer function `mxc_spi_transfer()`—Handles data transfers operations.
- SPI setup function `mxc_spi_setup()`—Initializes the current SPI device.
- SPI driver ISR `mxc_spi_isr()`—Called when the data transfer operation is completed and an interrupt is generated.

28.2.4 CSPI Synchronous Operation

Figure 28-3 shows how the CSPI provides synchronous read/write operations.

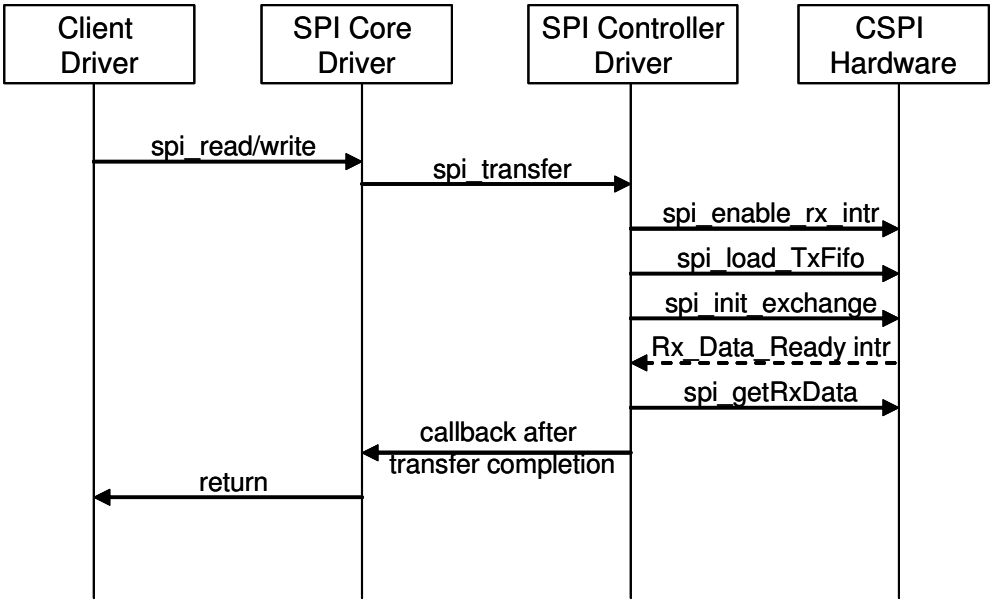


Figure 28-3. CSPI Synchronous Operation

28.3 Driver Features

The CSPI module supports the following features:

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

28.4 Source Code Structure

Table 28-1 shows the source files available in the devices directory:

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

Table 28-1. CSPI Driver Files

File	Description
mxc_spi.c	SPI Master Controller driver

28.5 Menu Configuration Options

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

- CONFIG_SPI—Build support for the SPI core. In menuconfig, this option is available under

Device Drivers > SPI Support.

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

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

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

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

  Device Drivers > SPI Support > LOOPBACK Testing of CSPIs.

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

28.6 Programming Interface

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

28.7 Interrupt Requirements

The SPI interface generates interrupts. CSPI interrupt requirements are listed in Table 28-2.

Table 28-2. CSPI Interrupt Requirements

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

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

Chapter 29
1-Wire Driver

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

29.1 Hardware Operation

The 1-Wire interface is used to connect to a battery monitor (Dallas DS2438Z). The 1-Wire interface reads battery current, voltage and other information. The EVK board by default does not support 1-Wire. Refer to the *MCIMX51 Multimedia Applications Processor Reference Manual* (MCIMX51RM) for more information.

29.2 Software Operation

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

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

29.3 Driver Features

The 1-Wire implementation supports the following features:

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

29.4 Source Code Structure

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

Table 29-1. 1-Wire Driver Files

File	Description
mxc_w1.c	1-Wire function implementation

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

29.5 Menu Configuration Options

In order to get to the one-wire configuration, use the command `./ltib -c` when located in the <`ltib dir`>. On the screen displayed, select **Configure the kernel** and exit. When the next screen appears select the following options to enable the 1-Wire driver:

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

Chapter 30
MMC/SD/SDIO Host Driver

The MultiMediaCard (MMC)/ Secure Digital (SD)/ Secure Digital Input Output (SDIO) Host driver implements a standard Linux driver interface to the enhanced MMC/SD host controller (eSDHC). The host driver is part of the Linux kernel MMC framework.

The MMC driver has following features:

- 1-bit or 4-bit operation for MCC/SD and SDIO cards
- 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 eSDHC module support MMC along with SD memory and I/O functions. The eSDHC controls the MMC, SD memory, and I/O cards by sending commands to cards and performing data accesses to and from the cards. The SD memory card system defines two alternative communication protocols: SD and SPI. The eSDHC only support the SD bus protocol.

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

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

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

For receiving data, the steps are as follows:

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

To transmitting data, the steps are as follows:

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

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

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

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 the eSDHC.

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

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

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

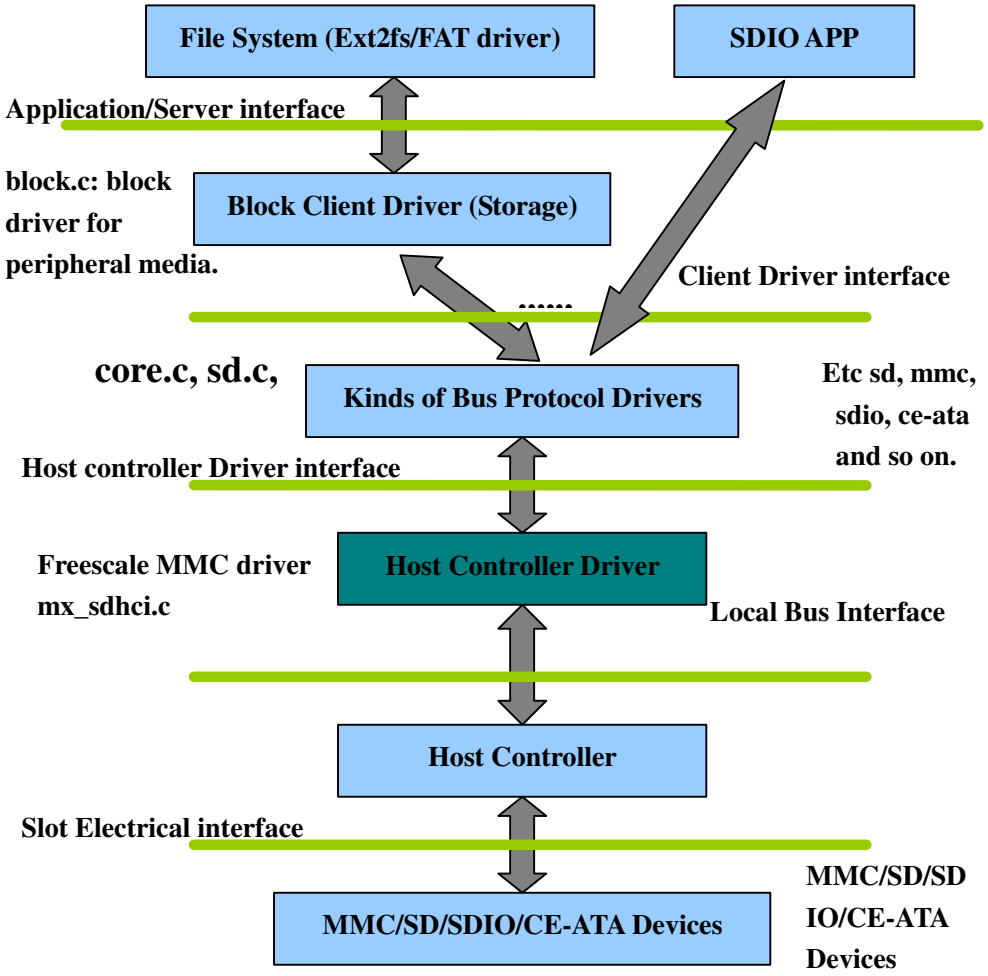


Figure 30-1. MMC Drivers Layering

30.3 Driver Features

The MMC driver supports the following features:

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

30.4 Source Code Structure

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

Table 30-1. MMC/SD Driver Files

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

30.5 Menu Configuration Options

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

- CONFIG_MMC—Build support for the MMC bus protocol. In menuconfig, this option is available under

  Device Drivers > MMC/SD/SDIO Card support

  By default, this option is Y
- CONFIG_MMC_BLOCK—Build support for MMC block device driver, which can be used to mount the file system. In `menuconfig`, this option is available under

  Device Drivers > MMC/SD Card Support > MMC block device driver support

  By default, this option is Y
- CONFIG_MMC_IMX_ESDHCI—Driver used for the i.MX eSDHC ports. In menuconfig, this option is found under

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

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

  This option is dependent on CONFIG_MMC_IMX_ESDHCI. By default, this option is not set and DMA mode is used.
- CONFIG_MMC_UNSAFE_RESUME—Used for embedded systems which use a MMC/SD/SDIO card for rootfs. In menuconfig, this option is found under

  Device drivers > MMC/SD/SDIO Card Support > Allow unsafe resume

30.6 Programming Interface

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

Chapter 31
Universal Asynchronous Receiver/Transmitter (UART) Driver

The low-level UART driver interfaces the Linux serial driver API to all the UART ports. It has the following features:

- Interrupt-driven and SDMA-driven transmit/receive of characters
- Standard Linux baud rates up to 4 Mbps
- Transmit and receive characters with 7-bit and 8-bit character lengths
- Transmits one or two stop bits
- Supports `TIOCMGET` IOCTL to read the modem control lines. Only supports the constants `TIOCM_CTS` and `TIOCM_CAR`, plus `TIOCM_RI` in `DTE` mode only
- Supports `TIOCMSET` IOCTL to set the modem control lines. Supports the constants `TIOCM_RTS` and `TIOCM_DTR` only
- Odd and even parity
- XON/XOFF software flow control. Serial communication using software flow control is reliable when communication speeds are not too high and the probability of buffer overruns is minimal
- CTS/RTS hardware flow control—both interrupt-driven software-controlled hardware flow and hardware-driven hardware-controlled flow
- Send and receive break characters through the standard Linux serial API
- Recognizes frame and parity errors
- Ability to ignore characters with break, parity and frame errors
- Get and set UART port information through the `TIOCGSSERIAL` and `TIOCSSERIAL` TTY IOCTL. Some programs like `setserial` and `dip` use this feature to make sure that the baud rate was set properly and to get general information on the device.The UART type should be set to 52 as defined in the `serial_core.h` header file.
- Serial IrDA
- Power management feature by suspending and resuming the URT ports
- Standard TTY layer IOCTL calls

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

31.1 Hardware Operation

Refer to the *i.MX51 Multimedia Applications Processor Reference Manual* to determine the number of UART modules available in the device. Each UART hardware port is capable of standard RS-232 serial

communication and has support for IrDA 1.0. Each UART contains a 32-byte transmitter FIFO and a 32-half-word deep receiver FIFO. Each UART also supports a variety of maskable interrupts when the data level in each FIFO reaches a programmed threshold level and when there is a change in state in the modem signals. Each UART can be programmed to be in DCE or DTE mode.

31.2 Software Operation

The Linux OS contains a core UART driver that manages many of the serial operations that are common across UART drivers for various platforms. The low-level UART driver is responsible for supplying information such as the UART port information and a set of control functions to this core UART driver. These functions are implemented as a low-level interface between the Linux OS and the UART hardware. They cannot be called from other drivers or from a user application. The control functions used to control the hardware are passed to the core driver through a structure called `uart_ops`, and the port information is passed through a structure called `uart_port`. The low level driver is also responsible for handling the various interrupts for the UART ports, and providing console support if necessary.

Each UART can be configured to use DMA for the data transfer. These configuration options are provided in the `mxc_uart.h` header file. The user can specify the size of the DMA receive buffer. The minimum size of this buffer is 512 bytes. The size should be a multiple of 256. The driver breaks the DMA receive buffer into smaller sub-buffers of 256 bytes and registers these buffers with the DMA system. The DMA transmit buffer size is fixed at 1024 bytes. The size is limited by the size of the Linux UART transmit buffer (1024).

The driver requests two DMA channels for the UARTs that need DMA transfer. On a receive transaction, the driver copies the data from the DMA receive buffer to the TTY Flip Buffer.

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

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

31.3 Driver Features

The UART driver supports the following features:

- Baud rates up to 4 Mbps
- Recognizes frame and parity errors only in interrupt-driven mode; does not recognize these errors in DMA-driven mode
- Sends, receives and appropriately handles break characters
- Recognizes the modem control signals
- Ignores characters with frame, parity and break errors if requested to do so

- Implements support for software and hardware flow control (software-controlled and hardware-controlled)
- Get and set the UART port information; certain flow control count information is not available in hardware-driven hardware flow control mode
- Implements support for Serial IrDA
- Power management
- Interrupt-driven and DMA-driven data transfer

31.4 Source Code Structure

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

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

Table 31-1. UART Driver Files

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

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

Table 31-2. UART Global Header Files

File	Description
<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/include/mach/mxc_uart.h	UART header that contains UART configuration data structure definitions
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51/board-mx51_3stack.h	Contains UART board specific configuration options

The source files, `serial.c and serial.h`, are associated with the UART driver that is available in the directory: `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51`. The source file contains UART configuration data and calls to register the device with the platform bus.

31.5 Configuration

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

31.5.1 Menu Configuration Options

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

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

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

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

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

  By default, this option is Y.

31.5.2 Source Code Configuration Options

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

31.5.2.1 Chip Configuration Options

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

31.5.2.2 Board Configuration Options

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

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51/board-mx51_3stack.h:`

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

31.6 Programming Interface

The UART driver implements all the methods required by the Linux serial API to interface with the UART port. The driver implements and provides a set of control methods to the Linux core UART driver. For more information about the methods implemented in the driver, see the API document.

31.7 Interrupt Requirements

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

The system requirements are listed in Table 31-3.

Table 31-3. UART Interrupt Requirements

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

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

31.8 Device Specific Information

31.8.1 UART Ports

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

31.8.2 Board Setup Configuration

Table 31-4. UART General Configuration

Platform	Number of UARTs	Max Baudrate
i.MX51	3	4 Mbps

Table 31-5. UART Active/Inactive Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	1	1	1	0	0	—

Table 31-6. UART IRDA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	NO_IRDA	NO_IRDA	NO_IRDA	NO_IRDA	NO_IRDA	—

Table 31-7. UART Mode Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	MODE_DCE	MODE_DCE	MODE_DTE	N/A	N/A	—
i.MX51	MODE_DCE	MODE_DCE	MODE_DCE	N/A	N/A	—

Table 31-8. UART Shared Peripheral Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	-1	-1	SPBA_UART3	-1	-1	—

Table 31-9. UART Hardware Flow Control Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	1	0	1	N/A	N/A	—
i.MX51	1	1	1	N/A	N/A	—

Table 31-10. UART DMA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	1	1	1	N/A	N/A	—

Table 31-11. UART DMA RX Buffer Size Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	1024	1024	1024	N/A	N/A	—

Table 31-12. UART UCR4_CTSTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	16	16	16	N/A	N/A	—

Table 31-13. UART UFCR_RXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	16	16	16	N/A	N/A	—

Table 31-14. UART UFCR_TXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	16	16	16	N/A	N/A	—

Table 31-15. UART Interrupt Mux Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	INTS_MUXED	INTS_MUXED	INTS_MUXED	N/A	N/A	—

Table 31-16. UART Interrupt 1 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	MXC_INT_UART1	MXC_INT_UART2	MXC_INT_UART3	N/A	N/A	—

Table 31-17. UART Interrupt 2 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	-1	-1	-1	N/A	N/A	—

Table 31-18. UART interrupt 3 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX51	-1	-1	-1	N/A	N/A	—

31.9 Early UART Support

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

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

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

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

Chapter 32
Bluetooth Driver

The Bluetooth driver provides synchronous and asynchronous wireless connection among multiple devices. The synchronous oriented channel provides voice transmission. The asynchronous channel allows more time delay in data transmission. The synchronous and asynchronous data transfer between the host and Bluetooth chip is performed by different hardware interfaces. The SSI interface is used to transfer voice from the host to the Bluetooth chip. UART or USB is used for asynchronous data communication.

Based on the wireless connection, many services can be supported by profiles defined by the Bluetooth Group. On the i.MX platform, the A2DP and AVRCP profile is used to play music (mp3, wav, and so forth). The FTP profile provides access to the file system on another device. The SPP profile emulates a serial cable to provide a simply implemented wireless replacement for the existing RS-232 based serial communications applications. The handset profile is reserved for future support, so the SSI interface is reserved. The UART interface is used for communication between the host and the Bluetooth chip.

32.1 Hardware Operation

The platform uses the APM6628, which is a Bluetooth and Wi-Fi combination module that integrates CSR Bluetooth and the Wi-Fi chip. The Bluetooth/Wi-Fi chip in the APM6628 module works independently.

32.2 Software Operation

BlueCore™ Host Software (BCHS) is a Bluetooth protocol provided by a third-party company, Cambridge Silicon Radio (CSR). The porting of BCHS to Linux is divided into:

- A **user space port**, in which the BCHS protocol stack runs in user space together with the application.
- A **kernel space port**, in which the BCHS protocol stack runs in kernel space and the application runs in user space.

There are two ways to set up the user space port:

- The application and the BCHS protocol stack are running within the same process.
- The application and the BCHS protocol stack are running in two different processes.

In i.MX platform, the BCHS protocol stack runs in user space. And the application runs in the same process, as shown in Figure 32-1.

Encoding is used to minimize the bandwidth required for transferring the audio data. Thus, the encoding compresses the audio before transmission over the air. The A2DP profile mandates support for SBC encoding, and other codecs, such as MP3 and WMA, are optional. The A2DP source checks the capabilities of sink and then configures sink to select the dedicated codec.

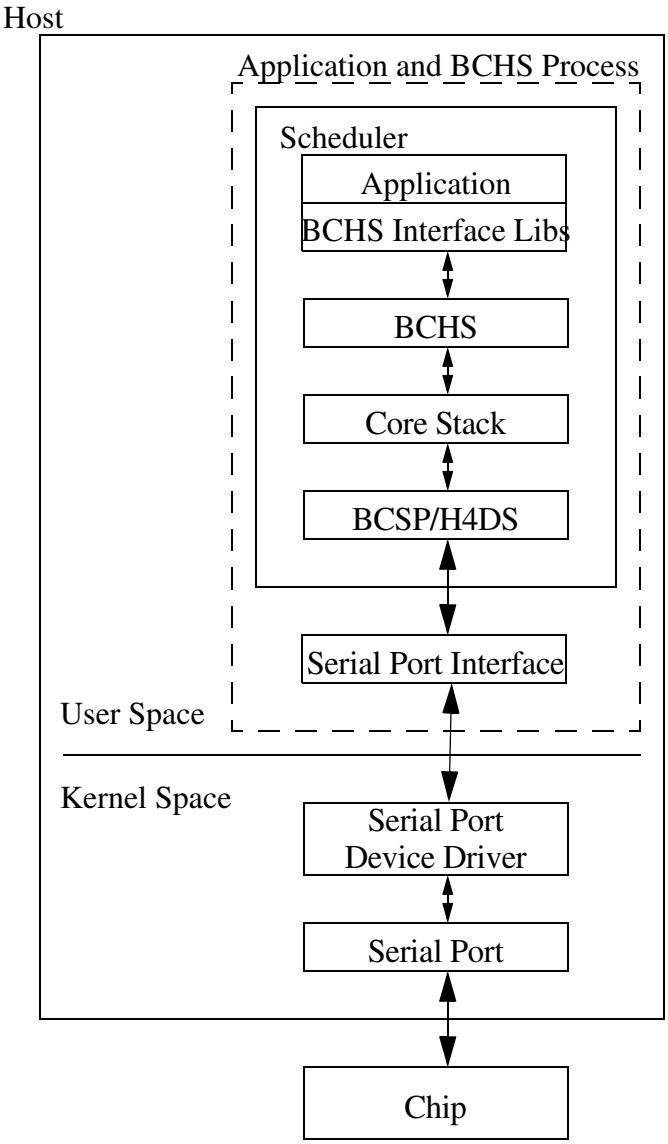


Figure 32-1. BCHS Protocol Stack

32.2.1 UART Control

For user space porting, first configure the universal asynchronous receiver transmitter (UART). On the i.MX platforms, UART2 is used for communication between the CPU chip and the Bluetooth chip. The BCHS protocol opens `/dev/ttymxc1` and configures the device according to profile requirements.

The minimum baud rate for the A2DP profile is 460.8 kbps; 921.6 kbps baudrate is recommended. Table 32-1 maps the relationship between the UART baud rate and maximum SBC bit rate.

Table 32-1. UART Mapping
Table 32-2.

Baud Rate (kbps)	Max SBC bit rate (kbps)
115.2	75
230.4	150
460.8	300
600.0	400
921.6	410

The following table describes the UART configuration files.

32.2.2 Reset and Power control

Besides BCHS and UART, the power control and reset for BT chip is also required. Table 32-3 lists the file for the driver.

Table 32-3. Bluetooth Driver File
Table 32-4.

File	Description
`<ltib_dir>/rpm/BUILD/linux/drivers/mxc/bt/mxc_bt.c`	bluetooth kernel driver

32.2.3 Configuration

To get to the Bluetooth 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 Linux kernel configuration option CONFIG_MXC_BLUETOOTH is provided for the MXC processors. In the menuconfig this option is available under `Device Drivers -> MXC support drivers -> MXC Bluetooth support -> MXC Bluetooth support`. By default, this option is M for all architectures.

Chapter 33
ATA Driver

The ATA module is an AT attachment host interface mainly used to interface with hard disk devices. The ATA driver is compliant to the ATA-6 standard, and supports the following protocols:

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

33.1 Hardware Operation

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

33.2 Software Operation

33.2.1 ATA Driver Architecture

Figure 33-1 shows ATA driver architecture. File systems are built upon the block device. The integrated internal DMA engine handles the DMA transmission through the ATA Controller driver.

The DMA engine used depends on chip capability. See Table 33-1 for detailed information.

Table 33-1. DMA Engine

DMA Engine Type	Platform
Internal DMA	i.MX51

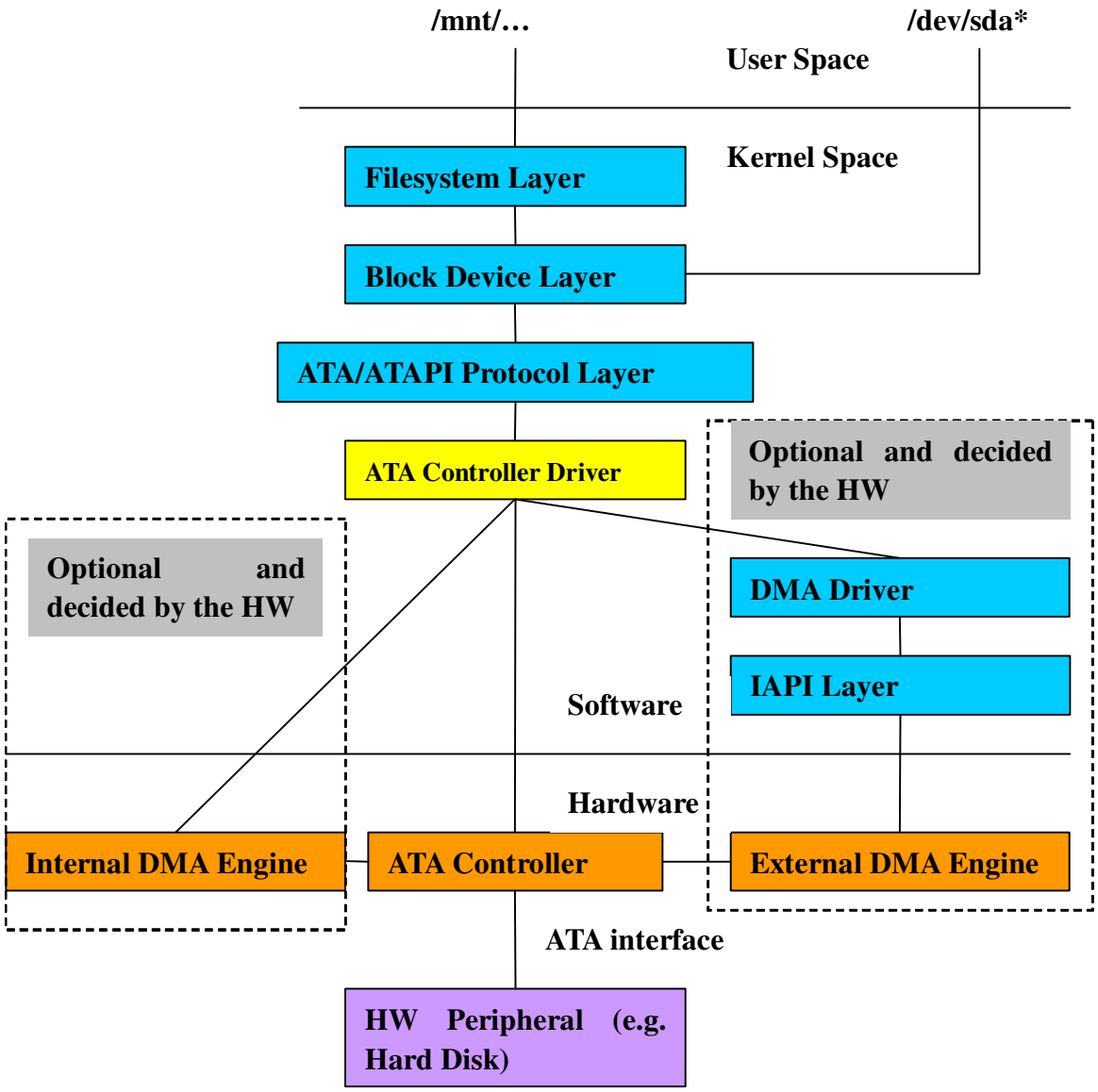


Figure 33-1. ATA Driver Layers

33.2.2 LibATA Driver

LibATA is a library used inside the Linux kernel to support ATA host controllers and devices. libATA provides an ATA driver API, class transports for ATA and ATAPI devices, and SCSI <-> ATA translation for ATA devices according to the T10 SAT specification driver. The hard disk is exposed to the application in user space by the /dev/sda interface.

33.3 Source Code Structure Configuration

33.3.1 LibATADriver

Table 33-2 lists the source file available in the directory, `<ltib_dir>/rpm/BUILD/linux/drivers/ata`

Table 33-2. LibATA Driver File List

File	Description
pata_fsl.c	ATA Driver Implementation file

33.4 Linux Menu Configuration Option

Enable these kernel configuration options as either modules (M) or built-in to the kernel (Y). These options are all under Device Drivers > Serial ATA (prod) and Parallel ATA (experimental) drivers > Freescale on-chip PATA support:

For ATA device support, enable these options: Device Drivers > SCSI device support > SCSI disk support.

33.5 Board Configuration Options

Table 33-3 lists the hardware configurations for 3-Stack boards:

Table 33-3. Hardware Configuration for 3-Stack Boards

Platform	Hardware Configuration
MX51	• Test the i.MX51 P-ATA driver using an NFS mount of the root file system, due to certain pin conflicts between the P-ATA and the NAND flash. Short JP30 in the i.MX51 personality board before loading ATA modules for P-ATA tests. When using the NAND driver, keep JP30 open.

Chapter 34
ARC USB Driver

The universal serial bus (USB) driver implements a standard Linux driver interface to the ARC USB-HS OTG controller. The USB provides a universal link that can be used across a wide range of PC-to-peripheral interconnects. It supports plug-and-play, port expansion, and any new USB peripheral that uses the same type of port.

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

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

34.1 Architectural Overview

A USB host system is composed of a number of hardware and software layers. Figure 34-1 shows a conceptual block diagram of the building block layers in a host system that support USB 2.0.

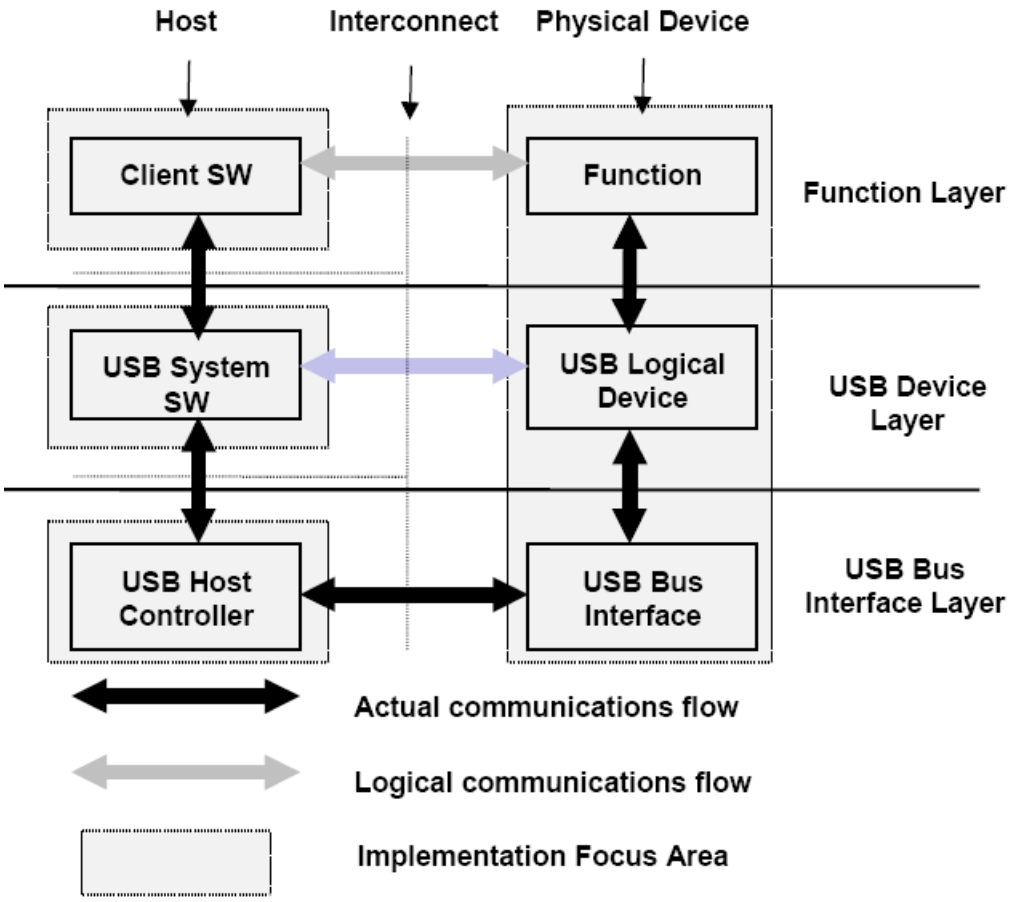


Figure 34-1. USB Block Diagram

34.2 Hardware Operation

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

34.3 Software Operation

The Linux OS contains a USB driver, which implements the USB protocols. For the USB host, it only implements the hardware specified initialization functions. For the USB peripheral, it implements the gadget framework.

```
static struct usb_ep_ops fsl_ep_ops = {
        .enable = fsl_ep_enable,
        .disable = fsl_ep_disable,
        .alloc_request = fsl_alloc_request,
        .free_request = fsl_free_request,
```

```
        .queue = fsl_ep_queue,
        .dequeue = fsl_ep_dequeue,
        .set_halt = fsl_ep_set_halt,
        .fifo_status = arcotg_fifo_status,
        .fifo_flush = fsl_ep_fifo_flush,            /* flush fifo */
        };
static struct usb_gadget_ops fsl_gadget_ops = {
        .get_frame = fsl_get_frame,
        .wakeup = fsl_wakeup,
/*      .set_selfpowered = fsl_set_selfpowered, */   /* Always selfpowered */
        .vbus_session = fsl_vbus_session,
        .vbus_draw = fsl_vbus_draw,
        .pullup = fsl_pullup,
        };
```

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

For OTG, an OTG finish state machine (FSM) is implemented.

34.4 Driver Features

The USB stack supports the following features:

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

34.5 Source Code Structure

Table 34-1 shows the source files available in the source directory, `<ltib_dir>/rpm/BUILD/linux/drivers/usb`.

Table 34-1. USB Driver Files

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

Table 34-2 shows the platform related source files.

Table 34-2. USB Platform Source Files

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

Table 34-3 shows the platform-related source files in the directory:
`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx51/`

Table 34-3. USB Platform Header Files

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

Table 34-4 shows the common platform source files in the directory:

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

Table 34-4. USB Common Platform Files

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

34.6 Menu Configuration Options

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

- CONFIG_USB—Build support for USB
- CONFIG_USB_EHCI_HCD—Build support for USB host driver. In menuconfig, this option is available under

  Device drivers > USB support > EHCI HCD (USB 2.0) support.

  By default, this option is M.
- CONFIG_USB_EHCI_ARC—Build support for selecting the ARC EHCI host. In menuconfig, this option is available 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 Host-side USB > EHCI HCD (USB 2.0) support > Support for Freescale controller.

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

  Device drivers > USB support > Use IRAM for USB.

  By default, this option is N.
- CONFIG_USB_EHCI_ROOT_HUB_TT—Build support for OHCI or UHCI companion. In menuconfig, this option is available under

  Device drivers > USB support > Root Hub Transaction Translators.

  By default, this option is Y selected by USB_EHCI_FSL && USB_SUPPORT.

- CONFIG_USB_STORAGE—Build support for USB mass storage devices. In menuconfig, this option is available under

  Device drivers > USB support > USB Mass Storage support.

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

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

  By default, this option is M.
- CONFIG_USB_HIDINPUT—Build support for USB HID input devices. In menuconfig, this option is available under

  Device drivers > HID devices.

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

  Device drivers > USB support > USB Gadget Support.

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

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

  By default, this option is Y.
- CONFIG_USB_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.

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

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

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

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

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

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

By default, this option is Y.

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

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

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

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

  By default, this option is M.

34.7 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.

34.8 Default USB Settings

Table 34-5 shows the default USB settings.

Table 34-5. Default USB Settings

Platform	OTG HS	OTG FS	Host1	Host2(HS)	Host2(FS)
i.MX51 3-Stack	enable	N/A	enable (HS)	N/A	N/A
i.MX51 EVK	enable	N/A	enable (HS)	N/A	N/A

To use the i.MX51 3-Stack USBOTG port, ensure the following hardware settings are done:

- Use min USB port (J26) nearby the debug board with the marker “USB OTG”.
- Keep J28 open for all USB gadget tests.
- Short J28 for USB host tests

To use the i.MX51 3-Stack USBHOST1 port, ensure the following hardware settings are done:

- Use min USB port (J19) nearby SD card slot with the marker “USB OTG ULPI”
- Short pin 2 and pin 3 of J21. Keep J20 open

The default configuration only enables USB-OTG device mode. To enable USB-OTG for both host and device mode, configure the kernel as follows and rebuild the kernel and modules:

- CONFIG_USB_EHCI_HCD—EHCI HCD (USB 2.0) support: built as M
- CONFIG_USB_GADGET—USB Gadget Support: built as M
- CONFIG_USB_OTG —OTG Support: built as M
- CONFIG_UTMI_MXC_OTG—USB OTG pin detect support for UTMI PHY: built as M

Then insert the following modules:

```
modprobe fsl_arc_otg
modprobe ehci_hcd
```

```
modprobe arcotg_udc
```

Chapter 35
Secure Real Time Clock (SRTC) Driver

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

35.1 Hardware Operation

The SRTC is a real time clock with enhanced security capabilities. It provides an accurate, constant time, regardless of the main system power state and without the need to use an external on-board time source, such as an external RTC. The SRTC can wake up the system when a pre-set alarm is reached.

35.2 Software Operation

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

35.2.1 IOCTL

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

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

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

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

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

35.2.2 Keep Alive in the Power Off State

To keep preserve the time when the device is in the power off state, the SRTC clock source should be set to CKIL and the voltage input, NVCC_SRTC_POW, should remain active. Usually these signals are connected to the PMIC and software can configure the PMIC registers to enable the SRTC clock source and power supply. For example, CKIL and NVCC_SRTC_POW can be connected to the MC13892 CLK32KMCU and VSRTC. Bit 4, DRM, of the MC13892 Power Control 0 Register can be enabled to keep VSRTC and CLK32KMCU on for all states.

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

35.3 Driver Features

The SRTC driver includes the following features:

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

35.4 Source Code Structure

The RTC module is implemented in the following directory:

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

Table 35-1 shows the RTC module files.

Table 35-1. RTC Driver Files

File	Description
rtc-mxc_v2.c	SRTC driver implementation file

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

35.5 Menu Configuration Options

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

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

Chapter 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, the WDOG times out. Upon a time-out, the WDOG either asserts the `wdog_b` signal or a `wdog_rst_b` system reset signal, depending on software configuration. The watchdog module cannot be deactivated once it is activated.

36.2 Software Operation

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

36.3 Generic WDOG Driver

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

36.3.1 Driver Features

This WDOG implementation includes the following features:

- Generates the reset signal if it is enabled but not serviced within a predefined timeout value (defined in milliseconds in one of the WDOG source files)
- Does not generate the reset signal if it is serviced within a predefined timeout value
- Provides IOCTL/read/write required by the standard WDOG subsystem

36.3.2 Menu Configuration Options

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

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

36.3.3 Source Code Structure

Table 36-1 shows the source files for WDOG drivers that are in the following directory:

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

Table 36-1. WDOG Driver Files

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

Watchdog system reset function is located under

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

36.3.4 Programming Interface

The following IOCTLs are supported in the WDOG driver:

- WDIOC_GETSUPPORT
- WDIOC_GETSTATUS
- WDIOC_GETBOOTSTATUS
- WDIOC_KEEPALIVE
- WDIOC_SETTIMEOUT
- WDIOC_GETTIMEOUT

For detailed descriptions about these IOCTLs, see

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

Chapter 37
Pulse-Width Modulator (PWM) Driver

The pulse-width modulator (PWM) has a 16-bit counter and is optimized to generate sound from stored sample audio images and generate tones. The PWM has 16-bit resolution and uses a 4×16 data FIFO to generate sound. The software module is composed of a Linux driver that allows privileged users to control the backlight by the appropriate duty cycle of the PWM Output (PWMO) signal.

37.1 Hardware Operation

Figure 37-1 shows the PWM block diagram.

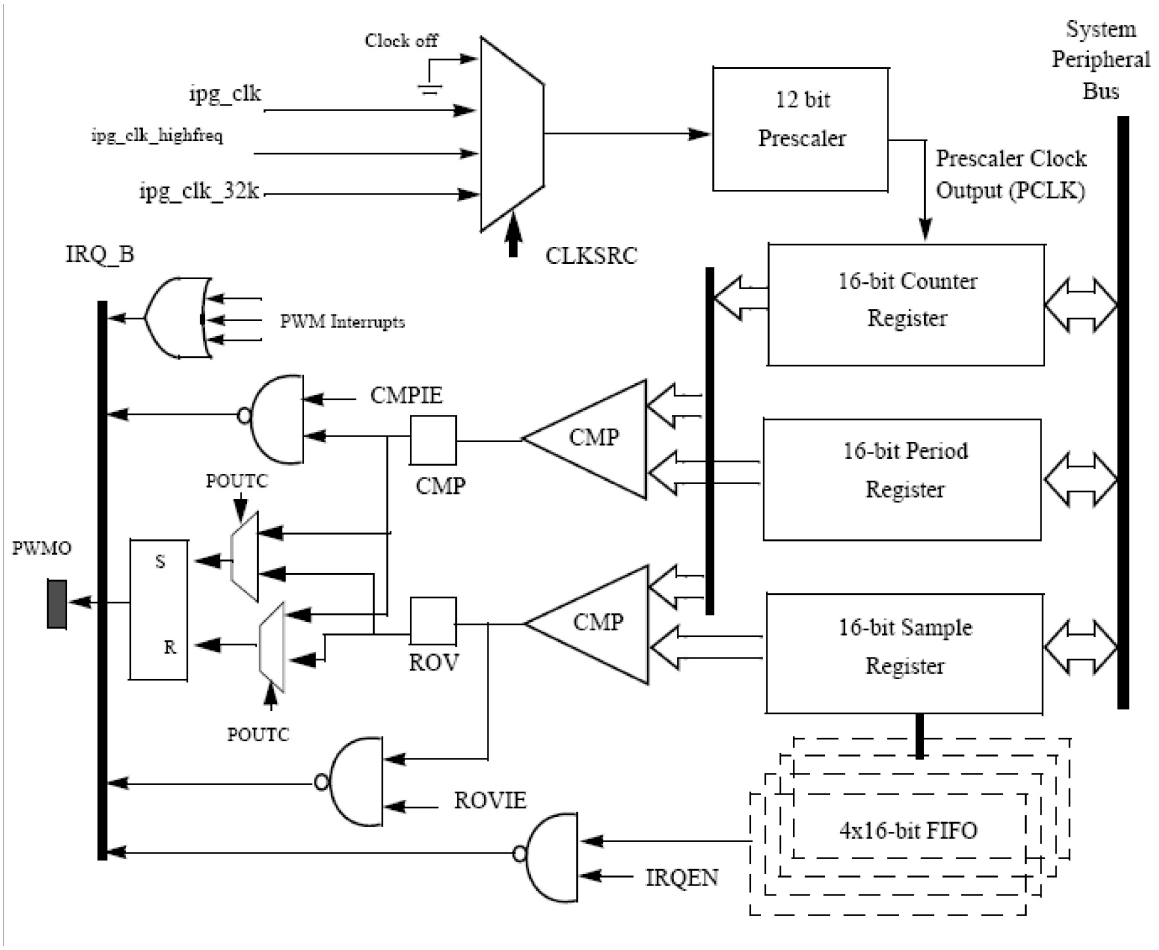


Figure 37-1. PWM Block Diagram

The PWM follows IP Bus protocol for interfacing with the processor core. It does not interface with any other modules inside the device except for the clock and reset inputs from the Clock Control Module (CCM) and interrupt signals to the processor interrupt handler. The PWM includes a single external output signal, PMWO. The PWM includes the following internal signals:

- Three clock inputs
- Four interrupt lines
- One hardware reset line
- Four low power and debug mode signals
- Four scan signals
- Standard IP slave bus signals

37.2 Clocks

The clock that feeds the prescaler can be selected from:

- High frequency clock—provided by the CCM. The PWM can be run from this clock in low power mode.
- Low reference clock—32 KHz low reference clock provided by the CCM. The PWM can be run from this clock in the low power mode.
- Global functional clock—for normal operations. In low power modes this clock can be switched off.

The clock input source is determined by the CLKSRC field of the PWM control register. The CLKSRC value should only be changed when the PWM is disabled.

37.3 Software Operation

The PWM device driver reduces the amount of power sent to a load by varying the width of a series of pulses to the power source. One common and effective use of the PWM is controlling the backlight of a QVGA panel with a variable duty cycle.

Table 37-1 provides a summary of the interface functions in source code.

Table 37-1. PWM Driver Summary

Function	Description
struct pwm_device *pwm_request(int pwm_id, const char *label)	Request a PWM device
void pwm_free(struct pwm_device *pwm)	Free a PWM device
int pwm_config(struct pwm_device *pwm, int duty_ns, int period_ns)	Change a PWM device configuration
int pwm_enable(struct pwm_device *pwm)	Start a PWM output toggling
int pwm_disable(struct pwm_device *pwm)	Stop a PWM output toggling

The function `pwm_config()` includes most of the configuration tasks for the PWM module, including the clock source option, and period and duty cycle of the PWM output signal. It is recommended to select the

peripheral clock of the PWM module, rather than the local functional clock, as the local functional clock can change.

37.4 Driver Features

The PWM driver includes the following software and hardware support:

- Duty cycle modulation
- Varying output intervals
- Two power management modes—full on and full of

37.5 Source Code Structure

Table 37-2 lists the source files and headers available in the following directories:

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

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

Table 37-2. PWM Driver Files

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

37.6 Menu Configuration Options

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

- System Type > Enable PWM driver and select the following option to enable the Backlight driver:

  Device Drivers > Graphics support > Backlight & LCD device support > Generic PWM based Backlight Driver

Chapter 38
FM Driver

Si4702 is used as the FM chip on the board. The Si4702 extends Silicon Laboratories Si4700 FM tuner family and further increases the ease and attractiveness of adding FM radio reception to mobile devices through small size and board area, minimum component count, and flexible programmability. A headset cable is used for antenna on the board.

38.1 FM Overview

The Si4702 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 2.7–5.5 V battery. The features of the FM module are as follows:

- Worldwide FM band support (76-108 MHz)
- Digital low-IF receiver
- Seek tuning
- Automatic frequency control (AFC)
- Automatic gain control (AGC)
- Signal strength measurement
- Adaptive noise suppression
- Volume control
- 32.768 KHz reference clock
- 2-wire and 3-wire control interface
- 2.7 to 5.5 V supply voltage
- Integrated LDO regulator allows direct connection to battery
- Integrated crystal oscillator

38.1.1 Hardware Operation

Si4702 supports both three-wire control and two-wire control. Two-wire control is chosen by driving SEN pin high during boot up.

For two-wire operation, a transfer begins with the START condition. The control word is latched internally on rising SCLK edges and is eight bits in length: a seven bit device address equal to 0010000b and a read/write bit (write = 0 and read = 1). The device acknowledges the address by setting SDIO low on the next falling SCLK edge.

For write operations, the device acknowledge is followed by an eight bit data word latched internally on rising edges of SCLK. The device always acknowledges the data by setting SDIO low on the next falling SCLK edge. An internal address counter automatically increments to allow continuous data byte writes, starting with the upper byte of register 02h, followed by the lower byte of register 02h, and onward until the lower byte of the last register is reached. The internal address counter then automatically wraps around to the upper byte of register 00h and proceeds from there until continuous writes cease. Data transfer ceases with the STOP command. After every STOP command, the internal address counter is reset.

For read operations, the device acknowledge is followed by an eight bit data word shifted out on falling SCLK edges. An internal address counter automatically increments to allow continuous data byte reads, starting with the upper byte of register 0Ah, followed by the lower byte of register 0Ah, and onward until the lower byte of the last register is reached. The internal address counter then automatically wraps around to the upper byte of register 00h and proceeds from there until continuous reads cease. After each byte of data is read, the controller IC returns an acknowledge if an additional byte of data is requested. Data transfer ceases with the STOP command. After every STOP command, the internal address counter is reset.

FM analog signals connect directly to the audio chip which routes them out to the headset.

38.1.2 Software Operation

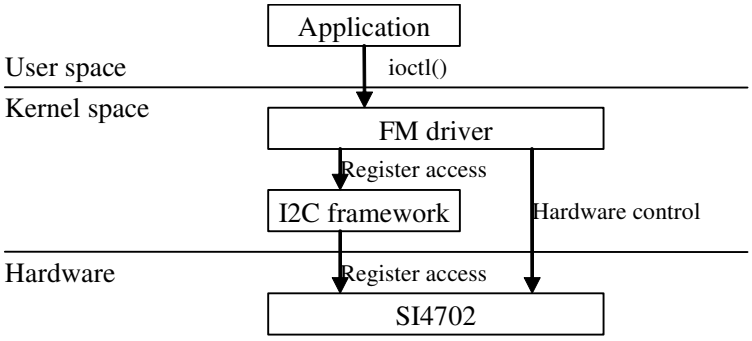


Figure 38-1. FM Driver Software Operation

The FM driver serves as an interface between kernel and user space. The driver can control hardware directly (for example, a reset operation) but most of the functional operation comes from the I^2C framework, which is especially convenient in the 2-wire control case.

The main software operation is as follows:

1. In initialization stage, register device in character sub-system, and then register it to I^2C framework.
2. In open operation, reset the chip, and initialize the register on the chip.
3. In release operation, shutdown the chip.
4. In ioctl operation, handle all the commands from user space, execute them and then feed back information if there is any.

38.2 Source Code Structure Configuration

Table 38-1 lists the source files associated with the FM driver that are available in the directory,

```
<ltib_dir>/rpm/BUILD/linux/drivers/char/mxc_si4702.c
<ltib_dir>/rpm/BUILD/linux/include/linux/mxc_si4702.h
```

Table 38-1. FM Driver Source and Header File List

File	Description
mxc_si4702.c	Source file for SI4702 FM driver
mxc_si4702.h	Header file for SI4702 FM driver

38.3 Linux Menu Configuration Options

The Linux kernel configurations are provided for this module. CONFIG_FM_SI4702 is the configuration option for the FM driver. By default, this option is M.

To load the FM drivers use the command:

```
insmod mxc_si4702.ko
```

It is located in `/lib/modules/2.6.28-515-g4eec389/kernel/drivers/char.`

Chapter 39
Global Positioning System (GPS) Driver

39.1 GPS Driver Overview

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

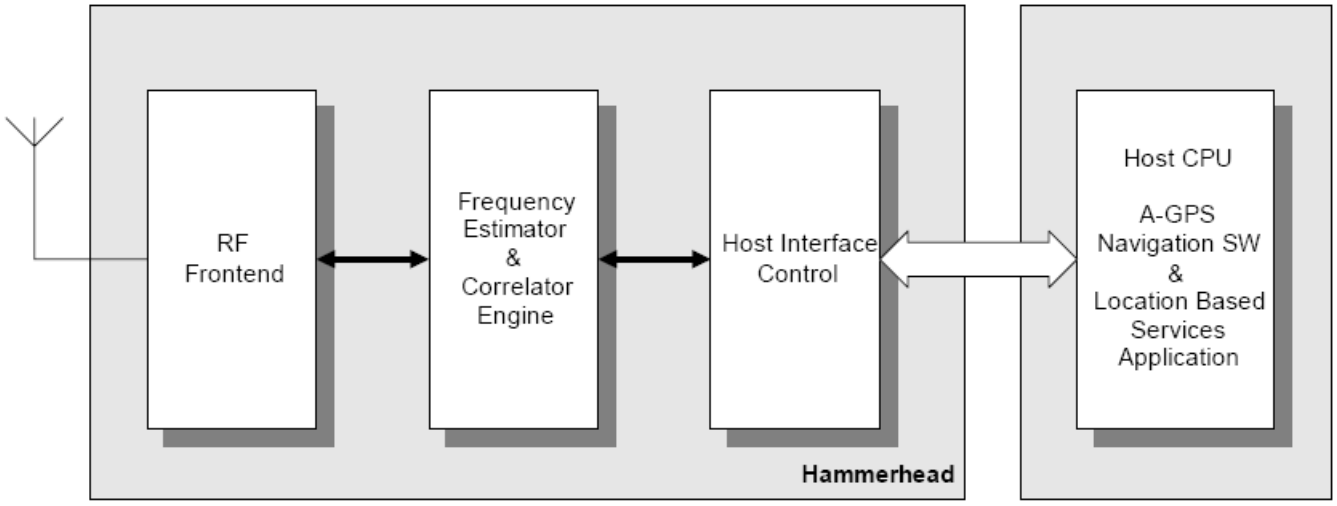


Figure 39-1. Barracuda GPS Coarse System Architecture Including Host CPU

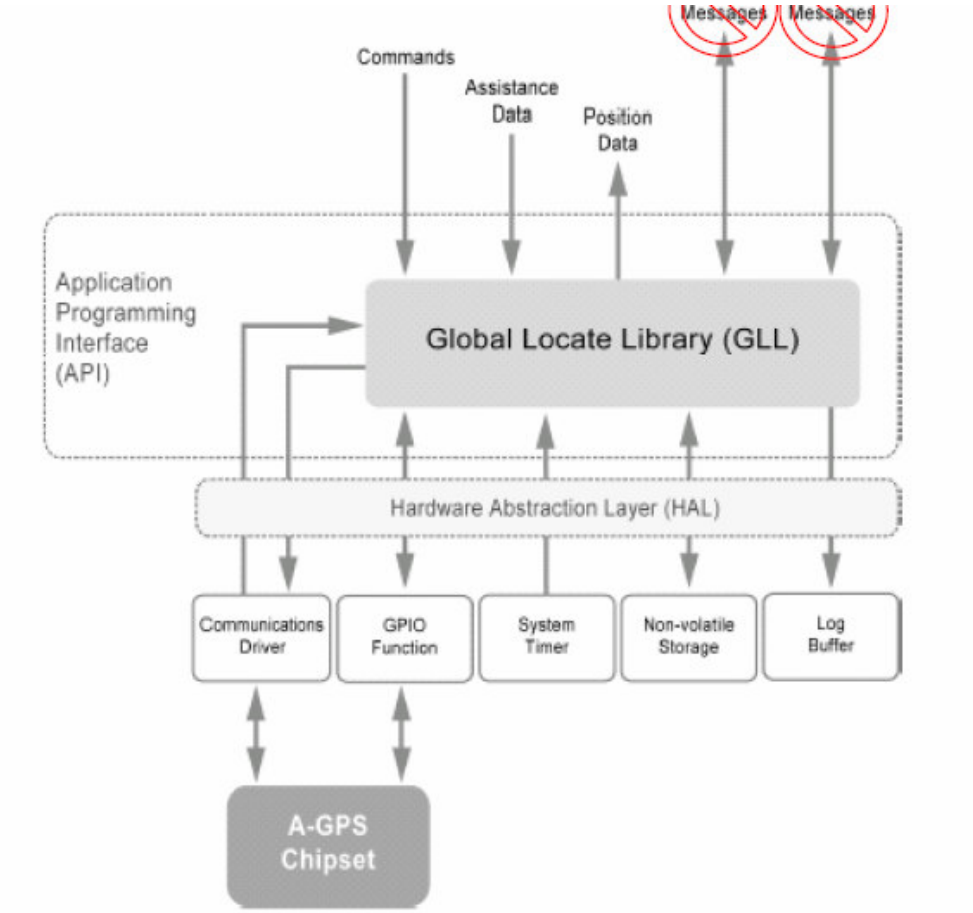


Figure 39-2. GL GPS SW Architecture

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, provides an accurate timer to the GPS core driver. Non-volatile storage is used to store assistance data and useful information for the last GPS position information received, which can accelerate the next position fix. A log buffer is generated by the GLL lib and tracks problems when integrating the GL GPS solution. The hardware abstraction layer (HAL) is used to abstract specific hardware platforms, which makes the GPS core driver hardware-independent. The core driver processes GPS data and controls the work flow with the GPS device. It accepts application requests and sends the GPS position information to the upper layer.

Most of the GPS software modules are provided in binary format only. BSP source code is available for the driver that handles the concrete hardware access, additional source code controls the GPS data flow when GPS is running.

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 connects to the UART2 slot of the platform hardware. Since GPS is very sensitive to timing accuracy, the platform provides a high-accuracy, non-drifting millisecond timer function to the GPS core drivers.

The GPS Control driver manages hardware-specific IOs to regulate power usage on the platform, power on/off and reset GPIO pins setting when the GPS is supported. Before GPS power up, the reset/init sequence is done. The reset pin is asserted for a few milliseconds, then de-asserted. The reset sequence is complete when the gps_gpiodrv.ko is loaded. When GPS is launched, the power pin is asserted.

39.2.1 UART Port

The GPS module uses UART2 to communicate with the host. The device name in the Linux system is shown in Table .

39.2.2 GPIO Control

GPIO pins are used to control the GPS module as shown in Table 39-2.

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 the assistance data shown in Table 39-2.

Software applications communicate with the GPS module through a NMEA pipe where the incoming NMEA data is read. The GPS application creates a NMEA pipe then receives the NMEA sentences from this pipe.

39.3.1 GLGPS Configuration

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

Example 39-1 shows the XML file.

Example 39-1. GLGPS Configuration

```
<?xml version="1.0" encoding="utf-8"?>
<glgps xmlns="http://www.glpals.com/" xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation="http://www.glpals.com/ glconfig.xsd" >

  <!--HAL Confguration  -->
  <hal acPortName="/dev/ttymxc2" lBaudRate="115200" cLogEnabled="true" acLogDirectory="./log"
ltoFileName="lto.dat"
       bPrintToConsole="false" />

  <!-- Parameters passed to GlEngine   -->
  <gll FrqPlan="FRQ_PLAN_26MHZ_2PPM" RfType="GL_RF_BARRACUDA"
       LogPriMask="LOG_DEBUG"
      LogFacMask="LOG_GLLAPI | LOG_DEVIA | LOG_NMEA | LOG_RAWDATA "
  />

  <!-- List of jobs can be performed by the GPS controller    -->
      <!-- The default job all parameters are set to default values  -->
    <job id="normal">
      <task>
          <req_pos />
      </task>
    </job>

    <job id="cold">
      <task>
            <!-- Instructs GLL to ignore all elements stored in NVRAM listed below -->
           <startup ignore_time="true" ignore_osc="true" ignore_pos="true" ignore_nav="true"
ignore_ram_alm="true"  />
          <req_pos />
      </task>
    </job>
</glgps>
```

The <hal> tag defines the glhal settings (serial COM settings, enable hal log, some other paths). The parameters are explained in Table 39-4.

Table 39-4. hal Attributes

hal Attributes	Description	Comment
acPortName	Serial COM port	

Table 39-4. hal Attributes (continued)

hal Attributes	Description	Comment
lBaudRate	Baud rate	9600, 19200, 38400, 57600, 115200
acLogDirectory	Directory where the logs are placed	

The `<gll>` tag defines the gll specific parameters, as explained in Table 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	Should be GL_RF_BARRACUDA

Different tasks are configured in Example 39-1. The task named *normal* makes an autonomous fix every second and the task named 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

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

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/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 is valid for several days, giving users the advantages of assisted GPS performance while preserving the freedom of autonomous GPS operation.

The reference board should be connected through a NFS server. Put the LTO file in the folder pointed to by the GPS driver. It is the responsibility of the user to implement the method used to get a TCP/IP connection on the platform (Bluetooth, IP over USB).

39.3.4.1 Enabling The LTO Feature on the Platform

Contact Broadcom to obtain the necessary license for enabling this feature on the Freescale reference platform. Submit the platform name and platform type to obtain the license. Also, ensure that the LTO feature is enabled through the different input parameters of the GPS driver.

39.3.5 Power Management

The GPS driver automatically manages the power management of the Broadcom chip set by toggling the different IOs of the chip set. The GPS chip set enters low power standby mode when the GPS driver is stopped.

39.3.6 irm Commands

While the GLGPS is running, commands can be sent to the process. There is a special FIFO file in `/var/run/glgpsctrl` where the commands are sent. Commands are ASCII strings with the `$pglirm, prefix + command name` format.

- `$pglirm,req_pos,name,period,N1,fixcount,N2,validfix,N3,duration_sec,N4`

  Creates a periodic position request.
  - Name – Unique request identifier, it also used by GLHAL to output NMEA data
  - Period – Update period in milliseconds
  - Fixcount – Stop after that number of fixes (valid or invalid) reported
  - Validfix – Stop after that number of valid fixes reported
  - Duration_sec – Stop after that many seconds
- `$pglirm,req_pos_single,name,acc,timeout,N1`

  Creates a single shot request.
  - Name – Unique request identifier, it also used by GLHAL to output NMEA data
  - Acc – Accuracy QoS parameter
  - Timeout – Time-out QoS parameter
- `$pglirm,req_aid,name`

  Queries for what assistance data is missing.
  - Name – Unique request identifier, it also used by GLHAL to output NMEA data

- `$pglirm,factory,test,prn,17,timeout,16,GL_FACT_TEST_MODE,GL_FACT_TEST_CONT,GL_FACT_TEST_ITEMS,GL_FACT_TEST_WER`

  Creates a factory request.
  - Test – Unique request identifier, it also used by GLHAL to output NMEA data
  - Prn – PRN number
  - Timeout – How long to run test for
  - GL_FACT_TEST_MODE – Test mode [GL_FACT_TEST_ONCE|GL_FACT_TEST_CONT]
  - GL_FACT_TEST_ITEMS – What to test GL_FACT_TEST_CN0|GL_FACT_TEST_FRQ|GL_FACT_TEST_WER
- `$pglirm,startup,{[ignore_osc|ignore_rom_alm|ignore_rom_alm|ignore_pos|ignore_ram_alm|ignore_time],[true|false]}`

  Tells the GLCT to ignore specified elements of the data previously stored in nonvolatile storage.
- `$pglirm,stop,name`

  Stops an ongoing request with a name "name"; The name "all" is reserved to stop all ongoing requests.
- `$pglirm,quit`

  Causes GLCT to exit.
- `$pglirm,pwm,[on|off]`

  Turns auto power management on or off

As an example, to send quit command in a shell, do the following:

```
echo '$pglirm,quit' >/var/run/glgpsctrl
```

Chapter 40
SIM Driver

The SIM driver implements a Linux driver interface to the Subscriber Identification Module (SIM).

Table 40-1. Available Platforms

Module Name	Available Platform
SIM	i.MX51

The SIM driver has following features:

- Supports card insertion and removal events
- Supports T0 Smart Card, and compatible with the ISO7816-3 spec

40.1 Hardware Operation

The detailed hardware operation of SIM module is detailed in the hardware documentation.

40.2 Software Operation

The SIM interface driver package for the i.MX familiy consist of two basic parts: the SIM kernel interface driver and the a user space library to simplify development.

The kernel device driver is pretty much built around a finite state machine for received characters. This FSM has three mayor states: card removed, discovering and parsing the ATR (answer to reset) after card detection and data transfer (command/response/status) during normal operation. The second part is the interfacing to user space, implemented as ioctl() calls. Finally, the kernel driver is completed by functions for ramping up / shutting down or cold/warm reset SIM cards.

The user space library with it's API is pretty mach a wrapper around the ioctl-calls plus an event handler that ramps up or powers down the interface apon card detection or removal.

Device driver state machines and the present state

The present state reflects the abstracted state of the interface. It can be

- SIM_PRESENT_REMOVED No card inserted
- SIM_PRESENT_DETECTED Card has just been inserted, ATR is under way
- SIM_PRESENT_OPERATIONAL After ATR receiption, interface is in operational state

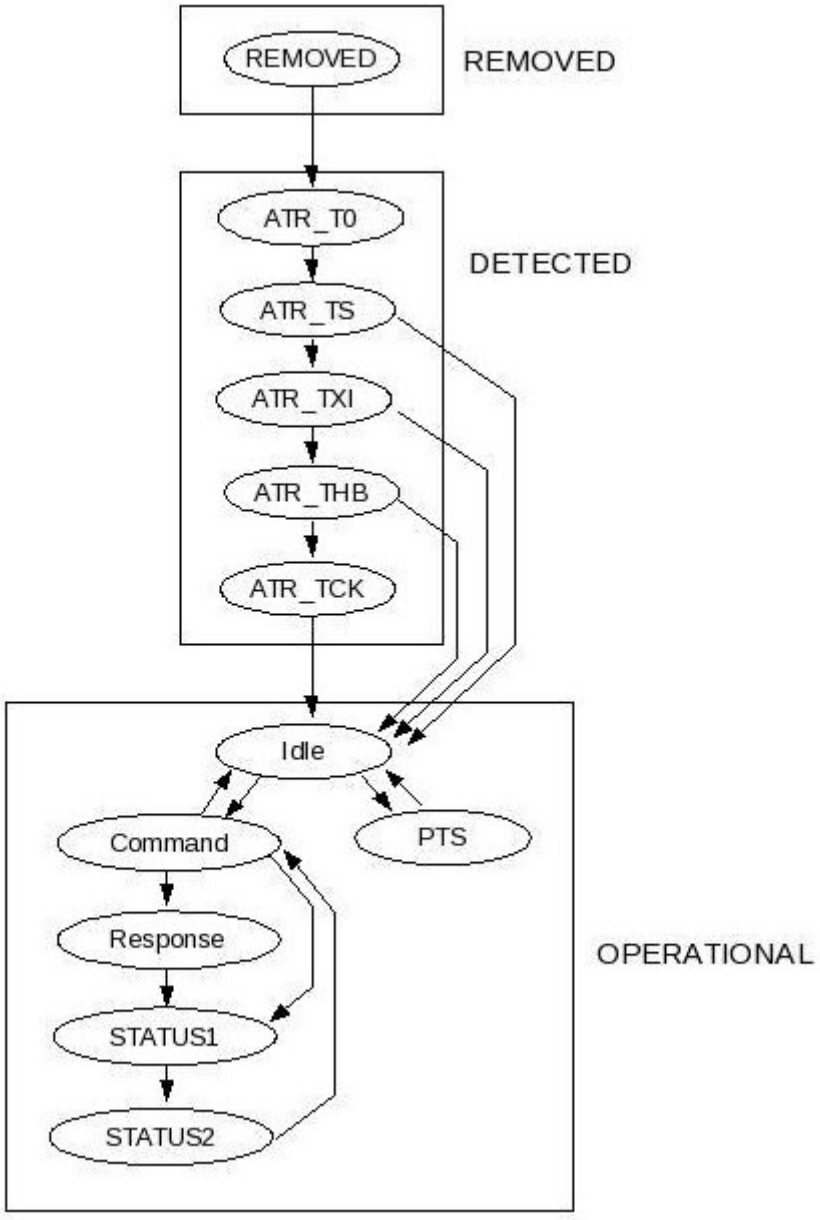


Figure 40-1. SIM driver

Right after card insertion, the driver is in "detected" state. The "detected" state holds a sequence of five sub-states which reflects the parsing of the ATR from T0, TS, the interface characters TXI (TA1, TB1, ... , TD4), historic bytes and the check sum. Certain states may be ommited while parsing the ATR in case they are not present in the answer to reset, i.e. TS may indicate that there are no interface characters, the number of historic bytes may be zero and TCK only applies for protocol T=1.

After reception of a valid ATR the state machine switches to the "operational" state. The "operational" state has six sub-states which split up into two paths: a TPDU transfer will run through a command, response and status word sequence, potentially ommiting the response state. A PTS transfer will transfer a given

number of bytes without checking for ACK or status words in order to support the protocol type selection string transfer.

When a card is removed, then the state machine goes to the "removed" state, no matter in which sub-state the driver currently is.

40.3 Requirements

- Support T0 Smart Card, and compatible with the ISO7816-3 spec
- Conforms to the Linux coding standards

40.4 Source Code Structure

The following tabke lists the SIM source files available in the source directory

The mxc_sim_interface.h is located in <ltib_dir>/rpm/BUILD/linux/include/linux/

<ltib_dir>/rpm/BUILD/linux/drivers/char/.

Table 40-2. SIM Driver File List

File	Description
imx_sim.c	SIM driver
mxc_sim_interface.h	Header file defining the programming interfaces and so on

40.5 Linux Menu Configuration Options

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

- CONFIG_IMX_SIM–Build support for the SIM driver. In menuconfig, this option is available under

  Device Drivers > Character devices > IMX SIM support.

  By default, this option is M.

40.6 Programming Interface

This driver implements the functions required by the Linux to interface with the i.MX SIM module. The following listed the programming interface.

- The ioctl interface

  The ioctl interface enables a user space application to access the SIM kernel driver. Amongst others, there are functions to check card presense and for data transfer.

  — SIM_IOCTL_GET_PRESENSE

  Check if the card is present and operational

  — SIM_IOCTL_GET_ATR

Get the received ATR string.

— SIM_IOCTL_GET_PARAM_ATR

Get communication parameters determined from the ATR. If you like to apply the communication parameters, you need to use SIM_IOCTL_SET_PARAM.

— SIM_IOCTL_GET_PARAM

Get currently set communication parameters. The default communication parameters are FI=372, DI=1, PI1=5V, II=50mA and WWT=10. Please note that PI1, II and WWT are currently not supported.

— SIM_IOCTL_SET_PARAM

Set desired set communication parameters. Please note that PI1, II and WWT are currently not supported.

— SIM_IOCTL_XFER

Transfer a TPDU or PTS. A TPDU needs to be at least five bytes ins size. A PTS needs to be at least one byte in size.

— SIM_IOCTL_POWER_ON

Run the power on sequence.

— SIM_IOCTL_POWER_OFF

Run the power off sequence.

— SIM_IOCTL_WARM_RESET

Run the warm reset sequence.

— SIM_IOCTL_COLD_RESET

Run the cold reset sequence.

— SIM_IOCTL_CARD_LOCK

Run the card lock sequence. The current implementation will indicate a card lock by LED no matter if a card is present.

— SIM_IOCTL_CARD_EJECT

Run the card eject sequence. The current implementation will indicate a card eject by LED no matter if a card is present.

Chapter 41
OProfile

OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. OProfile is released under the GNU GPL. It consists of a kernel driver, a daemon for collecting sample data, and several post-profiling tools for turning data into information.

41.1 Overview

OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

41.2 Features

The features of the OProfile are as follows:

- Unobtrusive—No special recompilations or wrapper libraries are necessary. Even debug symbols (-g option to `gcc`) are not necessary unless users want to produce annotated source. No kernel patch is needed; just insert the module.
- System-wide profiling—All code running on the system is profiled, enabling analysis of system performance.
- Performance counter support—Enables collection of various low-level data and association for particular sections of code.
- Call-graph support—With an 2.6 kernel, OProfile can provide gprof-style call-graph profiling data.
- Low overhead—OProfile has a typical overhead of 1–8% depending on the sampling frequency and workload.
- Post-profile analysis—Profile data can be produced on the function-level or instruction-level detail. Source trees, annotated with profile information, can be created. A hit list of applications and functions that utilize the most CPU time across the whole system can be produced.
- System support—Works with almost any 2.2, 2.4 and 2.6 kernels, and works on Cortex-A8 based platforms.

41.3 Hardware Operation

OProfile is a statistical continuous profiler. In other words, profiles are generated by regularly sampling the current registers on each CPU (from an interrupt handler, the saved PC value at the time of interrupt is stored), and converting that runtime PC value into something meaningful to the programmer.

OProfile achieves this by taking the stream of sampled PC values, along with the detail of which task was running at the time of the interrupt, and converting the values into a file offset against a particular binary

file. Each PC value is thus converted into a tuple (group or set) of binary-image offset. The userspace tools can use this data to reconstruct where the code came from, including the particular assembly instructions, symbol, and source line (through the binary debug information if present).

Regularly sampling the PC value like this approximates what actually was executed and how often and more often than not, this statistical approximation is good enough to reflect reality. In common operation, the time between each sample interrupt is regulated by a fixed number of clock cycles. This implies that the results reflect where the CPU is spending the most time. This is a very useful information source for performance analysis.

The ARM CPU provides hardware performance counters capable of measuring these events at the hardware level. Typically, these counters increment once per each event and generate an interrupt on reaching some pre-defined number of events. OProfile can use these interrupts to generate samples and the profile results are a statistical approximation of which code caused how many instances of the given event.

41.4 Software Operation

41.4.1 Architecture Specific Components

If OProfile supports the hardware performance counters available on a particular architecture. Code for managing the details of setting up and managing these counters can be located in the kernel source tree in the relevant `<ltib_dir>/rpm/BUILD/linux/arch/arm/oprofile` directory. The architecture-specific implementation operates through filling in the `oprofile_operations` structure at initialization. This provides a set of operations, such as setup(), start(), stop(), and so on, that manage the hardware-specific details the performance counter registers.

The other important facility available to the architecture code is `oprofile_add_sample`(). This is where a particular sample taken at interrupt time is fed into the generic OProfile driver code.

41.4.2 oprofilefs Pseudo Filesystem

OProfile implements a pseudo-filesystem known as oprofilefs, which is mounted from userspace at `/dev/oprofile`. This consists of small files for reporting and receiving configuration from userspace, as well as the actual character device that the OProfile userspace receives samples from. At setup() time, the architecture-specific code may add further configuration files related to the details of the performance counters. The filesystem also contains a `stats` directory with a number of useful counters for various OProfile events.

41.4.3 Generic Kernel Driver

The generic kernel driver resides in `<ltib_dir>/rpm/BUILD/linux/drivers/oprofile/`, and forms the core of how OProfile operates in the kernel. The generic kernel driver takes samples delivered from the architecture-specific code (through `oprofile_add_sample`()), and buffers this data (in a transformed configuration) until releasing the data to the userspace daemon through the `/dev/oprofile/buffer` character device.

41.4.4 OProfile Daemon

The OProfile userspace daemon takes the raw data provided by the kernel and writes it to the disk. It takes the single data stream from the kernel and logs sample data against a number of sample files (available in `/var/lib/oprofile/samples/current/`). For the benefit of the separate functionality, the names and paths of these sample files are changed to reflect where the samples were from. This can include thread IDs, the binary file path, the event type used, and more.

After this final step from interrupt to disk file, the data is now persistent (that is, changes in the running of the system do not invalidate stored data). This enables the post-profiling tools to run on this data at any time (assuming the original binary files are still available and unchanged).

41.4.5 Post Profiling Tools

The collected data must be presented to the user in a useful form. This is the job of the post-profiling tools. In general, they collate a subset of the available sample files, load and process each one correlated against the relevant binary file, and produce user readable information.

41.5 Requirements

The requirements of OProfile are as follows:

- Add Oprofile support with Cortex-A8 Event Monitor

41.6 Source Code Structure

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

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

Table 41-1. OProfile Source Files

File	Description
op_arm_model.h	Header File with the register and bit definitions
common.c	Source file with the implementation required for all platforms
op_model_v7.c	Source file for ARM V7 (Cortex A8) Event Monitor Driver
op_model_v7.h	Header file for ARM V7 (Cortex A8) Event Monitor Driver

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

41.7 Menu Configuration Options

The following Linux kernel configurations are provided for this module. To get to the Oprofile configuration, use the command `./ltib -c` from the <`ltib dir`>. On the screen, first go to Package list and select oprofile. Then return to the first screen and, select **Configure Kernel**, then exit, and a new screen appears.

- CONFIG_OPROFILE—configuration option for the oprofile driver. In the menuconfig this option is available under

  General Setup > Profiling support (EXPERIMENTAL) > OProfile system profiling (EXPERIMENTAL)

41.8 Programming Interface

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

41.9 Interrupt Requirements

The number of interrupts generated with respect to the OProfile driver are numerous. The latency requirements are not needed. The rate at which interrupts are generated depends on the event.

41.10 Example Software Configuration

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

1. Follow the instruction from Section 41.7, "Menu Configuration Options," to enable Oprofile in the kernel space
2. Download Oprofile userspace tool `oprofile-0.9.5.tar.gz` (which supports ARMV7) from http://oprofile.sourceforge.net/download/
3. Untar `oprofile-0.9.5.tar.gz`

   ```
   tar -xvzf oprofile-0.9.5.tar.gz
   cd oprofile-0.9.5
   ```
4. Build Oprofile 0.9.5. The code below shows how to build and run OProfile with Ubuntu rootfs. Install any necessary Debian packages if any issues are reported:

   ```
   #install debian packages
   apt-get install libpopt-dev
   apt-get install binutils-dev
   # Configure and then build
   sh ./configure --with-kernel-support
   make
   #output to rootfs
   make install DESTDIR=/
   ```
5. Copy vmlinux to `/boot` directory and run Oprofile

   ```
   root@ubuntu:/boot# opcontrol --separate=kernel --vmlinux=/boot/vmlinux
   root@ubuntu:/boot# opcontrol --reset
   Signalling daemon... done
   root@ubuntu:/boot# opcontrol --setup --event=CPU_CYCLES:100000
   ```

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

Chapter 42
Frequently Asked Questions

42.1 Downloading a File

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

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

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

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

42.2 Creating a JFFS2 Mount Point

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

1. Generate the JFFS2 file system under the host:

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

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

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

   NOTE

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

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

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

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

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

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

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

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

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

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

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

42.3 NFS Mounting Root File System

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

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

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

```
service nfs start
```

   Install NFS RPM if not already installed.

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

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

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

NOTE

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

42.4 Error: NAND MTD Driver Flash Erase Failure

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

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

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

42.6 How to Use the Memory Access Tool

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

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

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

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

42.7 How to Make Software Workable when JTAG is Attached

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