i.MX25 PDK Linux

Reference Manual

Part Number: 926-77210
Rev.2009.12
01/2010

How to Reach Us:

Home Page:
www.freescale.com

E-mail:
support@freescale.com

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

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

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

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

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

Contents

Chapter 3
Machine Specific Layer (MSL)

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.MX25 PDK Linux Quick Start Guide*
- *BSP API Document (BSP Doxygen Code Documentation)*
- *i.MX25 PDK Linux User's Guide*
- *i.MX25 PDK Hardware User's Guide*
- *i.MX25 Multimedia Applications Processor Reference Manual*
- [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.MX25 Applications Processor

Because of an order from the United States International Trade Commission, BGA-packaged product lines and part numbers indicated here currently are not available from Freescale for import or sale in the United States prior to September 2010: i.MX25,

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.MX family of integrated circuits (ICs) and their associated platforms (3-Stack board). It provides the software necessary to interface the standard open-source Linux kernel to the i.MX hardware. The goal 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 2.6.31 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 ARM9 interrupt controller. • Timer (GPT): The General Purpose Timer (GPT) is set up to generate an interrupt as programmed to provide OS ticks. Linux facilitates timer use through various functions for timing delays, measurement, events, alarms, high resolution timer features, and so on. Linux defines the MSL timer API required for the OS-tick timer and does not expose it beyond the kernel tick implementation. • GPIO/EDIO/IOMUX: The GPIO and EDIO components in the MSL provide an abstraction layer between the various drivers and the configuration and utilization of the system, including GPIO, IOMUX, and external board I/O. The IO software module is board-specific, and resides in the MSL layer as a self-contained set of files. I/O configuration changes are centralized in the GPIO module so that changes are not required in the various drivers. • SPBA: The Shared Peripheral Bus Arbiter (SPBA) provides an arbitration mechanism among multiple masters to allow access to the shared peripherals. The SPBA implementation under MSL defines the API to allow different masters to take or release ownership of a shared peripheral.	Chapter 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.MX25
Power Management IC (PMIC) Drivers			
PMIC Protocol	The PMIC protocol device driver provides low-level read/write access to PMIC hardware control registers.	Chapter 5, “PMIC (MC34704) Protocol Driver”	i.MX25

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

Feature	Description	Chapter Source	Applicable Platform
PMIC Regulator	The MC34704 regulator driver provides the low-level control of the power supply regulators, selection of voltage levels, and enabling/disabling of regulators.	Chapter 6, “PMIC (MC34704) Regulator Driver”	i.MX25
Power Management Drivers			
Low-level PM Drivers	The low-level power management driver is responsible for implementing hardware-specific operations to meet power requirements and also to conserve power on the development platforms. Driver implementations are often different for different platforms. It is used by the DPM layer.	Chapter 7, “i.MX25 Low-level Power Management (PM) Driver”	i.MX25
CPU Frequency Scaling	The CPU frequency scaling device driver allows the clock speed of the CPUs to be changed on the fly.	Chapter 8, “CPU Frequency Scaling (CPUFREQ) Driver”	i.MX25
Multimedia Drivers			
LCDC	The i.MX liquid crystal display controller (LCDC) provides display data for external gray-scale or color LCD panels. The LCDC is capable of supporting black-and-white, gray-scale, passive-matrix color (passive color or CSTN), and active-matrix color (active color or TFT) LCD panels.	Chapter 9, “Liquid Crystal Display Controller (LCDC) Driver”	i.MX25
OmniVision Camera (OV2640)	The OV2640 Camera driver is designed under Linux V4L2 architecture. It implements V4L2 capture interface.	Chapter 10, “OmniVision Camera (OV2640) Driver”	i.MX25
CSI	The CSI and camera drivers provide the interfaces to support image capture and video output	Chapter 11, “MXC Camera Sensor Interface (CSI) Driver	i.MX25
Sound Drivers			
ALSA Sound	The Advanced Linux Sound Architecture (ALSA) is a sound driver that provides ALSA and OSS compatible applications with the means to perform audio playback and recording functions using the audio components provided by Freescale’s PMIC chips. ALSA has a user-space component called ALSAlib that can extend the features of audio hardware by emulating the same in software (user space), such as resampling, software mixing, snooping, and so on. The ASoC Sound driver supports stereo codec playback and capture through SSI.	Chapter 12, “Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver”	i.MX25

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

Feature	Description	Chapter Source	Applicable Platform
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 13, "NAND Flash Memory Technology Device (MTD) Driver"	i.MX25
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 14, "Low-Level Keypad Driver"	i.MX25
Touch Screen and ADC	A touch screen and associated Analog to Digital Converter (ADC) drivers add measurement functions to the touch screen.	Chapter 15, "Touch Screen and ADC Drivers"	i.MX25
Networking Drivers			
LAN9217 Ethernet	The SMSC LAN9217 Ethernet driver interfaces SMSC LAN9217-specific functions with the standard Linux kernel network module.	Chapter 16, "SMSC LAN9217 Ethernet Driver"	i.MX25
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 17, "Fast Ethernet Controller (FEC) Driver"	i.MX25
DryIce	The DryIce driver controls the low-level encryption key management elements of the Dry Ice. The supported features include key establishment and selection as well as tamper detection.	Chapter 18, "DryIce Driver"	i.MX25

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

Feature	Description	Chapter Source	Applicable Platform
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 20, “Inter-IC (I2C) Driver”	i.MX25
CSPI	The low-level Configurable Serial Peripheral Interface (CSPI) driver interfaces a custom, kernel-space API to both CSPI modules. It supports the following features: • Interrupt-driven transmit/receive of SPI frames • Multi-client management • Priority management between clients • SPI device configuration per client	Chapter 21, “Configurable Serial Peripheral Interface (CSPI) Driver”	i.MX25
MMC/SD/SDIO - eSDHC	The MMC/SD/SDIO Host driver implements the standard Linux driver interface to eSDHC.	Chapter 22, “MMC/SD/SDIO Host Driver”	i.MX25
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 23, “Universal Asynchronous Receiver/Transmitter (UART) Driver”	i.MX25
General Drivers			
USB	The USB driver implements a standard Linux driver interface to the ARC USB-OTG controller.	Chapter 24, “ARC USB Driver”	i.MX25
FlexCAN	The FlexCAN driver is designed as a network device driver. It provides the interfaces to send and receive CAN messages. The CAN protocol was primarily designed to be used as a vehicle serial data bus, meeting the specific requirements of this field: real-time processing, reliable operation in the EMI environment of a vehicle, cost-effectiveness and required bandwidth.	Chapter 25, “FlexCAN Driver”	i.MX25
RTC via DryIce	This secure real time clock (RTC) module is part of the Dry Ice block.	Chapter 26, “Real Time Clock (RTC) (DryIce) Driver”	i.MX25

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 28, “Watchdog (WDOG) Driver”	i.MX25
SIM	The SIM driver implements a Linux driver interface to the Subscriber Identification Module (SIM).		i.MX25
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.MX25
uBoot	uBoot is an open source boot loader.	See uBoot User guide	i.MX25
GUI			

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.

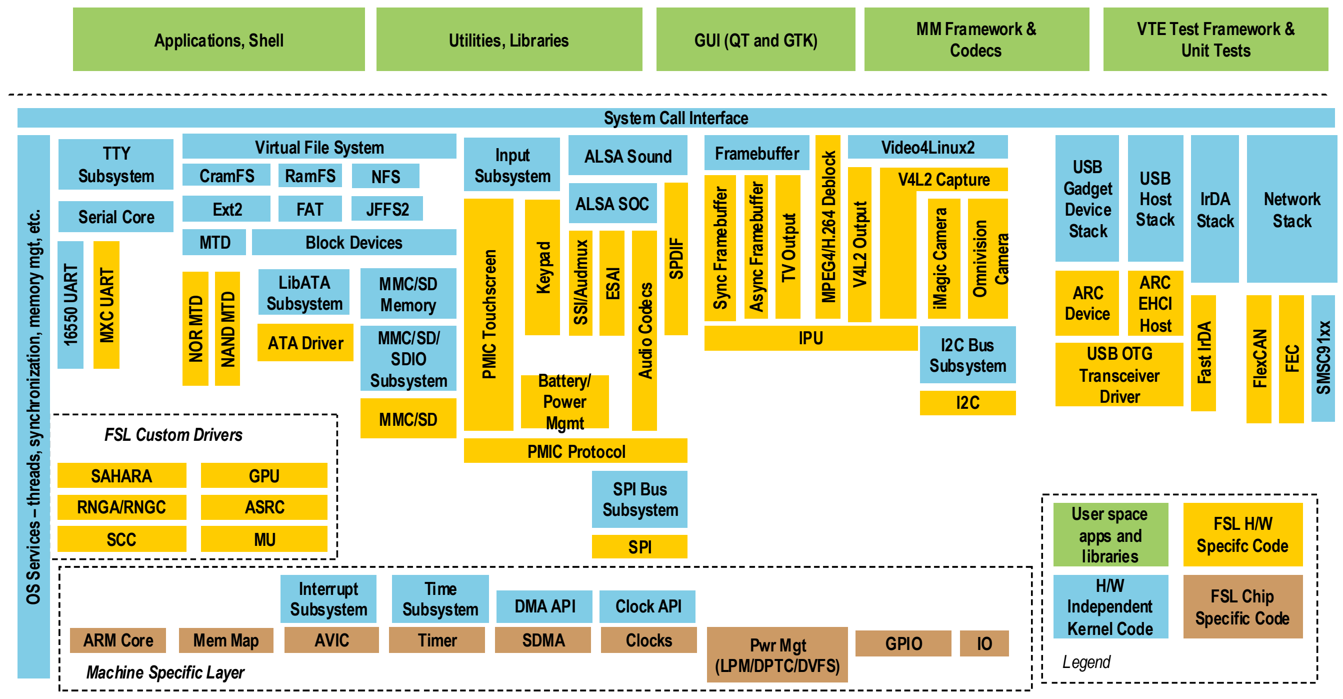

Figure 2-1. BSP Block Diagram

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, IPC)
- Power management
- File systems (VFS, cramfs, ext2, ramfs, NFS, devfs, JFFS2, FAT, UBIFS)
- 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
- ARM9
- 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.MX25 3-Stack	<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx25

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 the ARM9 Interrupt Controller (AITC).

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

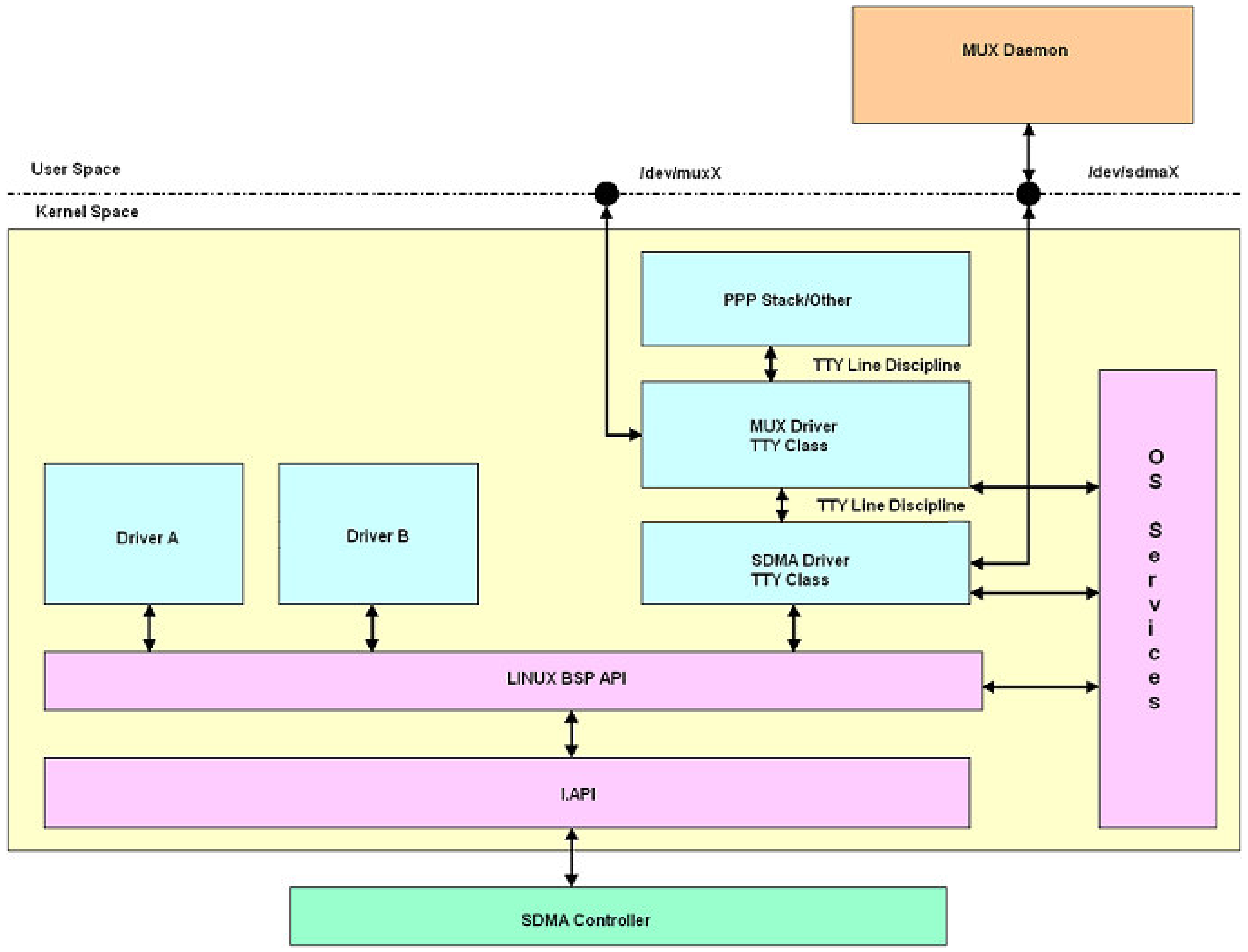

Figure 2-2. SDMA Block Diagram

The TTY driver is only used for IPC and is described in Section 2.3.10.4, “Configurable Serial Peripheral Interface (CSPI) Driver”.

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

The TTY driver is only used for IPC and is described in Chapter 4, “Smart Direct Memory Access (SDMA) API.”

2.3.5 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.

2.3.6 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.

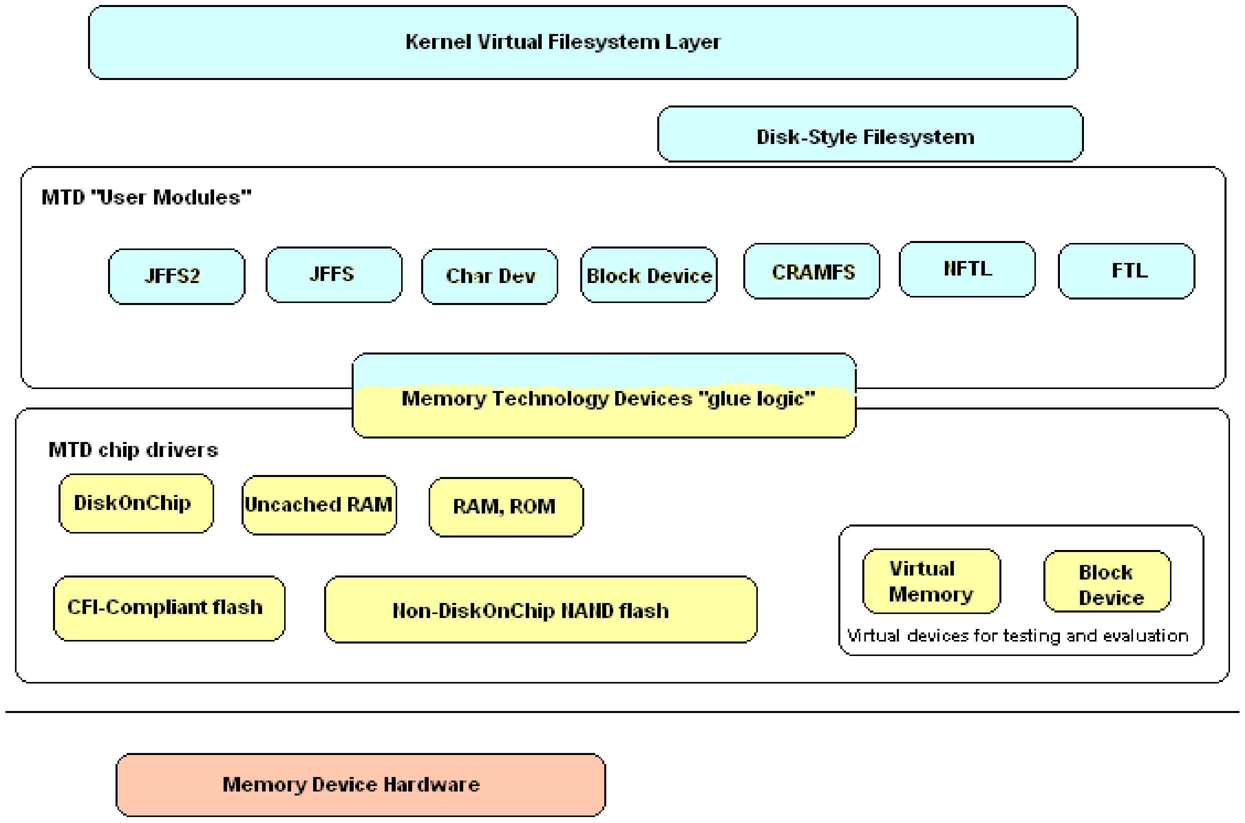

Figure 2-3. MTD Architecture

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.

2.3.7 Networking Drivers

The networking drivers are described in the next sections.

2.3.7.1 SMSC LAN9217 Ethernet Driver

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

The SMSC LAN9217 Ethernet Driver has the following features:

- 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.

2.3.7.2 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.

2.3.8 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.

2.3.8.1 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.

2.3.8.2 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.

2.3.8.3 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.

2.3.9 Security Drivers

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

2.3.9.1 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.

2.3.10 Power managementGeneral Drivers

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

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

2.3.10.1 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

2.3.10.2 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

2.3.10.3 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.

2.3.10.4 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.

2.3.10.5 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

2.3.10.5.1 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.

2.3.10.5.2 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.

2.3.10.5.3 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.

2.3.10.5.4 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.

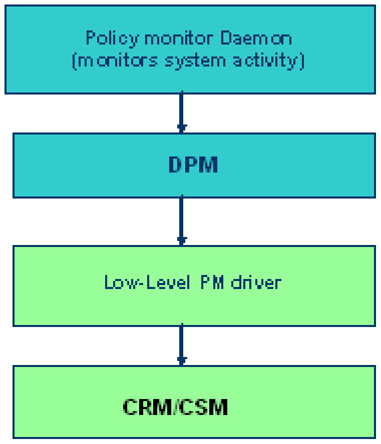

Figure 2-4. DPM High Level Design

Figure 2-5 shows the DPM architecture block diagram.

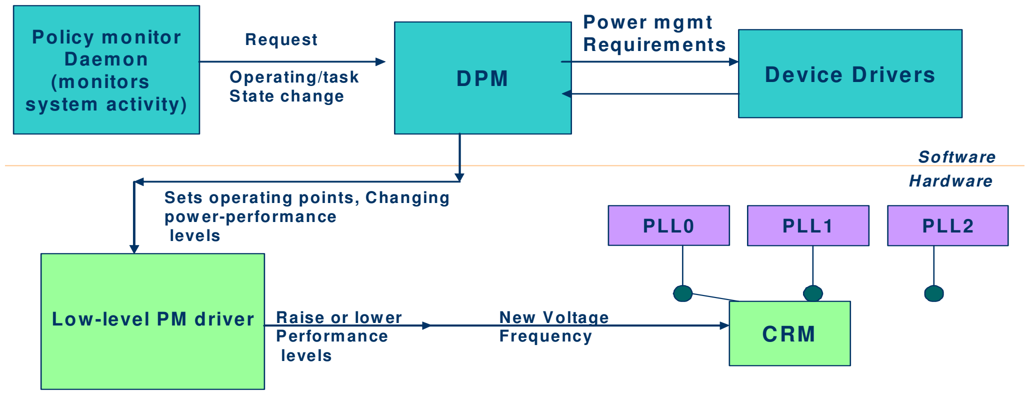
Policy monitor Daemon (monitors system activity)
Request
Operating/task State change
DPM
Power mgmt Requirements
Device Drivers
Software
Hardware
Sets operating points, Changing power-performance levels
PLL0
PLL1
PLL2
Low-level PM driver
Raise or lower Performance levels
New Voltage Frequency
CRM

Figure 2-5. DPM Architecture Block Diagram

2.3.10.6 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.

2.4 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/.

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-mx25 for MX25 platform
```

The header files are implemented under the following directory:

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

The MSL layer contains not only the modules common to all the boards using the same processor, such as the interrupts and timer, but it also contains modules specific to each board, such as the memory map. The following sections describe the basic hardware and software operation and the software interfaces for MSL modules. First, the common modules, such as Interrupts and Timer are discussed. Next, the board-specific modules, such as Memory Map and general purpose input/output (GPIO) (including IOMUX 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`):

```
          irq.c (If CONFIG_MXC_TZIC is not 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
irq.c	Actual interrupt functions

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-<xxx>/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-<platform>
```

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

Table 3-2. Memory Map Files

File	Description
mx25.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-mx<XX>/
          <XX> indicates different platforms.
```

Table 3-3. IOMUX Files

File	Description
iomux.c	IOMUX function implementation
mx*_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 `gpio_mux.c` file under the relevant MSL directory. The header file to define the pin names is under:

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

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

3.4.6.5 GPIO Programming Interface

Table 3-4. IOMUX Through GPIO Files

File	Description
`mx<xxx>_3stack_gpio.c`	IOMUX function implementation
`mx*_pins.h`	Pin name definitions

All the GPIO muxing functions required for the Linux port are implemented in the `gpio_mux.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
mx*_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/plat-mxc` directories.

Table 3-6. EDIO Files

File	Description
`mx25.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 32^{nd} 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 and the peripherals. It supports the following features:

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

4.2 Hardware Operation

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

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

4.3 Software Operation

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

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

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

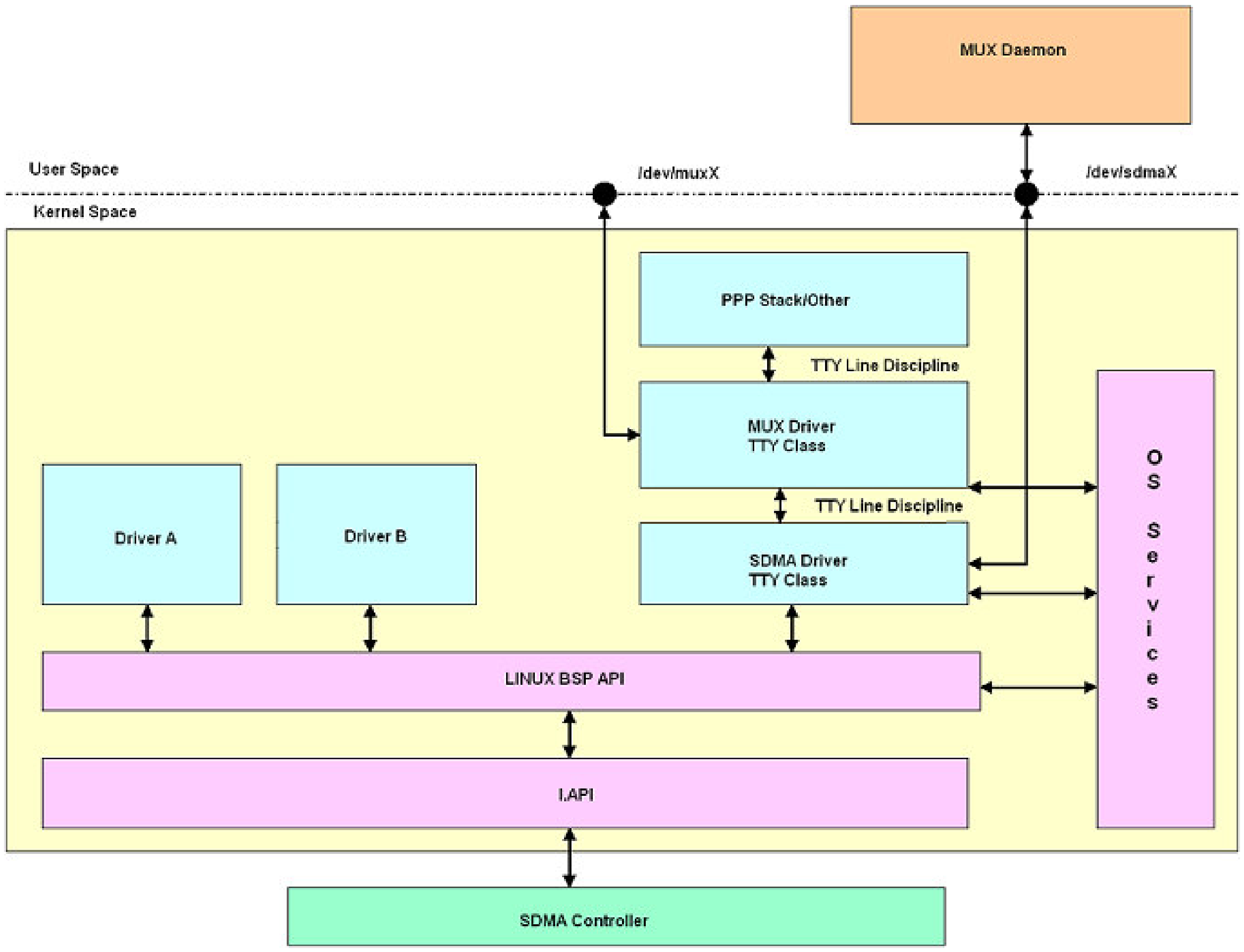

Figure 4-1. SDMA Block Diagram

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

Table 4-1 provides a list of drivers that use SDMA and the number of SDMA physical channels used by each driver. A driver can specify the SDMA channel number that it wishes to use (static channel allocation) or can have the SDMA driver provide a free SDMA channel for the driver to use (dynamic channel allocation). For dynamic channel allocation, the list of SDMA channels is scanned from channel 32 to channel 1. On finding a free channel, that channel is allocated for the requested DMA transfers.

Table 4-1. SDMA Channel Usage

Driver Name	Number of SDMA Channels	SDMA Channel Used
SDMA CMD	1	Static Channel allocation—uses SDMA channels 0
Unified IPC	8	Static Channel allocation—uses SDMA channels 1, 2, 3, 4, 5, 6, 7, 8
SSI	2 per device	Dynamic channel allocation
UART	2 per device	Dynamic channel allocation
Fast IR (FIRI)	2 per device	Dynamic channel allocation
SPDIF	2 per device	Dynamic channel allocation
ESAI	2 per device	Dynamic channel allocation
ATA	2	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-mx`*`<platform>`*`/`.

Table 4-3. SDMA Script Files

File	Description
sdma_script_code.h	SDMA RAM scripts for i.MX25

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 > MX25 Options > Use SDMA API.

  By default, this option is Y.
- CONFIG_SDMA_IRAM—This is the configuration option to support Internal RAM as SDMA buffer or control structures.
- CONFIG_SDMA_IRAM_SIZE: This is the configuration option to set the size of IRAM for SDMA. It must be a multiple of 512bytes.

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
PMIC (MC34704) Protocol Driver

This chapter describes the power management integrated circuit (PMIC) protocol device driver for Linux. The PMIC driver provides the low-level read/write access to the PMIC hardware control registers. The PMIC protocol driver handles all low-level communications between many other Linux device drivers and the PMIC hardware. The PMIC function of i.MX25 3-Stack is implemented by a Freescale MC34704 power IC.The PMIC protocol driver uses the I^{2}C1 bus to communicate with the PMIC chip.

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

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

5.1 Key PMIC Features and Capabilities

The MC34704 PMIC protocol provides hardware support for the following subset of i.MX PMIC functions:

- Power supply control and power management support
- Event notification of regulator faults through hardware polling

5.1.1 PMIC Register Access and Arbitration

The main purpose of the PMIC protocol driver is to provide the necessary read/write access to the PMIC control registers using the I^{2}C bus interfaces to support all of the higher-level PMIC client drivers.

- Access–access to the control registers of the PMIC is implemented through the I^{2}C1 bus interface. The MC34704 acts as a I^{2}C client. The lower part of the protocol driver registers chip as an I^{2}C client and provides APIs for accessing the actual registers of the chip. The upper part of the protocol driver implements a universal pseudo register space and maps these registers to the actual registers of MC34704. Thus the protocol driver as a whole can provide a set of common APIs for accessing the PMIC module.
- Arbitration–the arbitration of accessing PMIC registers is implemented inside the I^{2}C host driver.

5.1.2 Event Notification

The MC34704 does not provide an interrupt signal. The internal regulator fault conditions must be polled by software. When event notification is requested, the driver starts a kernel polling task to periodically sample the fault status register of the PMIC. As a result, there is a delay (up to 100 ms) in the triggering of event handling.

Table 5-1 lists all events that the PMIC protocol driver supports. Regulator faults can be over current, short circuit, over/under voltage, or thermal shutdown.

Table 5-1. Events Supported by Protocol Driver

Event	Description
Regulator 1 Fault	Fault indicated in Regulator 1
Regulator 2 Fault	Fault indicated in Regulator 2
Regulator 3 Fault	Fault indicated in Regulator 3
Regulator 4 Fault	Fault indicated in Regulator 4
Regulator 5 Fault	Fault indicated in Regulator 5
Regulator 6 Fault (unused on i.MX25 3-stack)	Fault indicated in Regulator 6
Regulator 7 Fault (unused on i.MX25 3-stack)	Fault indicated in Regulator 7
Regulator 8 Fault (unused on i.MX25 3-stack)	Fault indicated in Regulator 8

5.2 Driver Requirements

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

- Control Services
- Event Notification Services

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

5.3 Driver Software Operation

The PMIC protocol driver controls the PMIC by reading and writing the PMIC hardware control registers. Both read and write access to the PMIC hardware control registers is done through the I^2C driver.

5.4 Driver Implementation Details

This section describes implementation-specific details associated with the PMIC protocol driver. The device driver source files should also be consulted to fully understand the implementation of the PMIC protocol driver. Chapter 20, “Inter-IC (I2C) Driver” should also be consulted if required.

5.4.1 Driver Initialization

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

- Registers MC34704 as an I^{2}C client
- Creates either a `/dev/pmic` character device entry and registers the new device with the kernel
- Initializes all driver-specific global variables

5.4.2 Driver Unloading

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

- Remove the `/dev/pmic` device entry and tell the kernel to deregister this device
- Unregister MC34704 I^{2}C client

5.4.3 Register Access

The PMIC protocol driver exports APIs that allow other device drivers to read and write the PMIC control registers. The PMIC control registers are accessed using the I^{2}C interface. Externally, the PMIC protocol driver simply provides APIs to read and write to and from the PMIC control registers.

Some registers of the MC34704 are write-only. To provide readability to the upper layer, a register cache of MC34704 is employed.

5.5 Driver Source Code Structure

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

Table 5-2 shows the device driver source files.

Table 5-2. PMIC Protocol Driver Files

File	Description
`pmic_external.c`	This file contains client API implementation, define SPI interface
`pmic_event.c`	This file manage all event of PMIC component
`pmic-dev.c`	This provides `/dev` interface to the user-space programs
`pmic.h`	Declaration of all the functions whose implementation differs from PMIC chip to PMIC chip
`mfd/mc34704/core.h`	Define Regulator macros and values for the MC34704 PMIC
`mc34704.c`	Low level driver for the MC34704 PMIC

In addition to the driver-specific source files, there also exists a `Kconfig` file that is used to define the device driver build configuration (see Section 5.6, "Linux Menu Configuration Options") and a `Makefile` that is used during the Linux kernel image build process.

5.6 Linux Menu Configuration Options

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

The following Linux kernel configuration 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:

- Device Drivers > MXC Support Drivers > MXC PMIC Support > MC34704 PMIC

Chapter 6
PMIC (MC34704) Regulator Driver

The MC34704 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 (see Chapter 5, “PMIC (MC34704) Protocol Driver”) to access the PMIC hardware control registers.

6.1 PMIC Features

The MC34704 integrates eight high-performance, high-efficiency DC-DC switching regulators. Other functions include a reset driver and a 2-wire I^2C serial interface.

An I^2C-compatible, 2-wire serial interface controls a variety of MC34704 functions:

- Dynamic voltage scaling setting for each regulator
- On/Off for three regulator groups
- Fault status for each regulator (over/under voltage, over current, short-circuit, and high temperature shutdown)

6.2 Driver Requirements

The MC34704 PMIC regulator driver is based on the PMIC protocol driver and regulator core driver. It provides services for regulator control of the PMIC component.

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

6.3 Driver Software Operation

The MC34704 regulator client driver performs operations by reconfiguring the PMIC hardware control registers. This is done by calling protocol driver APIs with the required register settings.

6.4 Regulator APIs

The regulator power architecture is designed to provide a generic interface to voltage and current regulators within the Linux 2.6 kernel. It is intended to provide voltage and current control to client or consumer drivers and also to provide status information to user space applications through a sysfs interface.

The intention is to allow systems to dynamically control regulator output in order 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 operation can be done but the following unified API calls:

1. `regulator_get` – lookup and obtain a reference to a regulator
   `struct regulator *regulator_get(struct device *dev, const char *id);`
2. `regulator_put` – free the regulator source
   `void regulator_put(struct regulator *regulator, struct device *dev);`
3. `regulator_enable` – enable regulator output
   `int regulator_enable(struct regulator *regulator);`
4. `regulator_disable` – disable regulator output
   `int regulator_disable(struct regulator *regulator);`
5. `regulator_is_enabled` – is the regulator output enabled
   `int regulator_is_enabled(struct regulator *regulator);`
6. `regulator_set_voltage` – sets regulator output voltage
   `int regulator_set_voltage(struct regulator *regulator, int uV);`
7. `regulator_get_voltage` – gets regulator output voltage
   `int regulator_get_voltage(struct regulator *regulator);`

Find more APIs and details in the regulator core source code in `drivers/regulator/core.c`.

6.5 Driver Architecture

Figure 6-1 shows the basic architecture of the MC34704 regulator driver.

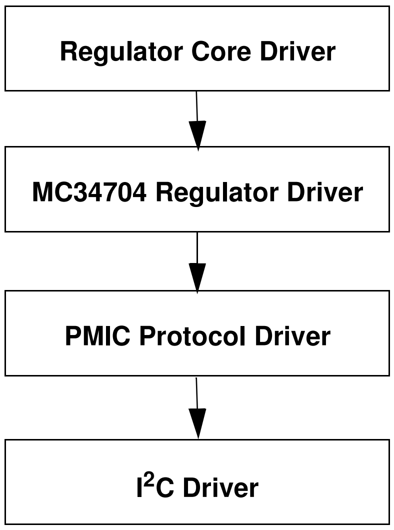

Figure 6-1. MC34704 Regulator Driver Architecture

6.6 Driver Implementation Details

The access to the MC34704 regulator is provided through the APIs of the regulator core driver. The MC34704 regulator driver provides the following regulator controls:

- MC34704 REG1 through REG5 supply voltage to BKLT, CPU, CORE, DDR, and PERS power supply rails

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

6.7 Driver Source Code Structure

Table 6-1 shows the MC34704 regulator driver files that are available in the directory, `<ltib_dir>/rpm/BUILD/linux/drivers/regulator`.

Table 6-1. MC34704 Power Management Driver Files

File	Description
reg-mc34704.c	Implementation of the MC34704 regulator client driver

6.8 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:

- Device Drivers > Voltage and Current Regulator Support> MC34704 Regulator Support

Chapter 7
i.MX25 Low-level Power Management (PM) Driver

This section describes the low-level PM driver which controls the low-power modes.

7.1 Hardware Operation

The low-power modes on the i.MX25 device are controlled by software using the clock controller module (CCM). The CCM:

- Controls the system frequency
- Distributes clocks to various parts of the chip
- Controls the reset mechanism of the chip
- Provides advanced low-power management

7.1.1 Lower Power Mode

The i.MX25 supports a versatile list of power modes, as shown in Table 7-1, including power and clock domains status and applied power techniques.

Table 7-1. Low Power Modes

Mode	Core	ARM MAX	Modules	MPLL	UPLL	Osc24M	Osc32K	Perclk
RUN	Active	Active	Active, Idle or Disable	On	On/off	On	On	Some On
WAIT	Disable	Active	Active, Idle or Disable	On	On/off	On	On	Some On
DOZE	Disable	Disable	Active, Idle or Disable	On	On/off	On	On	Some On
STOP	Disable	Disable	Disable	Off	Off	On/Off	On	Off

State-retention mode is when the logic core supply is lowered from 1.1 V to 1.0 V in Stop mode. Static stop mode is when the Osc24M is powered off and all clocks are off including CKIL.

NOTE

The i.MX25 3-Stack does not support VSTBY. VSTBY signal is connected to the PMIC and the PMIC does nothing when VSTBY is asserted. Therefore, state-retention mode is disabled in the CCM (VSTBY bit is set to ‘0’).

7.2 Software Operations

For Doze and Stop modes, software should disable interrupts before executing a wait-for-interrupt (WFI) instruction and then re-enable interrupts afterwards.

Use the following steps to enter and exit low power mode:

1. Setup wakeup interrupt before entering lower power mode.
2. Program the LP CTL field in the CCM CCTL register. For Stop mode, configure the VSTBY and OSC24M_DOWN bit according to the hardware design.
3. Call `cpu_do_idle` to execute WFI pending instructions.
4. Generate a wakeup interrupt and exit low power mode.

The i.MX25 PM driver maps the low-power modes to the kernel power management states as listed below:

- Standby – maps to Doze 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 is 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

7.3 Source Code Structure

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

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx25/`

Table 7-2. PM Driver Files

File	Description
pm.c	Supports suspend operation
system.c	Supports lower power modes

7.4 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:

- CONFIG_PM – Build support for power management. In `menuconfig`, this option is available under Power management options > Power Management support.

  By default, this option is Y.
- CONFIG_SUSPEND – Build support for suspend. In `menuconfig`, this option is available under Power management options > Suspend to RAM and standby.

7.5 Programming Interface

he mxc_cpu_lp_set API is provided for low-power modes. This implements all the steps required to put the system into Wait, Doze, or Stop mode.

Chapter 8
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 GP 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.

8.1 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. 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
```

8.2 Source Code Structure

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

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

Table 8-1. CPUFREQ Driver Files

File	Description
cpufreq.c	CPUFREQ functions

8.3 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

8.3.1 Board Configuration Options

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

Chapter 9
Liquid Crystal Display Controller (LCDC) Driver

The LCDC provides display data for external gray-scale or color LCD panels. The LCDC is capable of supporting black-and-white, gray-scale, passive-matrix color (passive color or CSTN), and active-matrix color (active color or TFT) LCD panels. The detailed hardware operation of the LCDC can be found in the *Multimedia Applications Processor Reference Manual*

9.1 LCD Driver Overview

The LCD driver is designed under the Linux frame buffer driver framework. It provides hardware ability to support the frame buffer driver. The frame buffer device provides an abstraction for the graphics hardware. It represents the frame buffer of video hardware and allows application software to access the graphics hardware through a well-defined interface, so the software does not need to know anything about the lower 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 depends on the Virtual Terminal (VT) console for switching from serial to graphics mode.

The device is accessed through special device nodes, usually located in the `/dev/fb*` directory. 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 LCDC 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. Support for new panels can be added by defining new timing values for the structure `fb_videomode`. The panel to be enabled during Linux booting up can be specified by appending video options onto the kernel command line.

9.1.1 Hardware Operation

The frame buffer interacts with the LCDC hardware module. Refer to the LCDC section in the *Multimedia Applications Processor Reference Manual* for more information.

9.1.2 Software Operation

A frame buffer device is a memory device like `/dev/mem` and it has the same features. It can be read from, written to, a location in it can be seeked and the `mmap()` function can be used (the main usage). The difference is that the memory that appears in the file is not the whole memory, but the frame buffer of some video hardware.

`/dev/fb*` also allows several IOCTLs to operate on it, by which information about the hardware can be queried and set. The color map handling operates through IOCTLs as well. `linux/fb.h` contains more information about what IOCTLs exist and which data structures they use. Here is a brief overview:

- Unchangeable information about the hardware such as the name, organization of the screen memory (planes, packed pixels, and so on) and address and length of the screen memory can be requested.
- Variable information about the hardware, such as the visible and virtual geometry, depth, color map format, timing, and so on can be requested and changed. If this information is changed, the driver may round up some values to meet the hardware capabilities (or return EINVAL if the change is not possible).
- Parts of the color map can be retrieved and set. Communication is with 16 bits per color part (red, green, blue, and transparency) to support all existing hardware. The driver makes the necessary computations to apply values to the hardware (round down to less bits, throw away transparency and so on).

The hardware abstraction makes the implementation of application programs easier and more portable. For example, the `Qt/Embedded` server operates completely on `/dev/fb*` and therefore does not need to know, for example, how the color registers of the concrete hardware are organized. Only the screen organization (bitplanes or chunky pixels) must be built into application programs, because they work on the frame buffer image data directly.

The frame buffer driver (`<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc/mx2fb.c`) interacts closely with the generic Linux frame buffer driver (`<ltib_dir>/rpm/BUILD/linux/drivers/video/fbmem.c`).

9.1.3 Graphics Window

The graphics window is supported by the LCDC for viewfinder functions in a color display. The graphics window and background plane can be alpha blended. In addition, one of the pixel colors can be chosen for color keying in which the selected pixel color is made totally transparent. Memory used by the graphics window can be different from the memory used by the background plane. Thus two frame buffer devices are implemented by this driver, one for the background plane and the other for the graphics window.

Since the graphics window is a special display (for example, the graphics window supports alpha blending and color keying), additional IOCTLs are provided to support these features.

9.1.4 Architecture

The architecture diagram is shown in Figure 9-1.

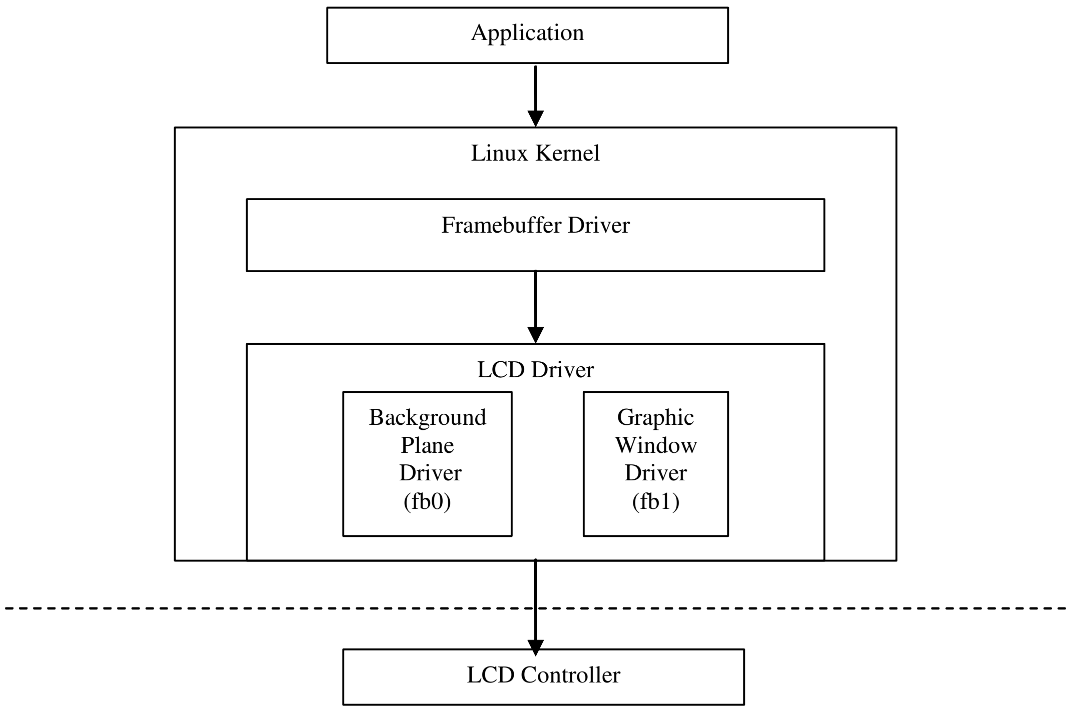

Figure 9-1. LCD Driver Architecture

9.2 Source Code Structure Configuration

Table 9-1 shows the LCD driver source files that are located in the directory `<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc`

Table 9-1. LCD Driver Files

File	Description
`mx2fb.c`	Source file
`mx2fb.h`	Header file

9.3 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_FB_MXC – This is the configuration option for the frame buffer driver. This option is dependent on the CONFIG_FB option. In the `menuconfig`, this option is available under

  Device Drivers > Graphics support > MXC Framebuffer support.

  By default, this option is Y.

- CONFIG_FB_MXC_SYNC_PANEL – This is the configuration option for the synchronous LCD frame buffer device. This option is dependent on CONFIG_FB_MXC option. In the `menuconfig` this option is available under

  Device Drivers > Graphics support > Synchronous Panel Framebuffer.

  By default, this option is Y.

Chapter 10
OmniVision Camera (OV2640) Driver

The OV2640FSL is a small on-board camera sensor and lens module with low power consumption. The camera driver is located under the Linux V4L2 architecture. and it implements the V4L2 capture interfaces. Applications cannot use the camera driver directly. Instead, the applications use the V4L2 capture driver to open and close the camera for preview and image capture, controlling the camera, getting images from camera, and starting the camera preview.

10.1 Hardware Operation

The OV2640FSL uses the serial camera control bus (SCCB) interface to control the sensor operation. It works as an I^2C client, and CSI interface of IPU works as the I^2C master, which uses I^2C bus to control camera operation.

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

Refer to OV2640 and OV2640FSL datasheet to get more information on the sensor. Refer to the *i.MX25 Multimedia Applications Processor Reference Manual* for more information on CSI and IPU (IPU is not supported on all platforms).

10.2 Software Operation

The camera driver implements the V4L2 capture interface and applications use the V4L2 capture interface to operate the camera. The supported operations of V4L2 capture are:

- Preview
- Capture still mode

The supported picture formats are:

- RGB565
- YUV422P
- YUV420

10.3 Source Code Structure

Table 10-1 shows the camera driver source files available in the directory
`<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture.`

Table 10-1. Camera Driver Files

File	Description
ov2640.c	Camera driver implementation

10.4 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:

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

Chapter 11
MXC Camera Sensor Interface (CSI) Driver

The CSI driver enables the i.MX device to directly connect to external CMOS sensors and CCIR656 video sources. The CSI and sensor drivers are implemented in the Video for Linux Two (V4L2) driver framework. It consists of the image capture driver and the video output driver.

11.1 Hardware Operation

The CSI driver configures and operates with the hardware registers for the CSI module. It provides:

- Configurable interface logic to support most commonly available CMOS sensors.
- Full control of 8-bit/pixel, 10-bit/pixel or 16-bit/pixel data format to 32-bit receive FIFO packing.
- 128×32 FIFO to store received image pixel data.
- Receive FIFO overrun protection mechanism.
- Embedded DMA controllers to transfer data from receive FIFO or statistic FIFO through AHB bus.
- Support for double bufferring two frames in the external memory.
- Single interrupt source to interrupt controller from maskable interrupt sources: Start of Frame, End of Frame and so on.
- Configurable master clock frequency output to sensor.

For more information, see the CSI chapter in the *i.MX25 Multimedia Applications Processor Reference Manual*.

11.2 Software Operation

11.2.1 CSI Software Operation

The CSI driver initializes the CSI interface. Applications use the V4L2 interface to operate the CSI interface.

11.2.2 Video for Linux 2 (V4L2) APIs

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

The V4L2 capture device includes two interfaces: the capture interface and the overlay interface. The capture and overlay interface use the CSI embeded DMA controller to implement the function. The V4L2 driver implements the standard V4L2 API for capture and overlay devices. The following is the data flow of capture and overlay.

1. The camera sends the data to the CSI receive FIFO, through the 8-bit/10-bit data port.
2. The embeded DMA controllers transfer data from the receive FIFO to external memory through the AHB bus.
3. The data is save to user space memory or output to the frame buffer directly.

11.2.2.1 V4L2 Capture Device

V4L2 capture support can be selected during kernel configuration. The driver for this device is in the `<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture/csi_v4l2_capture.c` file.

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

- `VIDIOC_QUERYCAP`
- `VIDIOC_G_FMT`
- `VIDIOC_S_FMT`
- `VIDIOC_OVERLAY`
- `VIDIOC_G_FBUF`
- `VIDIOC_S_FBUF`
- `VIDIOC_S_PARM`
- `VIDIOC_G_PARM`
- `VIDIOC_QUERYBUF`
- `VIDIOC_REQBUFS`
- `VIDIOC_DQBUF`
- `VIDIOC_QBUF`
- `VIDIOC_STREAMON`
- `VIDIOC_STREAMOFF`

11.2.2.2 Use of the V4L2 Capture APIs

The following are some sample use cases for the V4L2 capture APIs:

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.

- VIDIOC_STREAMON and VIDIOC_OVERLAY cannot be enabled simultaneously.

11.3 Source Code Structure

Table 11-1 shows the CSI sensor and V4L2 driver source files available in the following directory:

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

Table 11-1. V4L2 and SI Driver Files

File	Description
fsl_csi.c	CSI driver source file
fsl_csi.h	CSI driver header file
csi_v4l2_capture.c	V4L2 capture device driver source file
mxc_v4l2_capture.h	V4L2 capture device driver header file
ov2640.c	Camera driver source file

11.4 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:

- VIDEO_MXC_CSI_CAMERA – Includes support for the CSI Unit and V4L2 capture device. In `menuconfig`, this option is available under:

  Device Drivers > Multimedia devices > Video For Linux > Video Capture Adapters > MXC Camera/V4L2 PRP Features support

  By default, this option is M.
- CONFIG_MXC_CAMERA_OV2640 – Option for the OV2640 sensor driver. In `menuconfig`, this option is available under:

  Device Drivers > Multimedia devices > Video For Linux > Video Capture Adapters > MXC Camera/V4L2 PRP Features support

  By default, this option is M.

11.5 Programming Interface

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

Chapter 12
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.

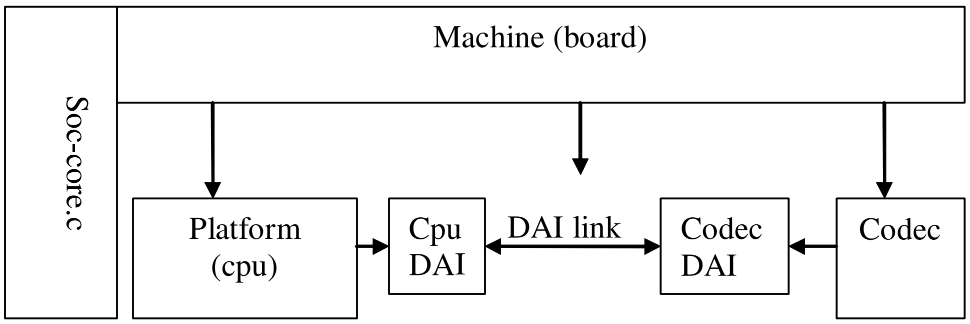

Figure 12-1. ALSA SoC Software Architecture

The ALSA SoC driver has the following components as seen in Figure 12-1:

- 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^{2}S, 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.

12.1 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

The 5.1 codec is only supported on the i.MX35 and i.MX25 platform.

The 4-channel ADC codec is only supported on the i.MX25 platform.

12.1.1 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

12.1.2 5.1 Codec Features

- Supported sample rates for playback are:

  8 KHz, 11.025 KHz, 16 KHz, 22.05 KHz, 32 KHz, 44.1 KHz,
  48 KHz, 64 KHz, 88.2 KHz, 96 KHz, 176.4 KHz, and 192 KHz
- Supported channels for playback: 1-6 channels
- Supported audio formats for playback:
  - SNDRV_PCM_FMTBIT_S16_LE
  - SNDRV_PCM_FMTBIT_S24_LE

12.1.3 4-Channel ADC Codec Features

- Supported sample rates for record are:

  8 kHz, 11.025 kHz, 12 kHz, 16 kHz, 22.05 kHz, 24kHz, 32 kHz, 44.1 kHz, 48 kHz
- Supported channels for record: 1-4 channels
- Supported audio formats are:

  SNDRV_PCM_FMTBIT_S16_LE

12.1.4 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
      card 1: imx3stack_1 [imx-3stack], device 0: wm8580 WM8580 PAIFRX-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
```

12.2 ASoC Driver Source Architecture

As illustrated in Figure 12-2, `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.

The 5.1 codec is connected to the CPU through the ESAI interface. imx-esai registers the CPU DAI driver for the 5.1 ALSA SoC and configures the on-chip ESAI interface. `wm8580.c` is the codec driver that operates on the 5.1 codec directly, as well as on the ESAI configuration on the codec side. The machine layer code is implemented in `imx-3stack-wm8580.c` to register the sound card and setup the link between the CPU and the codec.

The 4-channel ADC is connected to the CPU through the ESAI interface. imx-esai registers the CPU DAI driver for the 4-channel ALSA SoC and configures the on-chip ESAI interface. ak5702.c is the codec driver that operates on the 4-channel ADC directly, as well as on the ESAI configuration on the codec side. The machine layer code is implemented in imx-3stack-ak5702.c to register the sound card and setup the link between the cpu and the codec.

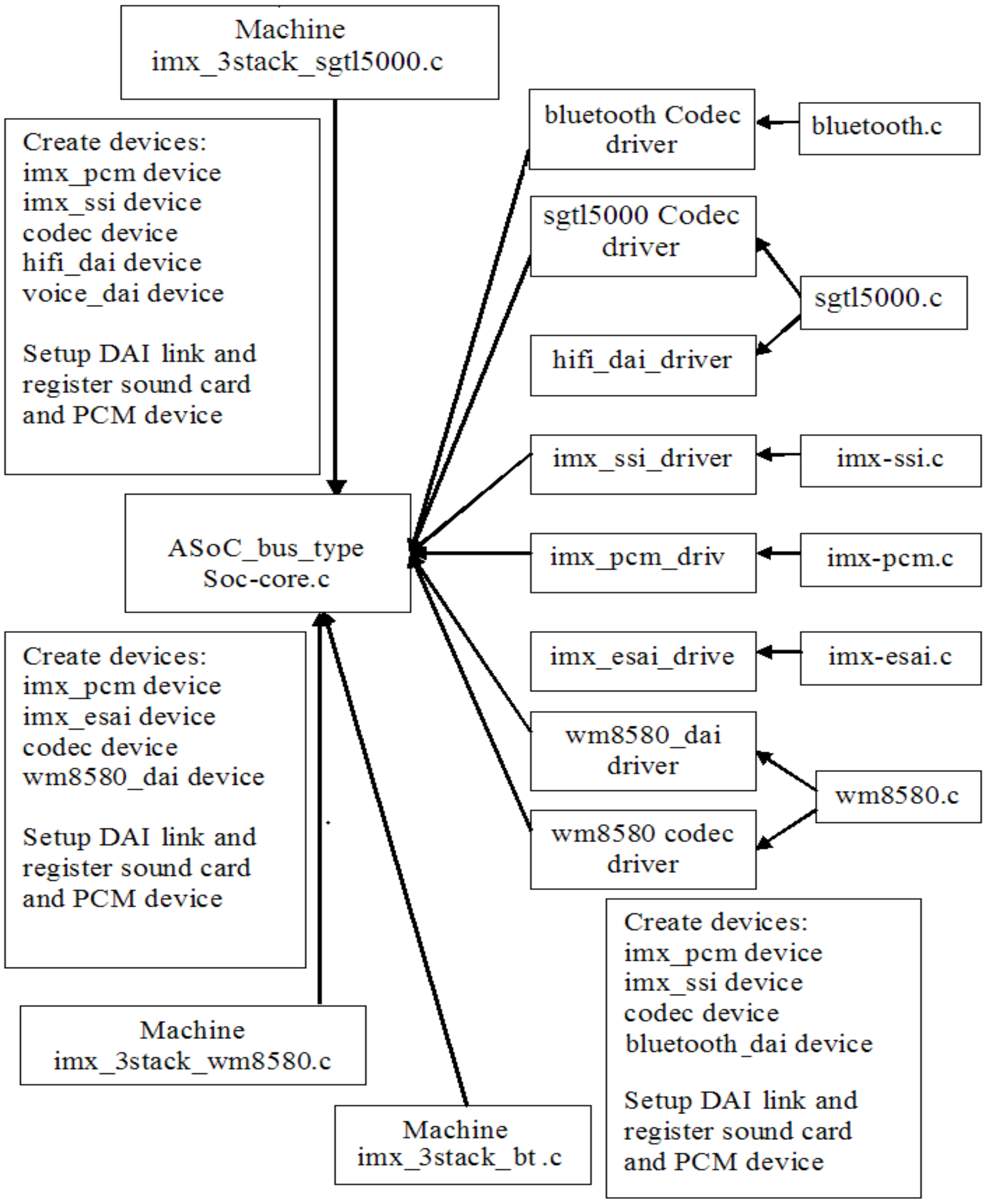

Figure 12-2. ALSA SoC Source File Relationship

Table 12-1 shows the stereo codec SoC driver source files. These files are under the `<ltib_dir>/rpm/BUILD/linux/sound/soc` directory.

Table 12-1. 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

Table 12-1. Stereo Codec SoC Driver Files (continued)

File	Description
imx/imx-pcm.h	Header file for PCM driver and AUDMUX register definitions
imx/imx-ssi.c	Platform DAI link for stereo codec ALSA SoC
imx/imx-ssi.h	Header file for platform DAI link and SSI register definitions
imx/imx-ac97.c	AC97 driver for i.MX chips
codecs/sgtl5000.c	Codec layer for stereo codec ALSA SoC
codecs/sgtl5000.h	Header file for stereo codec driver

Table 12-2 shows the 5.1 codec SoC driver source files. These files are also under the `<ltib_dir>/rpm/BUILD/linux/sound/soc` directory.

Table 12-2. 5.1 Codec SoC Driver Files

File	Description
imx/imx-3stack-wm8580.c	Machine layer for 5.1 ALSA SoC
imx/imx-pcm.c	Platform layer for 5.1 codec ALSA SoC
imx/imx-pcm.h	Header file for pcm driver
imx/imx-esai.c	Platform DAI link for 5.1 codec ALSA SoC
imx/imx-esai.h	Header file for platform DAI link
codecs/wm8580.c	Codec layer for 5.1 codec ALSA SoC
codecs/wm8580.h	Header file for 5.1 codec driver

Table 12-3 shows the 4-channel ADC SoC driver source files. These files are also under the `<ltib_dir>/rpm/BUILD/linux/sound/soc` directory.

Table 12-3. 4 channel ADC codec ASoC Driver Source File

File	Description
imx/imx-3stack-ak5702.c	Machine layer for 4-channel ADC ALSA SoC
imx/imx-pcm.c	Platform layer for 4-channel ADC ALSA SoC
imx/imx-pcm.h	Header file for pcm driver
imx/imx-esai.c	Platform DAI link for 4-channel ADC ALSA SoC
imx/imx-esai.h	Header file for platform DAI link
codecs/ak5702.c	codec layer for 4-channel ADC ALSA SoC
codecs/ak5702.h	Header file for 4-channel ADC driver

12.3 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
- CONFIG_SND_MXC_SOC_IRAM: This config is used to allow audio DMA playback buffers in IRAM. In menuconfig, this option is available under

  Device drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > Locate Audio DMA playback buffers in IRAM
- Device drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio for the Freescale i.MX CPU, SoC Audio support for IMX - WM8580
- Device drivers-> Sound card support-> Advanced Linux Sound Architecture-> ALSA for SoC audio support > SoC Audio for the Freescale i.MX CPU, SoC Audio support for IMX - AK5702

12.4 Hardware Operation

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

12.4.1 Stereo Audio Codec

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

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

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

- Codec DAI and PCM configuration
- Codec control I/O—using I^{2}C
- Mixers and audio controls
- Codec audio operations
- DAC Digital mute control

The SGTL5000 codec is registered as an I^{2}C 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.

12.4.2 5.1 Audio Codec

The 5.1 audio codec is controlled by the SPI interface. The audio data is transferred from the user data buffer to the ESAI FIFO through a DMA channel. The DMA channel is selected according to the audio sample bits. The 5.1 codec works in master mode and the codec provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate. The ESAI supports up to three TX ports, and each port transmits two channels of data in I^2S format. The TX port is enabled or disabled according to the audio channel number.

12.4.3 4-Channel ADC Codec

The 4-channel ADC is controlled by the I^2C interface. The audio data is transferred from the user data buffer to the ESAI fifo through a DMA channel. The DMA channel is selected according to audio sample bits. The 4-channel ADC works in master mode as the codec provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate. The ESAI supports up to 4 receivers. On the i.MX25 3-stack board, two receivers are used, each receives two channels of data in the I2S format. Both receivers are enabled for 4-channel record.

12.5 Software Operation

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

12.5.1 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

12.5.2 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 13
NAND Flash Memory Technology Device (MTD) Driver

13.1 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.

13.1.1 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.

13.1.2 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/.

13.2 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

13.3 Source Code Structure

Table 13-1 shows the source files available for the NAND MTD driver. These files are under the `<ltib_dir>/rpm/BUILD/linux/drivers/mtd/nand` directory.

Table 13-1. NAND MTD Driver Files

File	Description
mxc_nd2.c	Hardware-specific layer for NAND MTD driver for NFC version 2 and above
mxc_nd2.h	Register declaration for NFC version 2 and above

13.4 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_V2 – This is the configuration option for the NAND MTD driver for the i.MX processors having NFC hardware version 2.

13.5 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 14 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.

14.1 Hardware Operation

The KPP supports a keypad matrix with as many as eight rows and eight columns. The i.MX25 3-Stack keypad has 16 keys in a 4×4 array. 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.

14.2 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 14-1. The `mxc_kpp_scan_matrix` function is called for every keypress and release interrupt.

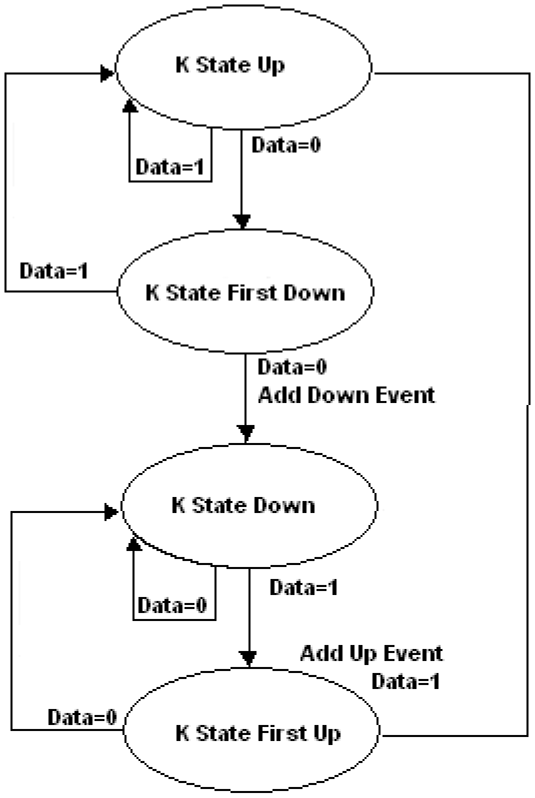

Figure 14-1. Keypad Driver State Machine

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 14-3 for map codes and scan codes.

14.3 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.

14.4 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

14.5 Source Code Structure

Table 14-1 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-mx25
```

Table 14-1. 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
arch/arm/mach-mx25/mx25_3stack.c, , , ,	Contains the platform-specific keymapping keycode array

14.6 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 14-1).
- Debounce delay—The user can configure the debounce delay by changing the variable `KScanRate` defined in `mxc_keyb.c`

14.7 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/`.

14.8 Interrupt Requirements

Table 14-2 lists the keypad interrupt timer requirements.

Table 14-2. 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

14.9 Device-Specific Information

Table 14-3 shows key connections, key scan codes, and key map codes of the keys on the keypad for a specific platform.

Table 14-3. Key Connections for Keypad

Key	Row	Column	Scancode	Linux Key Code	Platform
SW40	0	0	0	KEY_UP	i.MX25
SW36	0	1	1	KEY_DOWN	i.MX25
SW34	0	2	2	KEY_VOLUMEDOWN	i.MX25

Table 14-3. Key Connections for Keypad (continued)

Key	Row	Column	Scancode	Linux Key Code	Platform
SW32	0	3	3	KEY_HOME	i.MX25
SW39	1	0	4	KEY_RIGHT	i.MX25
SW31	1	1	5	KEY_LEFT	i.MX25
SW18	1	2	6	KEY_ENTER	i.MX25
SW17	1	3	7	KEY_VOLUMEUP	i.MX25
SW38	2	0	8	KEY_F6	i.MX25
SW29	2	1	9	KEY_F8	i.MX25
SW14	2	2	10	KEY_F9	i.MX25
SW13	2	3	11	KEY_F10	i.MX25
SW37	3	0	12	KEY_F1	i.MX25
SW30	3	1	13	KEY_F2	i.MX25
SW10	3	2	14	KEY_F3	i.MX25
SW9	3	3	15	KEY_POWER	i.MX25

Figure 14-2 shows the button switch placement on the bottom of the i.MX25 Personality board.

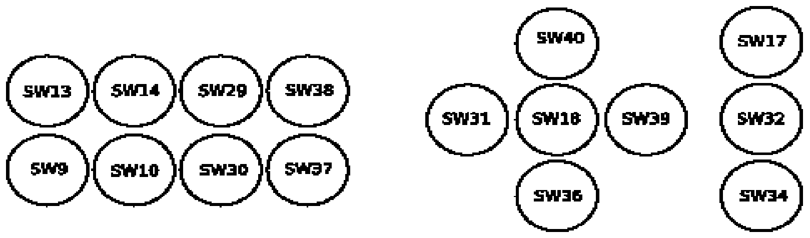

Figure 14-2. Keypad Button Placement

Chapter 15
Touch Screen and ADC Drivers

15.1 Driver Overview

The Touch Screen controller and the associated Analog to Digital Converter (ADC) together provide a resistive touch screen solution for low cost PDAs, cell phones, ePOS devices, and multi-media players. The module implements simultaneous touch screen control and auxiliary ADC operation for temperature, voltage and other measurement functions. It includes the driver switches for controlling the screen and an input multiplexer to allow one of four additional inputs to be supported. The ADC reference voltage can be configurable in differential and single ended modes. The controller supports pen touching screen detection for automatically interrupting the processor to measure only as needed.

15.2 Hardware Operation

The touch screen controller includes the following features:

- Supports 12-bit, 125 KHz ADC
- Supports ratiometric measurements drivers configurable in single ended or differential (ratiometric) topologies
- Supports either built-in voltage reference generator or external reference voltage
- Supports 4-wire and 5-wire touch screens with five inputs channels for touch screen purpose measurement (x+, x–, y+, y–, w)
- Supports general purpose measurements (for example temperature, voltage) with three input channels (aux0, aux1, aux2)
- Two independent measurement queues (TCQ for touch screen purpose, GCQ for general purpose measurement)
- Includes two independent FIFOs, each with 16 entries × 16 bits, for storing TCQ and GCQ conversion results
- Supports a touch detection interrupt feature to awaken the system from sleep mode
- Supports three power modes: always-off mode, power-saving mode, always-on mode
- Configurable pen down de-bounce logic
- Configurable LCD noise reducing logic
- Configurable settling time before each measurement
- Configurable multi-sample for each measurement

15.3 Software Operation

The ADC driver implements a complete IOCTL interface. Applications use the IOCTL interface to operate the ADC. The supported operations of the IOCTL interface are init, deinit, conversion with single channel, and conversion with multiple channels. The touch screen driver is designed as a Linux standard input device. It uses some functions provided by the ADC driver to get the samples of the X and Y values, and then transfers these values to the Linux input subsystem.

15.4 Source Code Structure

Table 15-1 shows the ADC driver source files available in the directory

`<ltib_dir>/rpm/BUILD/linux/drivers/mxc/adc.`

Table 15-1. ADC Driver Files

File	Description
`imx_adc.c`	Implementation file
`imx_adc_reg.h`	Header file

Table 15-2 shows the touch screen driver source files found in the directory

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

Table 15-2. Touch Screen Driver Files

File	Description
`imx_adc_ts.c`	Touch screen driver implementation file

15.5 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:

- IMX_ADC—Provided for the ADC driver. In the menuconfig, this option is found under

  Device Driver > MXC support drivers > i.MX ADC support > i.MX ADC.
- TOUCHSCREEN_IMX_ADC—Provided for the Touch screen driver. This option depends on the IMX_ADC configuration option. In the menuconfig, this option is found under

  Device Driver > Input device support > Touchscreens > Freescale i.MX ADC touchscreen input driver.

15.6 Programming Interface (Exported API)

The ADC driver, `imx_adc_ts.c`, provides a complete IOCTL programming interface to control the ADC hardware. The application interface to the ADC driver is the standard POSIX device interface (for example, open, close IOCTL). The application interface to the touch screen driver is the standard Linux input device interface.

15.7 Interrupt Requirements

The touch screen module generates interrupts when the pen is down. The ADC driver does not generate interrupts.

Chapter 16
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.

16.1 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.

16.2 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

16.3 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

16.4 Source Code Structure

Table 16-1 shows the source files available in the

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

Table 16-1. Ethernet Driver Files

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

16.5 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 17
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).

17.1 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. In addition, the FEC supports 10/100 Mbps RMII.

A brief overview of the device functionality is provided here. For details see the FEC chapter of the *i.MX25 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 and RMII modes uses a subset of the 18 signals. These signals are listed in Table 17-1.

Table 17-1. Pin Usage in MII, RMII 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 17-1. Pin Usage in MII, RMII 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.MX25 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.

17.2 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

17.3 Source Code Structure

Table 17-2 shows the source files available in the `<ltib_dir>/rpm/BUILD/linux/drivers/net directory.`

Table 17-2. 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.

17.4 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.

17.5 Programming Interface

Table 17-2 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.

17.5.1 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;
```

17.5.2 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`.

Chapter 18
DryIce Driver

This chapter describes the DryIce driver for Linux that provides low-level encryption key storage. The DryIce driver controls the key management elements of the Dry Ice hardware block of the i.MX device. The supported features include key establishment and selection as well as tamper detection. A different driver controls the Real Time Clock of the Dry Ice block as described in Chapter 26, “Real Time Clock (RTC) (DryIce) Driver”.

18.1 Dry Ice Driver Features and Capabilities

The Dry Ice peripheral provides the following features:

- Volatile storage for a software-programmable secret key and a hardware-generated secret key
- Key selection for a hardware encryption engine
- Non-secure RTC with alarm
- Secure RTC with alarm and monotonic counter
- Tamper detection circuits to monitor voltage, temperature and clock inputs, as well as wire mesh and external tamper detect pins
- Locks to protect against re-provisioning or re-configuration
- Separately supplied LP domain to maintain clock, counter and tamper detection when IC is powered down.

The Secure Hardware (SHW) driver API is a largely SHW-independent API, which integrates a number of underlying SHW-specific drivers. The idea is that applications in both user and kernel mode interface to the SHW API rather than the SHW-specific drivers themselves. The major exception to this is the Security Controller (SCC) driver, which has a specific kernel-mode interface. The concept is illustrated in Figure 18-1.

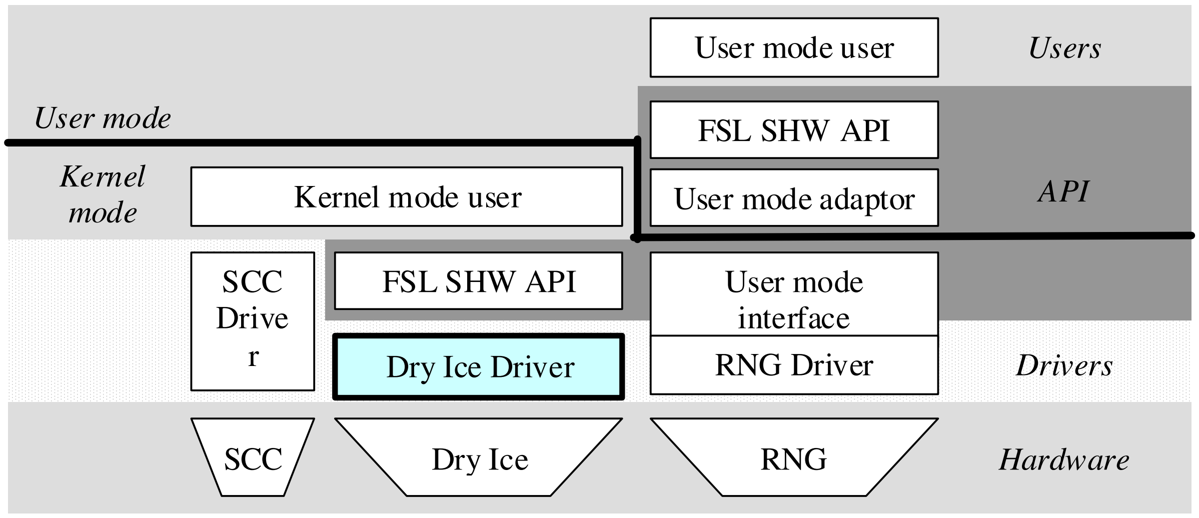

Figure 18-1. Software Architecture

18.2 Driver Requirements

Table 18-1 shows the Dry Ice driver features.

Table 18-1. DryIce Features

Interface Name	Description
Set Programmed Key	Write a given key into the Dry Ice Programmed Key
Get Programmed Key	Read out the Dry Ice Programmed Key to a given buffer
Release Programmed Key	Allow a fresh Programmed Key to be set
Set Random Key	Generate and load a new Dry Ice Random Key
Select Key	Select the key to use in the SCC
Check Key Selection	Confirm the selected key is used in the SCC
Release Key Selection	Allow a fresh key selection to be made
Get Tamper Event	Return Dry Ice tamper detection status and optional timestamp

The DryIce driver is integrated into the FSL SHW API. Most of the function calls can be blocking or non-blocking.

18.3 Driver Software Operation

The Dry Ice driver is integrated into the FSL SHW API, and key management functions should be accessed at the FSL SHW API level. See the DOXYGEN documentation provided driver implementation details (see files in Drivers > mxc > security)

18.4 Driver Source Code Structure

Table 18-2 shows the DryIce driver source files that are available in the directory, `<ltib_dir>/rpm/BUILD/linux/drivers/mxc/security`

Table 18-2. Dry Ice Driver Files

File	Description
`dryice.c`	Implementation of the dryice driver
`dryice.h`	Interface definitions of the dryice driver

18.5 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, please do as follows:

- Enable DryIce driver:

Device Drivers > MXC Support Drivers > MXC Security Drivers > MXC DryIce driver

18.6 Hardware Configuration

The following jumpers must be set on the i.MX25 CPU board:

- J7[1–2]
- J7[9-10]
- J7[11–12]

All others must be left open.

Chapter 19 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

19.1 Hardware Overview

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

19.1.1 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.

19.1.2 Secure RAM

Figure 19-1 shows the SCC-Secure RAM and its modules. Individual modules are described in the following sections.

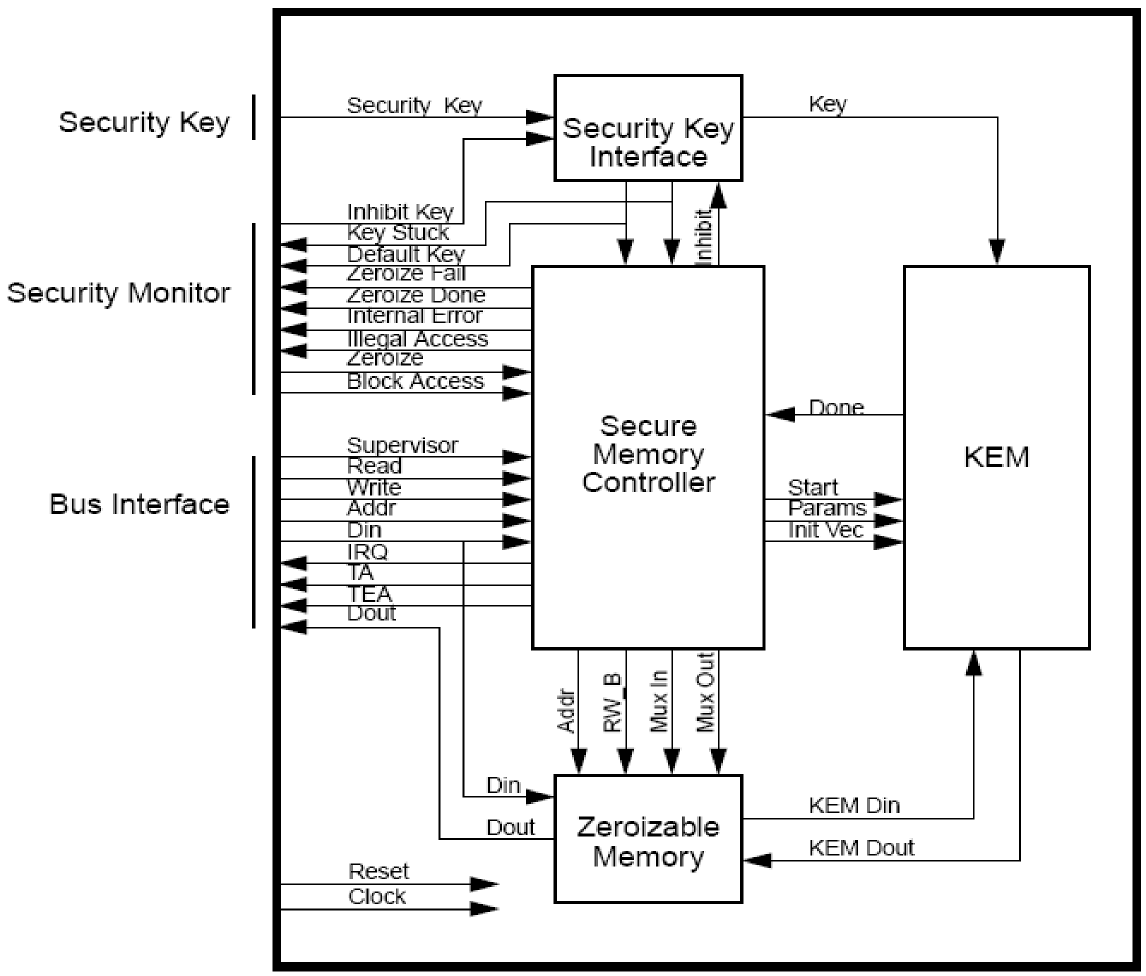

Figure 19-1. Secure RAM Block Diagram

19.1.3 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.

19.1.4 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.

19.1.5 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.

19.1.6 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.

19.1.7 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 19-2 shows a block diagram of the SMN.

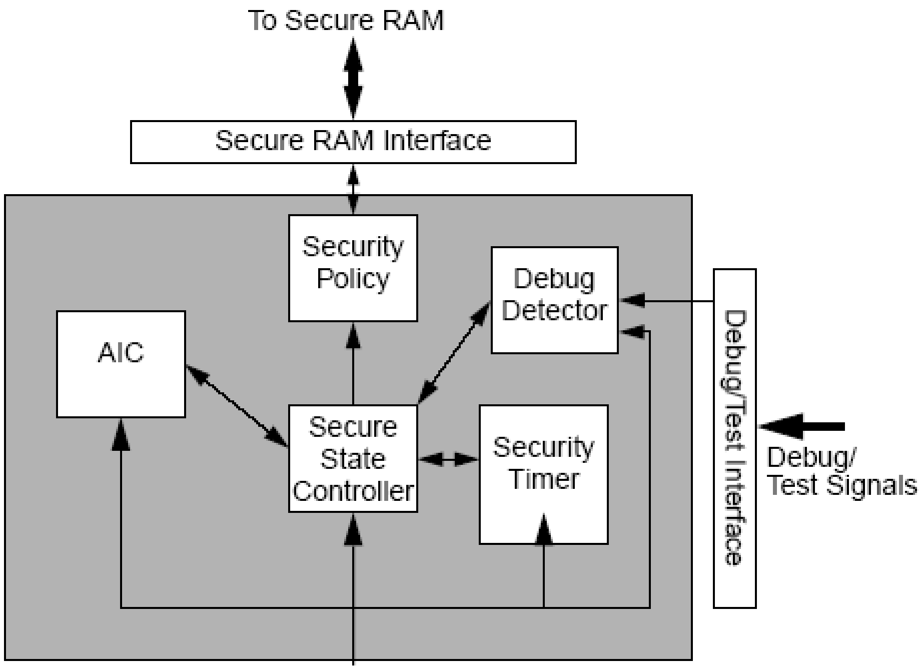

Figure 19-2. Security Monitor Block Diagram

19.1.8 Secure State Controller

The Secure State Controller, shown in Figure 19-3, is a state machine that controls the security states of the chip.

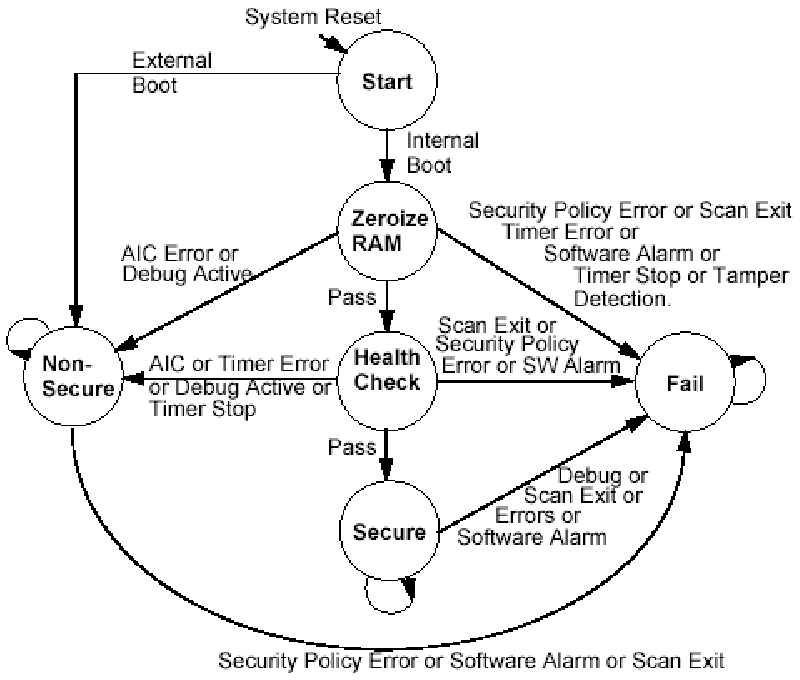
System Reset
External Boot
Start
Internal Boot
Zeroize RAM
AIC Error or Debug Active
Security Policy Error or Scan Exit Timer Error or Software Alarm or Timer Stop or Tamper Detection.
Pass
Scan Exit or Security Policy Error or SW Alarm
Non-Secure
AIC or Timer Error or Debug Active or Timer Stop
Health Check
Fail
Pass
Secure
Debug or Scan Exit or Errors or Software Alarm
Security Policy Error or Software Alarm or Scan Exit

Figure 19-3. Secure State Controller State Diagram

19.1.9 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.

19.1.10 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.

19.1.11 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.

19.1.12 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.

19.2 Software Operation

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

19.2.1 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.

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 3DES 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.

19.3 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

19.4 Source Code Structure

This section contains the various files that implement the Security modules. Table 19-1 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 19-1. SCCDriver Files

File	Description
Makefile	Used to compile, link and generate the final binary image
`rng/rng_driver.c`	Contains the core driver
`rng/include/`	Contains the include files

Table 19-1. SCCDriver Files (continued)

File	Description
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.

19.5 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_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.

19.5.1 Source Code Configuration Options

19.5.1.1 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 *i.MX25 Multimedia Applications Processor Reference Manual* 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 20
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

20.1 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

20.2 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.

20.3 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

20.4 Software Operation

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

20.4.1 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.

20.4.2 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.

20.5 Driver Features

The I^{2}C driver supports the following features:

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

20.6 Source Code Structure

Table 20-1 shows the I^{2}C bus driver source files available in the directory:

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

Table 20-1. I^{2}C Bus Driver Files

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

20.7 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.

20.8 Programming Interface

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

20.9 Interrupt Requirements

The I^{2}C module generates many kinds of interrupts. The highest interrupt rate is associated with the transfer complete interrupt as shown in Table 20-2.

Table 20-2. I^{2}C 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^{2}C interface).

Chapter 21
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

21.1 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)

21.2 Software Operation

The following sections describe the CSPI software operation.

21.2.1 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 21-1 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.

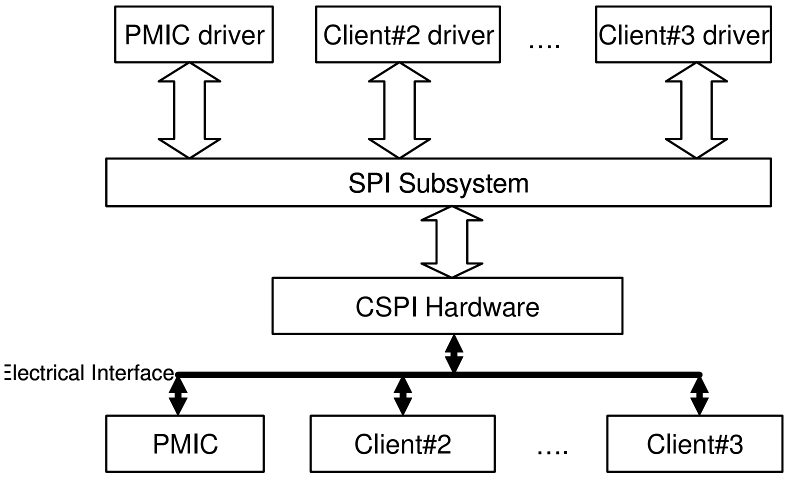

Figure 21-1. SPI Subsystem

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 21-2 shows how the different SPI drivers are layered in the SPI subsystem.

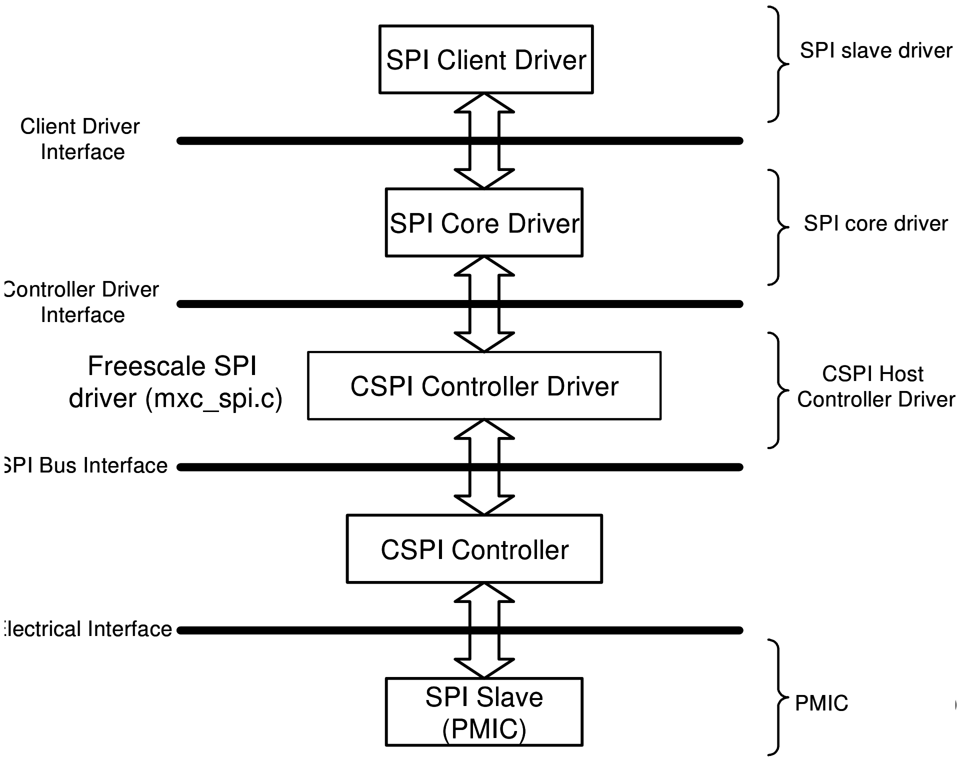

Figure 21-2. Layering of SPI Drivers in SPI Subsystem

21.2.2 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 support the SDMA function for CSPI1
- Does not currently implement the user space interface with the help of the device node entry but supports `sysfs` interface

21.2.3 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.

21.2.4 CSPI Synchronous Operation

Figure 21-3 shows how the CSPI provides synchronous read/write operations.

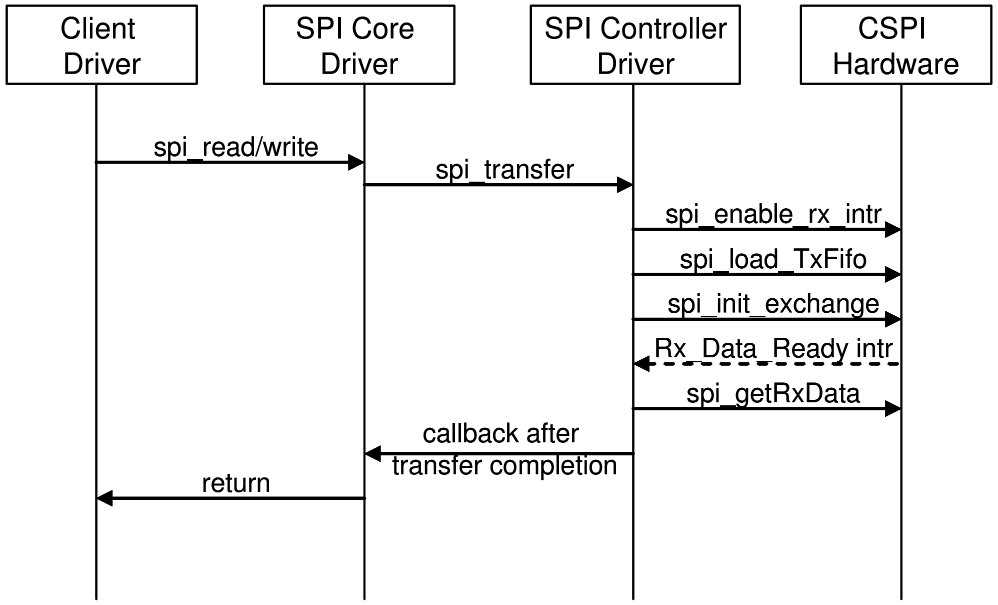

Figure 21-3. CSPI Synchronous Operation

21.3 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

21.4 Source Code Structure

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

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

Table 21-1. CSPI Driver Files

File	Description
mxc_spi.c	SPI Master Controller driver

21.5 Menu Configuration Options

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

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

Device Drivers > SPI Support.

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

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

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

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

  Device Drivers > SPI Support > LOOPBACK Testing of CSPIs.

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

21.6 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).

21.7 Interrupt Requirements

The SPI interface generates interrupts. CSPI interrupt requirements are listed in Table 21-2.

Table 21-2. 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 22
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

22.1 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.

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.

22.2 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 22-1 shows how the MMC-related drivers are layered.

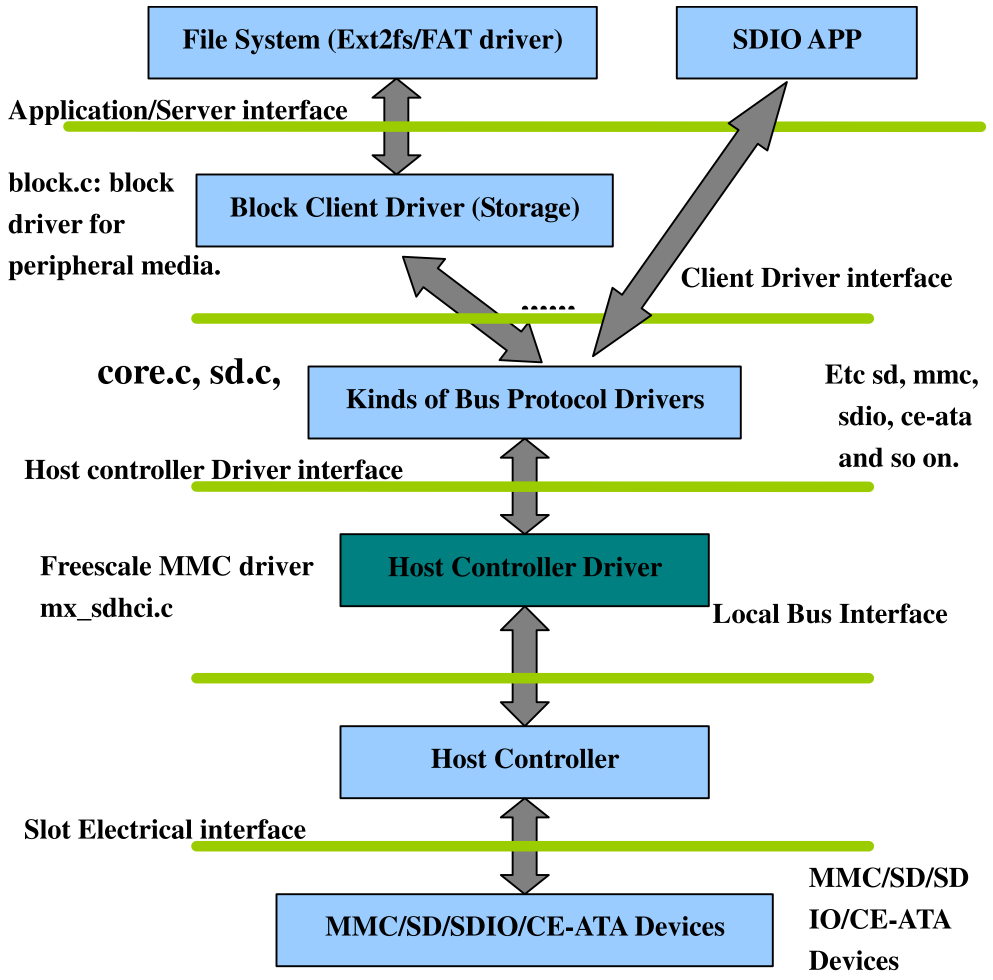
File System (Ext2fs/FAT driver)
SDIO APP
Application/Server interface
block.c: block driver for peripheral media.
Block Client Driver (Storage)
Client Driver interface
......
core.c, sd.c,
Kinds of Bus Protocol Drivers
Etc sd, mmc, sdio, ce-ata and so on.
Host controller Driver interface
Freescale MMC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

Figure 22-1. MMC Drivers Layering

22.3 Driver Features

The MMC driver supports the following features:

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

22.4 Source Code Structure

Table 22-1 shows the eSDHC source files available in the source directory: `<ltib_dir>/rpm/BUILD/linux/drivers/mmc/host/`.

Table 22-1. eSDHC Driver Files MMC/SD Driver Files

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

22.5 Menu Configuration Options

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

- CONFIG_MMC—Build support for the MMC bus protocol. In menuconfig, this option is available under

  Device Drivers > MMC/SD/SDIO Card support

  By default, this option is Y
- CONFIG_MMC_BLOCK—Build support for MMC block device driver, which can be used to mount the file system. In `menuconfig`, this option is available under

  Device Drivers > MMC/SD Card Support > MMC block device driver

  By default, this option is Y
- CONFIG_MMC_IMX_ESDHCI—Driver used for the i.MX eSDHC ports. In menuconfig, this option is found under

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

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

  This option is dependent on CONFIG_MMC_IMX_ESDHCI. By default, this option is not set and DMA mode is used.
- CONFIG_MMC_UNSAFE_RESUME—Used for embedded systems which use a MMC/SD/SDIO card for rootfs. In menuconfig, this option is found under

  Device drivers > MMC/SD/SDIO Card Support > Allow unsafe resume

22.6 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 23 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.

23.1 Hardware Operation

Refer to the *i.MX25 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.

23.2 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.

23.3 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

23.4 Source Code Structure

Table 23-1 shows the UART driver source files that are available in the directory:

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

Table 23-1. 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 23-2 shows the header files associated with the UART driver.

Table 23-2. 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-mx35/board-mx35_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-mx25`. The source file contains UART configuration data and calls to register the device with the platform bus.

23.5 Configuration

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

23.5.1 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.

23.5.2 Source Code Configuration Options

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

23.5.2.1 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 23-5.

23.5.2.2 Board Configuration Options

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

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx25/board-mx25_.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

23.6 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.

23.7 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 23-3.

Table 23-3. 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.

23.8 Device Specific Information

23.8.1 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 23-4.

23.8.2 Board Setup Configuration

Table 23-4. UART General Configuration

Platform	Number of UARTs	Max Baudrate
i.MX25	5	4000000 (4 Mbps)

Table 23-5. UART Active/Inactive Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	1	1	0	0	0	--

Table 23-6. UART IRDA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	NO_IRDA	NO_IRDA	NO_IRDA	NO_IRDA	NO_IRDA	--

Table 23-7. UART Mode Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	MODE_DCE	MODE_DCE	MODE_DCE	MODE_DTE	MODE_DTE	--

Table 23-8. UART Shared Peripheral Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	-1	-1	SPBA_UART3	SPBA_UART4	SPBA_UART5	--

Table 23-9. UART Hardware Flow Control Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	1	1	1	1	1	--

Table 23-10. UART DMA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	0	0	1	1	1	--

Table 23-11. UART DMA RX Buffer Size Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	0	0	512	512	512	--

Table 23-12. UART UCR4_CTSTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	16	16	16	16	16	--

Table 23-13. UART UFCR_RXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	16	16	16	16	16	--

Table 23-14. UART UFCR_TXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	16	16	16	16	16	--

Table 23-15. UART Interrupt Mux Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	INTS_MUXED	INTS_MUXED	INTS_MUXED	INTS_MUXED	INTS_MUXED	--

Table 23-16. UART Interrupt 1 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	INT_UART1	INT_UART2	INT_UART3	INT_UART4	INT_UART5	--

Table 23-17. UART Interrupt 2 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	-1	-1	-1	-1	-1	--

Table 23-18. UART interrupt 3 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX25	-1	-1	-1	-1	-1	--

23.9 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 24
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

24.1 Architectural Overview

A USB host system is composed of a number of hardware and software layers. Figure 24-1 shows a conceptual block diagram of the building block layers in a host system that support USB 2.0.

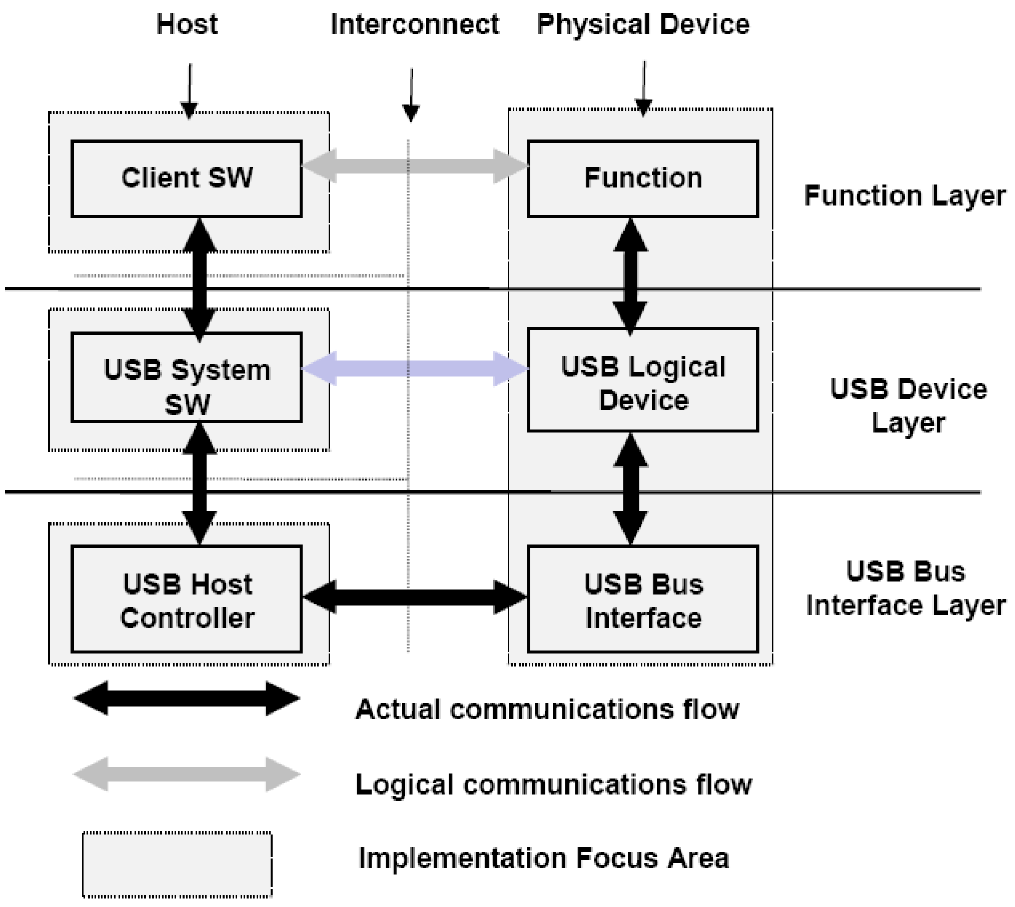

Figure 24-1. USB Block Diagram

24.2 Hardware Operation

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

24.3 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.

24.4 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

24.5 Source Code Structure

Table 24-1 shows the source files available in the source directory, `<ltib_dir>/rpm/BUILD/linux/drivers/usb`.

Table 24-1. 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 24-2 shows the platform related source files.

Table 24-2. 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 24-3 shows the platform-related source files in the directory:
`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx25/`

Table 24-3. USB Platform Header Files

File	Description
usb_dr.c	Platform-related initialization
usb_h2.c	Platform-related initialization

Table 24-4 shows the common platform source files in the directory:

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

Table 24-4. USB Common Platform Files

File	Description
utmixc.c	Internal UTMI transceiver driver
usb_common.c	Common platform related part of USB driver

24.6 Menu Configuration Options

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

- CONFIG_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_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_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_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.

24.7 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.

24.8 Default USB Settings

Table 24-5 shows the default USB settings.

Table 24-5. Default USB Settings

Platform	OTG HS	OTG FS	Host1	Host2(HS)	Host2(FS)
i.MX25 3DS	enabled	N/A	N/A	N/A	enabled

Only TO 1.1 silicon is supported for i.MX25.

Chapter 25
FlexCAN Driver

25.1 Driver Overview

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

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

25.2 Hardware Operation

For the information on hardware operations, see the *i.MX25 Multimedia Applications Processor Reference Manual*.

25.3 Software Operation

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

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

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

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

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

- `ifconfig can0 up`
- `ifconfig can0 down`

25.4 Source Code Structure

Table 25-1 shows the driver source file available in the directory, `<ltib_dir>/rpm/BUILD/linux/drivers/net/can/flexcan.`

Table 25-1. FlexCAN Driver Files

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

25.5 Linux Menu Configuration Options

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

- CONFIG_CAN – Build support for PF_CAN protocol family. In `menuconfig`, this option is available under

  Networking > CAN bus subsystem support.

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

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

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

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

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

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

Chapter 26
Real Time Clock (RTC) (DryIce) Driver

Each i.MX processor has an integrated RTC module. The RTC is used to keep the time and date while the system is turned off. The driver can also generate an interrupt using the alarm feature (AIE).

26.1 Hardware Operation

The RTC has its own 32.768 KHz clock, which is used to increment a 47-bit counter. The RTC also includes a 47-bit alarm register which is used to generate an interrupt whenever the value in the timer register matches the value contained in the alarm register. The RTC is part of the LP (Low Power) domain of the i.MX chip so that it can maintain time, even when the processor is powered off. The RTC hardware cannot generate periodic interrupts.

26.2 Software Operation

The RTC module software implementation is through an RTC driver. Besides the initialization function, it provides IOCTL `f`unctions to set up the RTC timer, interrupt, and so on. Since the RTC driver does not deal with fractional seconds, hardware time and alarm values are truncated to one-second resolution.

26.3 Requirements

This RTC implementation meets the following requirements:

- Implements all the functions required by Linux to provide the real time clock and alarm interrupt
- Conforms to the Linux coding standard as documented in the *Coding Conventions* chapter

26.4 Source Code Structure

Table 26-1 shows the RTC driver source files.

Table 26-1. RTC Driver Files

File	Description
rtc-imxdi.c	Implementation file

The source file, `rtc-imxdi.c`, implements the RTC functions.

26.5 Programming Interface

All the Linux RTC functions are implemented in the `drivers/rtc/class.c` and `drivers/rtc/interface.c` files. The `include/linux/rtc.h` file specifies all the IOCTLs for RTC.

The following RTC IOCTLs are supported by this driver.

- `RTC_RD_TIME`
- `RTC_SET_TIME`
- `RTC_ALM_READ`
- `RTC_ALM_SET`
- `RTC_WKALM_RD`
- `RTC_WKALM_SET`
- `RTC_AIE_ON`
- `RTC_AIE_OFF`

See the API documentation for the detailed programming interface.

Chapter 27 SIM Driver

The SIM driver implements a Linux driver interface to the Subscriber Identification Module (SIM).

Table 27-1. Available Platforms

Module Name	Available Platform
SIM	i.MX25

The SIM driver has following features:

- Supports card insertion and removal events
- Supports T0 Smart Card, and compatible with the ISO7816-3 spec

27.1 Hardware Operation

The detailed hardware operation of SIM module is detailed in the hardware documentation.

27.2 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

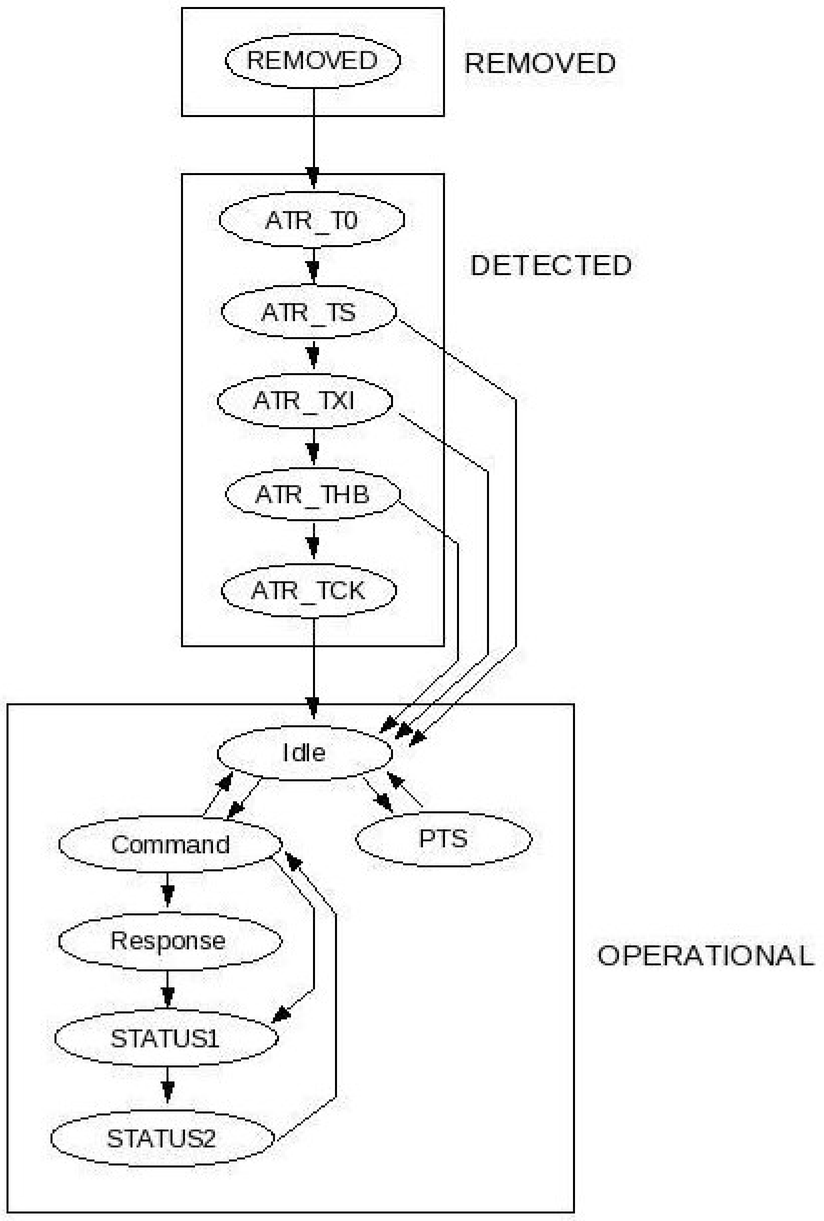

Figure 27-1. SIM driver

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.

27.3 Requirements

- Support T0 Smart Card, and compatible with the ISO7816-3 spec
- Conforms to the Linux coding standards

27.4 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 27-2. SIM Driver File List

File	Description
`imx_sim.c`	SIM driver
`mxc_sim_interface.h`	Header file defining the programming interfaces and so on

27.5 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.

27.6 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 28 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.

28.1 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.

28.2 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.

28.3 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.

28.3.1 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

28.3.2 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.

28.3.3 Source Code Structure

Table 28-1 shows the source files for WDOG drivers that are in the following directory:

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

Table 28-1. 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`

28.3.4 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 29 Frequently Asked Questions

29.1 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.

29.2 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.

29.3 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.

29.4 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.

29.5 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.

29.6 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.

29.7 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.