i.MX35 PDK Linux

Reference Manual

Part Number: 926-77210
Rev. 2009.12
02/2010

freescale™ semiconductor

How to Reach Us:

Home Page:
www.freescale.com

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

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

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

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

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

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

Contents

About This Book

Chapter 1
Introduction

Chapter 2
Architecture

Chapter 3
Machine Specific Layer (MSL)

Chapter 4
Smart Direct Memory Access (SDMA) API

Chapter 1
MC9S08DZ60-PMIC Protocol Driver

Chapter 6
MC13892 Regulator Driver

Chapter 7
MC13892 Digitizer Driver

Chapter 1
MC13892 RTC Driver

Chapter 2
i.MX35 Low-level Power Management Driver

Chapter 3
Image Processing Unit (IPU) Drivers

Chapter 4
Video for Linux Two (V4L2) Driver

Chapter 5
TV Decoder (TV-In) Driver

Chapter 6
OmniVision Camera (OV2640) Driver

Chapter 7
Advanced Linux Sound Architecture (ALSA)
System on a Chip (ASoC) Sound Driver

Chapter 8
The Sony/Philips Digital Interface (S/PDIF) Rx/Tx Driver

Chapter 9
NOR Flash Memory Technology Device (MTD) Driver

Chapter 10
NAND Flash Memory Technology Device (MTD) Driver

Chapter 11
SMSC LAN9217 Ethernet Driver

Chapter 12
Fast Ethernet Controller (FEC) Driver

Chapter 13
Media Local Bus Driver

Chapter 14
Inter-IC (I2C) Driver

Chapter 15
I^2C Slave Driver

Chapter 16
Configurable Serial Peripheral Interface (CSPI) Driver

Chapter 17
Universal Asynchronous Receiver/Transmitter (UART) Driver

Chapter 18
MMC/SD/SDIO Host Driver

Chapter 19
ARC USB Driver

Chapter 20
FlexCAN Driver

Chapter 21
Real Time Clock (RTC) Driver

Chapter 22
ATA Driver

Chapter 23
Asynchronous Sample Rate Converter (ASRC) Driver

Chapter 24
Watchdog (WDOG) Driver

Chapter 25
FM Driver

Chapter 26
Graphics Processing Unit (GPU)

Chapter 27
Global Positioning System (GPS) Driver

Chapter 28
OProfile

Chapter 29
Frequently Asked Questions

Figures

2-1 BSP Block Diagram 2-1
2-2 SDMA Block Diagram 2-5
2-3 MTD Architecture 2-9
2-4 DPM High Level Design 2-17
2-5 DPM Architecture Block Diagram 2-17
4-1 SDMA Block Diagram 4-2
1-1 MC9S08DZ60 Device Driver 5-3
1-2 MC9S08DZ60 Connection With PMIC (i.MX35 3Stack v1.x) 5-3
1-3 MC9S08DZ60 Connection With PMIC (i.MX35 3Stack v2.x) 5-4
6-1 MC13892 Regulator Driver Architecture 6-3
2-1 DVFS Load Tracking Module Block Diagram 9-2
3-1 IPUv1 hardware module overview 10-2
3-2 Graphics/Video Drivers Software Interaction 10-3
4-1 Video4Linux2 Capture API Interaction 11-4
7-1 ALSA SoC Software Architecture 14-1
7-2 ALSA SoC Source File Relationship 14-4
8-1 S/PDIF Transceiver Data Interface Block Diagram 15-1
8-2 S/PDIF Rx Application Program Flow 15-6
9-1 Components of a File System in a Flash-Based System 16-2
13-1 MLB Device Top-Level Block Diagram 20-1
13-2 MLB Driver Architecture Diagram 20-3
16-1 SPI Subsystem 23-2
16-2 Layering of SPI Drivers in SPI Subsystem 23-2
16-3 CSPI Synchronous Operation 23-4
18-1 MMC Drivers Layering 25-3
19-1 USB Block Diagram 26-2
22-1 ATA Driver Layers 29-2
23-1 Audio Driver Interactions 30-2
25-1 FM Driver Software Operation 32-2
27-1 Barracuda GPS Coarse System Architecture Including Host CPU 34-1
27-2 GL GPS SW Architecture 34-2

Tables

1-1 Linux BSP Supported Features 1-2
2-1 MSL Directories 2-3
3-1 Interrupt Files 3-2
3-2 Memory Map Files 3-5
3-3 IOMUX Files 3-6
3-4 IOMUX Through GPIO Files 3-8
3-5 GPIO Files 3-9
3-6 EDIO Files 3-10
3-7 SPBA Files 3-11
4-1 SDMA Channel Usage 4-3
4-2 SDMA API Source Files 4-3
4-3 SDMA Script Files 4-3
1-1 MC9S08DZ60 Driver Sources File List 5-5
6-1 MC13892 Power Management Driver Files 6-4
7-1 MC13892 Digitizer Driver Files 7-3
1-1 MC9S08DZ60 RTC Driver Files 8-2
2-1 Low Power Modes 9-2
2-2 Source Code for i.MX35 PM Drivers 9-4
3-1 IPU Driver Files 10-7
3-2 IPU Global Header Files 10-7
4-1 V2L2 Driver Files 11-6
5-1 TV-In Driver Source File 12-1
6-1 Camera Driver Files 13-2
7-1 Stereo Codec SoC Driver Files 14-4
7-2 5.1 Codec SoC Driver Files 14-5
7-3 Bluetooth Codec SoC Driver Source Files 14-5
8-1 S/PDIF Rx Driver Interfaces 15-5
8-2 S/PDIF Driver Files 15-7
10-1 NAND MTD Driver Files 17-2
11-1 Ethernet Driver Files 18-2
12-1 Pin Usage in MII and SNI Modes 19-1
12-2 FEC Driver Files 19-3
13-1 MLB Driver Source File List 20-5
14-1 I2C Bus Driver Files 21-3
14-2 I2C Interrupt Requirements 21-3
15-1 I2C Bus Driver Files 22-2
16-1 CSPI Driver Files 23-4
16-2 CSPI Interrupt Requirements 23-5
17-1 UART Driver Files 24-3

17-2 UART Global Header Files 24-3
17-3 UART Interrupt Requirements 24-5
17-4 UART General Configuration 24-5
17-5 UART Active/Inactive Configuration 24-5
17-6 UART IRDA Configuration 24-5
17-7 UART Mode Configuration 24-5
17-8 UART Shared Peripheral Configuration 24-6
17-9 UART Hardware Flow Control Configuration 24-6
17-10 UART DMA Configuration 24-6
17-11 UART DMA RX Buffer Size Configuration 24-6
17-12 UART UCR4_CTSTL Configuration 24-6
17-13 UART UFCR_RXTL Configuration 24-6
17-14 UART UFCR_TXTL Configuration 24-6
17-15 UART Interrupt Mux Configuration 24-6
17-16 UART Interrupt 1 Configuration 24-6
17-17 UART Interrupt 2 Configuration 24-7
17-18 UART interrupt 3 Configuration 24-7
18-1 eSDHC Driver FilesMMC/SD Driver Files 25-4
19-1 USB Driver Files 26-4
19-2 USB Platform Source Files 26-4
19-3 USB Platform Header Files 26-4
19-4 USB Common Platform Files 26-5
19-5 Default USB Settings 26-7
20-1 FlexCAN Driver Files 27-2
21-1 RTC Driver File List 28-1
22-1 DMA Engine 29-1
22-2 LibATA Driver File List 29-3
22-3 Hardware Configuration for 3-Stack Boards 29-3
23-1 ASRC Source File List 30-3
23-2 ASRC Exported Functions 30-3
24-1 WDOG Driver Files 31-2
25-1 FM Driver Source and Header File List 32-3
26-1 GPU Related File List 33-2
27-1 UART Port 34-3
27-2 GPIO Control Signals 34-3
27-3 Hardware Dependent Parameters 34-4
27-4 hal Attributes 34-5
27-5 gll Attributes 34-5
27-6 GPS Driver Source Code 34-6
28-1 OProfile Source Files 35-3

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.MX35 PDK Linux Quick Start Guide*
- *BSP API Document (BSP Doxygen Code Documentation)*
- *i.MX35 PDK Linux User's Guide*
- *i.MX35 PDK Hardware User's Guide*
- *MCIMX35 Multimedia Applications Processors Reference Manual*, (*MCIMX35RM*)
- [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.MX35 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.MX35,

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 ARM11 interrupt controller. • Timer (GPT): The General Purpose Timer (GPT) is set up to generate an interrupt as programmed to provide OS ticks. Linux facilitates timer use through various functions for timing delays, measurement, events, alarms, high resolution timer features, and so on. Linux defines the MSL timer API required for the OS-tick timer and does not expose it beyond the kernel tick implementation. • GPIO/EDIO/IOMUX: The GPIO and EDIO components in the MSL provide an abstraction layer between the various drivers and the configuration and utilization of the system, including GPIO, IOMUX, and external board I/O. The IO software module is board-specific, and resides in the MSL layer as a self-contained set of files. I/O configuration changes are centralized in the GPIO module so that changes are not required in the various drivers. • SPBA: The Shared Peripheral Bus Arbiter (SPBA) provides an arbitration mechanism among multiple masters to allow access to the shared peripherals. The SPBA implementation under MSL defines the API to allow different masters to take or release ownership of a shared peripheral.	Chapter 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.MX35
Power Management IC (PMIC) Drivers			
PMIC Protocol	The MCU-PMIC protocol device driver provides low-level read/write access to MCU-PMIC hardware control registers.	Chapter 1, “MC9S08DZ60-PMIC Protocol Driver”	i.MX35

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

Feature	Description	Chapter Source	Applicable Platform
MC13892 Regulator	MC13892 regulator driver provides the low-level control of the power supply regulators, setting voltage level and enable/disable regulators.	Chapter 6, "MC13892 Regulator Driver"	i.MX35
MC13892 RTC	MC13892 RTC driver for Linux provides the access to PMIC RTC control circuits	Chapter 1, "MC13892 RTC Driver"	i.MX35
MC13892 Digitizer Driver	MC13892 digitizer driver for Linux that provides low-level access to the PMIC analog-to-digital converters	Chapter 7, "MC13892 Digitizer Driver"	i.MX35
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 2, "i.MX35 Low-level Power Management Driver"	i.MX35
DVFS	The Dynamic Voltage Frequency Scaling (DVFS) device driver allows simple dynamic voltage frequency scaling. The frequency of the core clock domain and the voltage of the core power domain can be changed on the fly with all modules continuing their normal operations.	Chapter 10, "Dynamic Voltage Frequency Scaling (DVFS) Driver"	i.MX35
Multimedia Drivers			
IPU	The Image Processing Unit (IPU) is designed to support video and graphics processing functions and to interface with video/still image sensors and displays. The IPU driver is a self-contained driver module in the Linux kernel. It contains a custom kernel-level API to manipulate logical channels. A logical channel represents a complete IPU processing flow. The IPU driver includes a frame buffer driver, a V4L2 device driver, and low-level IPU drivers.	Chapter 3, "Image Processing Unit (IPU) Drivers"	i.MX35
TV-IN (ADV7180)	The ADV7180 TV-IN driver is designed under Linux V4L2 architecture. It implements the V4L2 capture interface.	Chapter 5, "TV Decoder (TV-In) Driver"	i.MX35
V4L2 Output	The Video for Linux 2 (V4L2) output driver uses the IPU post-processing functions for video output. The driver implements the standard V4L2 API for output devices.	Chapter 4, "Video for Linux Two (V4L2) Driver"	i.MX35
V4L2 Capture	The Video for Linux 2 (V4L2) capture device includes two interfaces: the capture interface and the overlay interface. The capture interface records the video stream. The overlay interface displays the preview video.	Chapter 4, "Video for Linux Two (V4L2) Driver"	i.MX35
OmniVision Camera (OV2640)	The OV2640 Camera driver is designed under Linux V4L2 architecture. It implements V4L2 capture interface.	Chapter 6, "OmniVision Camera (OV2640) Driver"	i.MX35

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

Feature	Description	Chapter Source	Applicable Platform
GPU	Graphic Process Unit 2D graphic accelerator	Chapter 26, "Graphics Processing Unit (GPU)	i.MX35
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 7, "Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver"	i.MX35
S/PDIF	The S/PDIF driver is designed under the Linux ALSA subsystem. It implements one playback device for Tx and one capture device for Rx. MX35 supports the S/PDIF transceiver.	Chapter 8, "The Sony/Philips Digital Interface (S/PDIF) Rx/Tx Driver"	i.MX35
Memory Drivers			
NOR MTD	The NOR MTD driver is board-specific as it depends on the actual NOR Flash chip (Common Flash Interface or CFI-compliant) on the board and can have file systems, such as CRAMFS and JFFS2 on top of it. The driver implementation supports the lowest level operations on the Flash chip, such as read, write and erase. The NOR MTD supports XIP on Flash devices.	Chapter 9, "NOR Flash Memory Technology Device (MTD) Driver"	i.MX35
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 10, "NAND Flash Memory Technology Device (MTD) Driver"	i.MX35
Networking Drivers			
LAN9217 Ethernet	The SMSC LAN9217 Ethernet driver interfaces SMSC LAN9217-specific functions with the standard Linux kernel network module.	Chapter 11, "SMSC LAN9217 Ethernet Driver"	i.MX35

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

Feature	Description	Chapter Source	Applicable Platform
FEC	The FEC Driver performs the full set of IEEE 802.3/Ethernet CSMA/CD media access control and channel interface functions. The FEC requires an external interface adaptor and transceiver function to complete the interface to the Ethernet media. It supports half or full-duplex operation on 10 Mbps- or 100 Mbps-related Ethernet networks.	Chapter 12, “Fast Ethernet Controller (FEC) Driver”	i.MX35
Bus Drivers			
MLB	MediaLB is an on-PCB or inter-chip communication bus, specifically designed to standardize a common hardware interface and software API library.	Chapter 13, “Media Local Bus Driver”	i.MX35
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 14, “Inter-IC (I2C) Driver”	i.MX35
I^2C Slave	I^2C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices. The MXC I^2C slave driver is divided into two layers: the I^2C slave core and the I^2C slave chip driver. The I^2C core driver stays at top level and it is the interface for the registered I^2C slave device to the Linux device driver model. The I^2C slave chip driver handles the low-level hardware operation.	Chapter 15, “I2C Slave Driver”	i.MX35
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 16, “Configurable Serial Peripheral Interface (CSPI) Driver”	i.MX35
MMC/SD/SDIO - eSDHC	The MMC/SD/SDIO Host driver implements the standard Linux driver interface to eSDHC.	Chapter 18, “MMC/SD/SDIO Host Driver”	i.MX35
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 17, “Universal Asynchronous Receiver/Transmitter (UART) Driver”	i.MX35

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

Feature	Description	Chapter Source	Applicable Platform
General Drivers			
USB	The USB driver implements a standard Linux driver interface to the ARC USB-OTG controller.	Chapter 19, “ARC USB Driver”	i.MX35
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 20, “FlexCAN Driver”	i.MX35
ATA	The ATA module is an AT attachment host interface. Its main use is to interface with hard disk devices. The ATA driver is compliant with the ATA-6 standard, and supports the following protocols: • PIO mode 0, 1, 2, 3, and 4 • Multi-word DMA mode 0, 1, and 2 • Ultra DMA mode 0, 1, 2, 3, and 4and 3 with bus clocks of 50MHz or higher • Ultra DMA mode 5 with bus clock of 80MHz or higher. It supports the IDE and LibATA interfaces.	Chapter 22, “ATA Driver”	i.MX35
RTC	This is the integrated Real Time Clock (RTC) module. The RTC is used to keep the time and date while the system is turned off. Additionally, it provides the PIE (periodic interrupt at a specific frequency) and AIE (Wake up the system by providing an alarm) features. The RTC driver is designed to support MXC RTC modules to keep the time and date.	Chapter 21, “Real Time Clock (RTC) Driver”	i.MX35
ASRC	The Asynchronous Sample Rate Converter (ASRC) driver provides the interfaces to access the asynchronous sample rate converter module.	Chapter 23, “Asynchronous Sample Rate Converter (ASRC) Driver”	i.MX35
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 24, “Watchdog (WDOG) Driver”	i.MX35
FM (Si4702)	The FM (Si4702) driver provides the interfaces to control Silicon Laboratories Si4702 FM tuner integrated circuit.	Chapter 25, “FM Driver”	i.MX35
GPS	The communications driver provides a serial interface to the core driver and communicates with the external GPS chip set.	Chapter 27, “Global Positioning System (GPS) Driver	i.MX35

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

Feature	Description	Chapter Source	Applicable Platform
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.MX35
uBoot	uBoot is an open source boot loader.	See uBoot User guide	i.MX35
GUI			
gnome	gnome is a Network Object Model Environment supported by the GUN.	See Gnome mobile Note	i.MX35
Tools			
OProfile	OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.	Chapter 28, “OProfile”	i.MX35

Chapter 2 Architecture

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

2.1 Linux BSP Block Diagram

Figure 2-1 shows the architecture of the BSP for the i.MX family of processors. It consists of user-space executables, standard kernel components that come from the Linux community, as well as hardware-specific drivers and functions provided by Freescale for the i.MX processors.

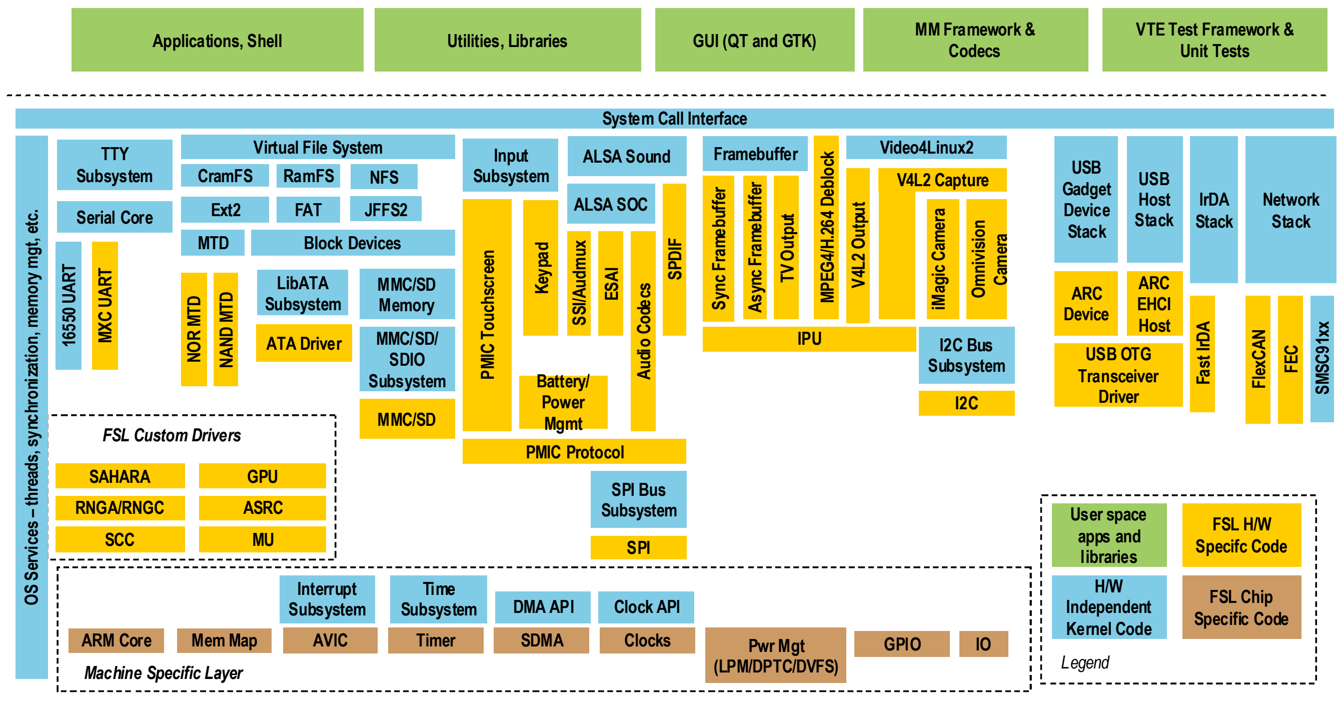


Figure 2-1. BSP Block Diagram

2.2 Kernel

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

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

The serial port is labeled UART-DCE on the i.MX35 3-Stack board.

- Embedded mode
- Module loading/unloading
- ARM11
- File formats supported: ELF binaries, a.out and ECOFF
- Block devices: Loopback, Ramdisk
- i.MX internal UART
- File systems: ext2, dev, proc, sysfs, cramfs, ramfs, JFFS2, FAT, pramfs
- 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.MX35 3-Stack	<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx35

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 ARM11™ vectored interrupt controller (AVIC) .

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
- Static mapping of one interrupt source as Fast Interrupt Request (FIQ)

Vectored interrupts and fast interrupt are not supported.

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.

i.MX35 provides static mapping of one interrupt source as Fast Interrupt Request (FIQ). Vectored interrupts and fast interrupt are not supported.

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

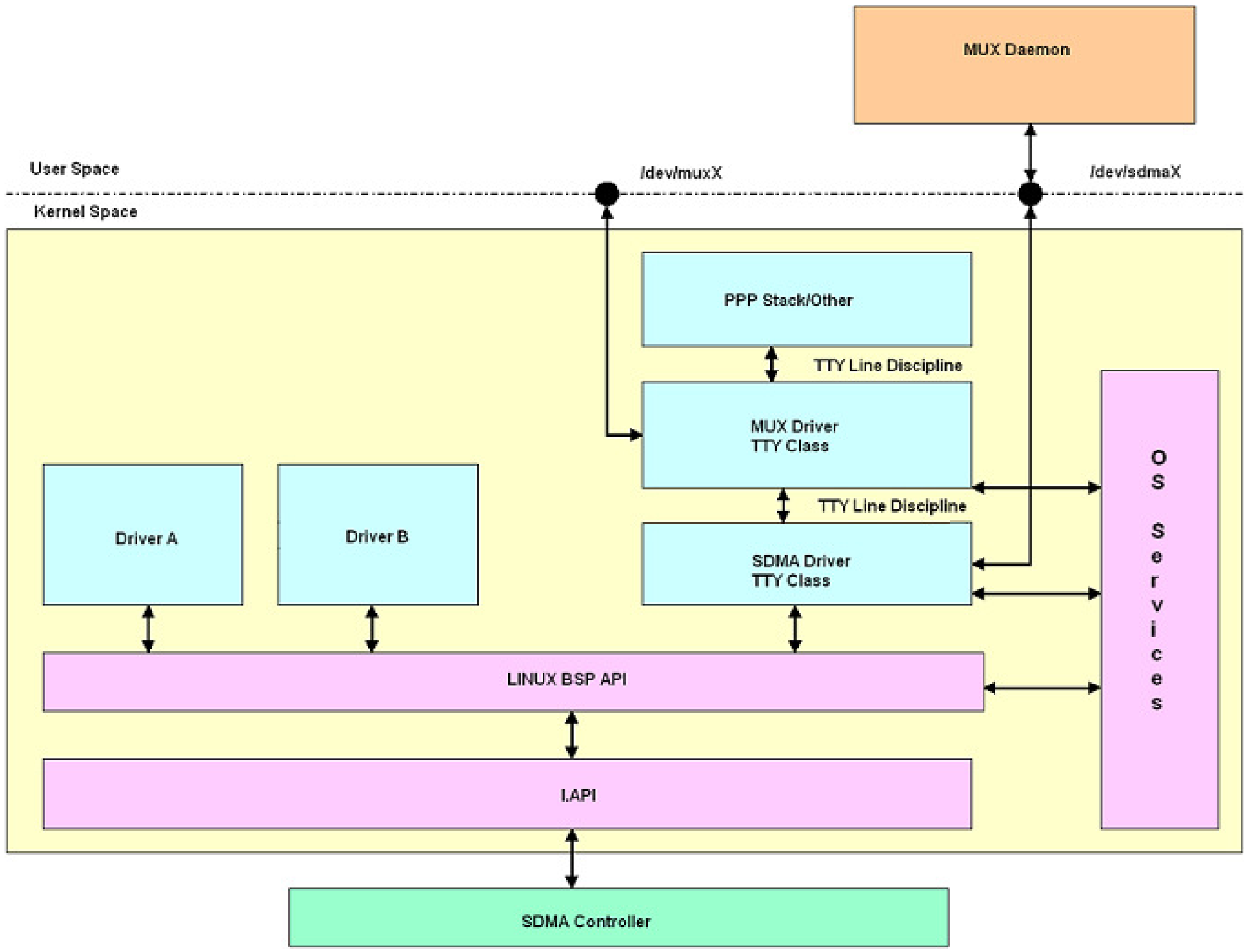


Figure 2-2. SDMA Block Diagram

The SDMA API is present on i.MX35.

2.2.2.5 Input/Output (I/O)

The Input/Output (I/O) component in the MSL provides an abstraction layer between the various drivers and the configuration and utilization of the system, including GPIO, IOMUX, and external board I/O. The I/O software module is board-specific and resides in the MSL layer as a self-contained set of files. It provides the following features as part of a custom kernel-space API:

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

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

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

2.2.2.6 Shared Peripheral Bus Arbiter (SPBA)

The SPBA provides an arbitration mechanism to allow multiple masters to have access to the shared peripherals. The SPBA implementation under MSL defines the API to allow different masters to take or release ownership of a shared peripheral. These functions are also exported so that they can be used by other loadable modules.

2.3 Drivers

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

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

2.3.1 Universal Asynchronous Receiver/Transmitter (UART) Driver

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

2.3.1.1 UART Driver

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

- Interrupt-driven and SDMA-driven transmit/receive of characters
- Standard Linux baud rates up to 1.5 Mbps
- Transmitting and receiving characters with 7-bit and 8-bit character lengths
- Transmitting one or two stop bits
- Odd and even parity
- XON/XOFF software flow control
- CTS/RTS hardware flow control (both interrupt-driven software controlled hardware flow control and hardware-driven hardware flow control)

- TIOCMGET IOCTL to read the modem control lines. Supports the constants TIOCM_CTS and TIOCM_CAR, TIOCM_RI (only in DTE mode) only
- TIOCMSET IOCTL sets modem control lines. Supports the constants TIOCM_RTS and TIOCM_DTR only
- Send and receive of break characters through the standard Linux serial API
- Recognize frame and parity errors
- Ability to ignore characters with break, parity and frame errors
- Get and set UART port information through the TIOCGSSERIAL and TIOCSSERIAL TTY IOCTLs
- Slow IrDA (IrDA at or below 115200 baud)
- Power management features - suspends and resumes the UART ports
- The standard TTY layer IOCTL calls
- Includes console support which is needed to bring up the command prompt through one of the UART ports

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

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

2.3.2 Real-Time Clock (RTC) Driver

The RTC is the clock that keeps the date and time while the system is running and even when the system is inactive. The RTC implementation supports IOCTL calls to read time, set time, set up periodic interrupts, and set up alarms. Linux defines the RTC API.

2.3.3 Watchdog Timer (WDOG) Driver

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

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

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

2.3.4 SDMA API Driver

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

Functions of the SDMA API include:

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

2.3.5 Image Processing Unit (IPU) Driver

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

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

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

2.3.6 Video for Linux 2 (V4L2) Driver

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

2.3.7 Sound Driver

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

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

2.3.8 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.

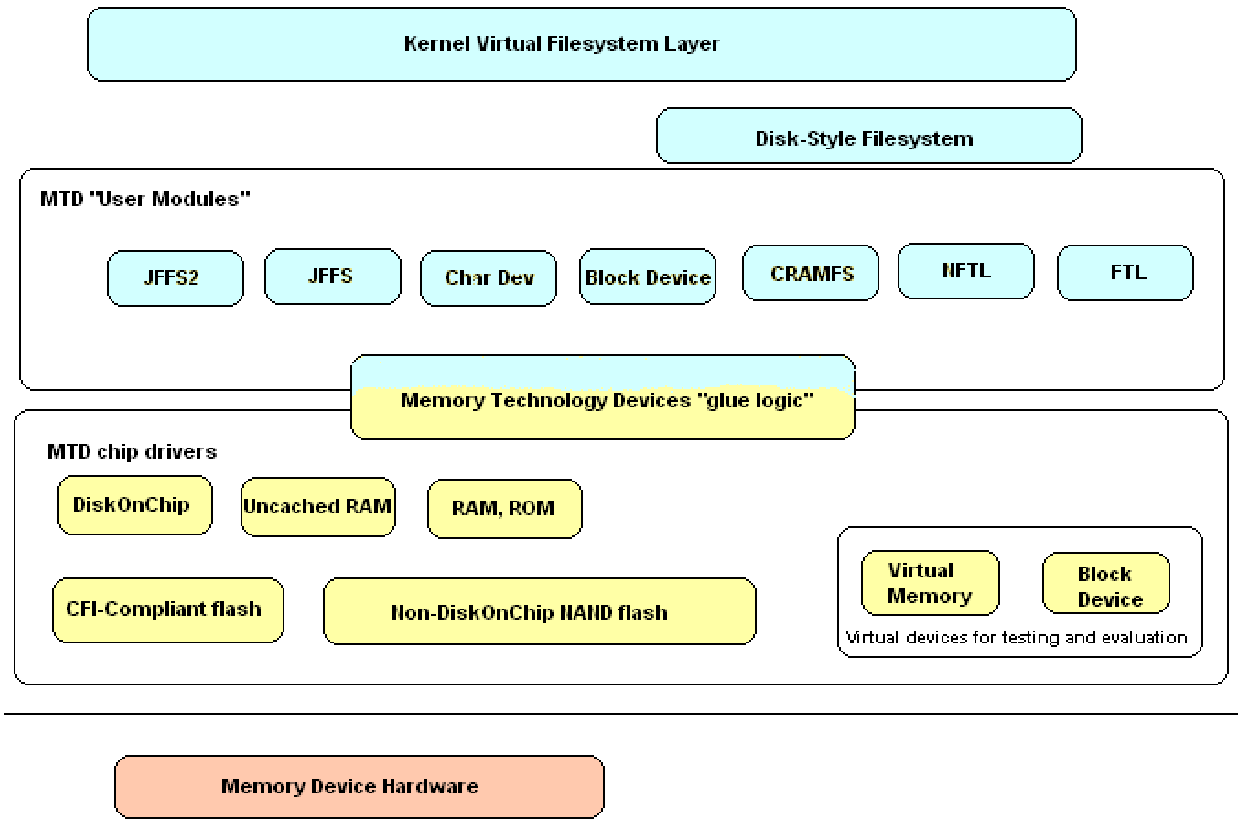


Figure 2-3. MTD Architecture

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.

2.3.8.1 NOR MTD Driver

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

2.3.9 Networking Drivers

The networking drivers are described in the next sections.

2.3.9.1 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.

2.3.9.2 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.

2.3.10 Disk Drivers

The disk drivers include the ATA driver.

2.3.10.1 ATA Disk Driver

The ATA module is an AT attachment host interface. Its main use is to interface with hard disk devices. The ATA driver is compliant to the ATA-6 standard, and supports the following protocols:

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

2.3.11 USB Driver

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

2.3.11.1 USB Host-Side API Model

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

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

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

The device model seen by USB drivers is relatively complex:

- USB supports four kinds of data transfer (control, bulk, interrupt, and isochronous). Two transfer types use bandwidth as it is available (control and bulk), while the other two types of transfer (interrupt and isochronous) are scheduled to provide guaranteed bandwidth.

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

2.3.11.2 USB Device-Side Gadget Framework

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

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

- Peripheral Controller Drivers—implement the Gadget API, and are the only layers that talk directly to the hardware. Different controller hardware needs different drivers, which may also need board-specific customization. These provide a software gadget device, visible in sysfs. This device can be thought of as being the virtual hardware to which the higher-level drivers are written.
- Gadget Drivers—use the Gadget API, and can often be written to be hardware-neutral. A gadget driver implements one or more functions, each providing a different capability to the USB host, such as a network link or speakers.
- Upper Layers, such as the network, file system, or block I/O subsystems—generate and consume the data that the gadget driver transfers to the host through the controller driver.

2.3.11.3 USB OTG Framework

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

- Gadget drivers test the is_otg flag, and use it to determine whether or not to include an OTG descriptor in each of their configurations.
- Gadget drivers may need changes to support the two new OTG protocols, exposed in new gadget attributes such as b_hnp_enable flag. HNP support should be reported through a user interface (two LEDs could suffice), and is triggered in some cases when the host suspends the peripheral. SRP support can be user-initiated just like remote wakeup, probably by pressing the same button.
- On the host side, USB device drivers need to be taught to trigger HNP at appropriate moments, using usb_suspend_device(). That also conserves battery power, which is useful even for non-OTG configurations.

- Also on the host side, a driver must support the OTG Targeted Peripheral List, a whitelist used to reject peripherals not supported with a given Linux OTG host. This whitelist is product-specific—each product must modify `otg_whitelist.h` to match its interoperability specification.

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

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

2.3.12 Security Drivers

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

2.3.12.1 Security Controller (SCC) Module Driver

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

- Autonomous hardware security state controller with debug inputs that are tied to all platform test access detection signals to trigger a security shutdown
- Controls to ensure supervisory mode only configuration access
- Controls to ensure that high assurance internal boot is the only mechanism to reach the Secure state after Reset
- Autonomous hardware security state controller with debug inputs that are tied to all platform test access detection signals to trigger shutdown
- Self-clearing (zeroing) 2 Kbyte RAM block, which clears itself upon command and can therefore be used to store security sensitive Red data (that is, security sensitive plain text), such as cryptographic keys

- Security Timer which is an independent security watchdog timer whose time-out triggers a security violation
- Algorithm Sequence Checker (ASC) which can be used by software to force software synchronization to the ASCs internal linear feedback shift register (LFSR) as a software assurance check
- Bit Bank counter that can be used with the ASC to ensure that a scrambler function uses the same number of algorithm bits as traffic bits to ensure that no traffic data is accidentally left in the clear
- Plaintext/Ciphertext comparator that may be used to ensure that a cryptographic algorithm scrambler has not been replaced with a simple pattern EXOR function
- Some portion of the SCC is used during initial boot-up from the iROM
- Some portion is used as a security measure during runtime, for example, tampering of the hardware. This is used to clear the secure data either in the internal RAM or externally encrypted data RAM.
- Power management

2.3.13 General Drivers

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

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

2.3.13.1 MMC/SD Host Driver

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

- MMC/SD cards
- Standard MMC/SD commands
- 1-bit or 4-bit operation
- Card insertion and removal events
- Write protection signal

2.3.13.2 MMC/SD Slot Driver

The MMC/SD driver implements a standard Linux slot driver as well as a block driver interface to the MMC/SDHC controller. The interface to the upper layer follows the standard Linux driver API. This driver supports the following features:

- SDHC module supports MMC and SD cards
- MMC version 3.0 spec is supported. SD Memory Card spec 1.0 and SD I/O card spec 1.0 are supported.
- Hardware contains 32×16 bit data buffer built in

- Plug and play support
- 100 Mbps Maximum hardware data rate in 4-bit mode
- 1-bit or 4-bit operation
- For SD card access, only SD bus mode is supported. SPI mode is not supported.
- Supports card insertion and removal events
- Supports the standard MMC/SD/SDIO commands
- Supports Power management
- Supports set/reset of password or card lock/unlock commands
- Power management

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

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

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

The I^2C slave mode is not supported by this driver. Slave mode is supported by the I^2C slave driver, refer to Chapter 15, "I2C Slave Driver".

2.3.13.4 Configurable Serial Peripheral Interface (CSPI) Driver

The low-level Configurable Serial Peripheral Interface (CSPI) driver interfaces a custom, kernel-space API to the CSPI modules. It supports the following features:

- Interrupt-driven transmit/receive of SPI frames
- Multi-client management
- Priority management between clients
- SPI device configuration per client

DMA is not supported.

2.3.13.5 Dynamic Power Management (DPM) Driver

DPM refers to power management schemes implemented while programs are running. DPM focuses on system wide energy consumption while it is running. In any CPU-intensive application, lowering bus frequencies from their maximum performance points can result in system wide energy savings. DPM implementation includes the following data structures:

- Operating points
- Operating states
- Policies
- Policy manager

2.3.13.5.1 Policy Architecture

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

2.3.13.5.2 Operating Points

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

2.3.13.5.3 Operating States

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

2.3.13.5.4 Policy Managers

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

Figure 2-4 shows the high level design for DPM.

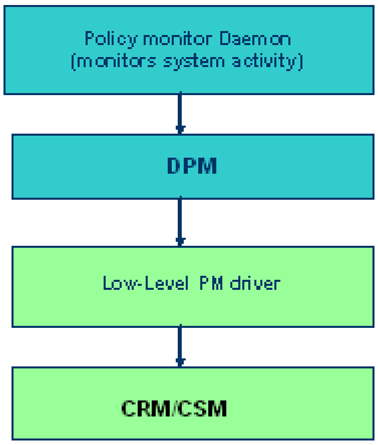


Figure 2-4. DPM High Level Design

Figure 2-5 shows the DPM architecture block diagram.

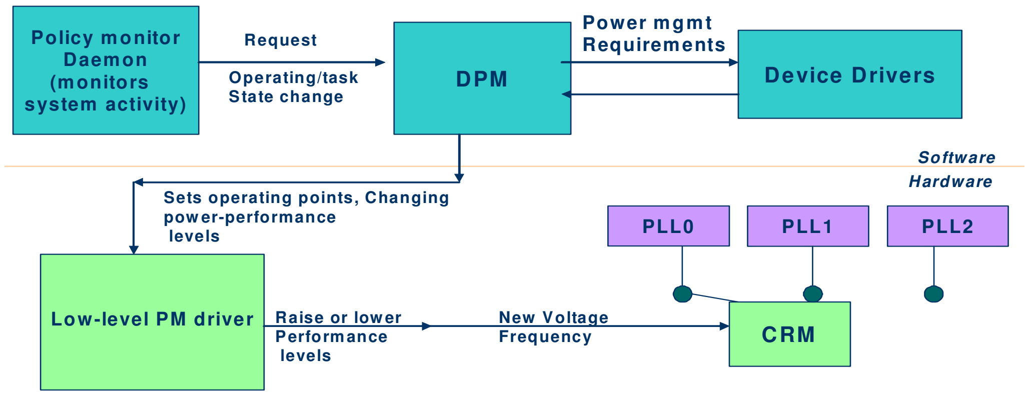

Policy monitor Daemon (monitors system activity)
Request
Operating/task State change
DPM
Power mgmt Requirements
Device Drivers
Software
Hardware
Sets operating points, Changing power-performance levels
PLL0
PLL1
PLL2
Low-level PM driver
Raise or lower Performance levels
New Voltage Frequency
CRM

Figure 2-5. DPM Architecture Block Diagram

2.3.13.6 Low-Level Power Management Driver

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

2.3.13.7 Dynamic Voltage and Frequency Scaling (DVFS) Driver

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

2.3.13.8 Dynamic Process and Temperature Compensation (DPTC) Driver

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

2.4 Boot Loaders

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

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

NOTE

Not all boot loaders are supported on all boards.

2.4.1 Functions of Boot Loaders

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

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

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

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

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

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

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

The following boot loaders are provided in the BSP:

- RedBoot

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

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

2.4.2 RedBoot

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

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

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

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-mx35 for MX35 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
mx35.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
`mx35.h`	Header files for the IO module physical addresses
irq.c	Common functions for various boards

3.6.5 EDIO Programming Interface

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

3.7 SPBA Bus Arbiter

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

3.7.1 SPBA Hardware Operation

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

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

3.7.2 SPBA Software Operation

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

3.7.3 SPBA Features

This SPBA implementation supports the following features:

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

3.7.4 SPBA Source Code Structure

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

Table 3-7. SPBA Files

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

3.7.5 SPBA Programming Interface

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

Chapter 4
Smart Direct Memory Access (SDMA) API

4.1 Overview

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

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

4.2 Hardware Operation

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

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

4.3 Software Operation

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

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

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

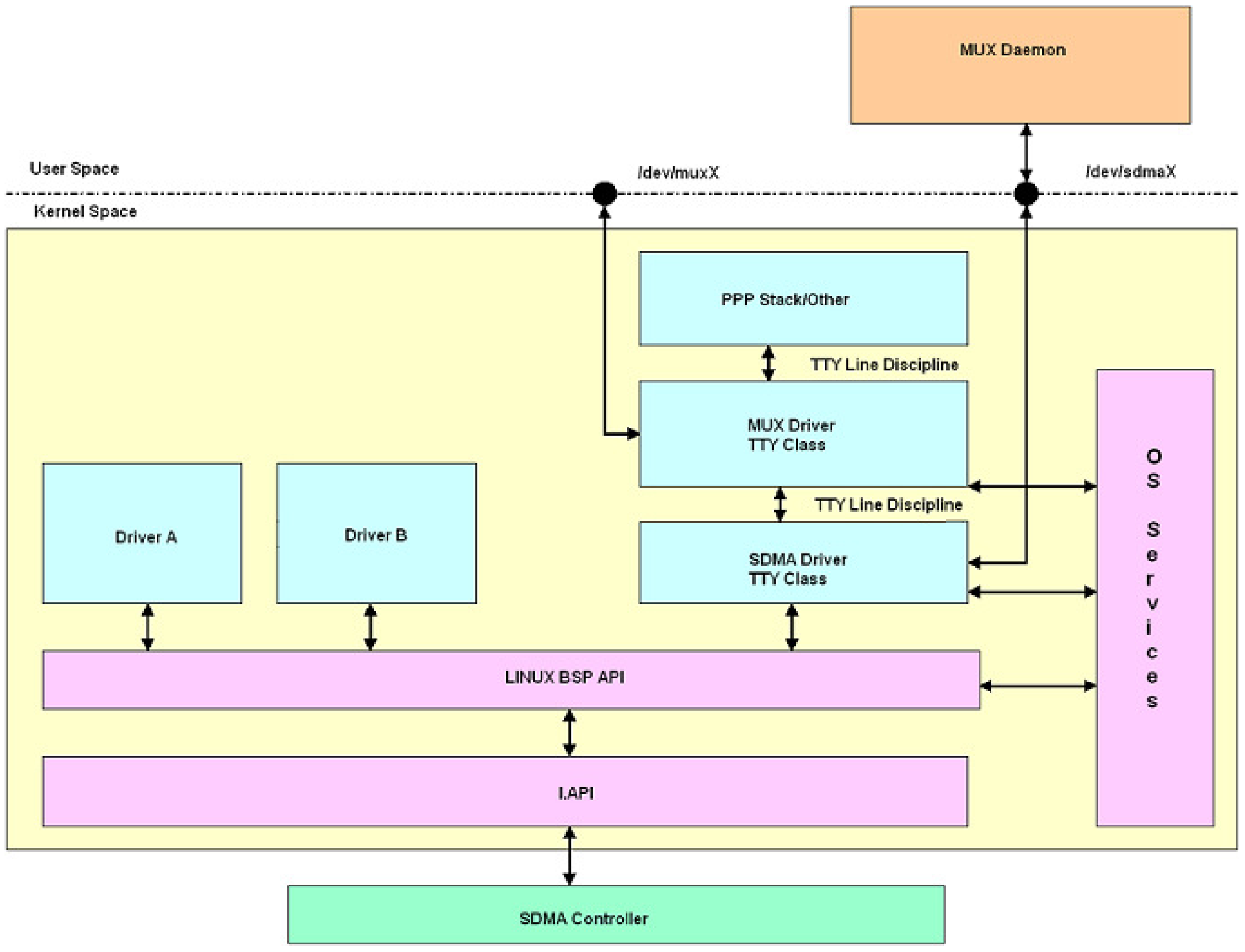


Figure 4-1. SDMA Block Diagram

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

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

Table 4-1. SDMA Channel Usage

Driver Name	Number of SDMA Channels	SDMA Channel Used
SDMA CMD	1	Static Channel allocation—uses SDMA channels 0
SSI	2 per device	Dynamic channel allocation
UART	2 per device	Dynamic channel allocation
SPDIF	2 per device	Dynamic channel allocation
ASRC	6 per device	Dynamic channel allocation
ESAI	2 per device	Dynamic channel allocation

4.4 Source Code Structure

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

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

Table 4-2. SDMA API Source Files

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

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

Table 4-3. SDMA Script Files

File	Description
sdma_script_code.h	SDMA RAM scriptsfor i.MX35 TO1
sdma_script_code_v2.h	SDMA RAM scripts for i.MX35 TO2

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 > MX3 Options > Use SDMA API.

By default, this option is Y.

System type > Freescale MXC implementations > MX35 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 1
MC9S08DZ60-PMIC Protocol Driver

This chapter describes the Linux MC9S08DZ60 device driver that provides the low-level read/write access to the MC9S08DZ60 hardware control registers and functions as an extension for system interrupts and GPIO pins. One key objective of the MC9S08DZ60 driver is to provide a complete API interface to all supported MC9S08DZ chips, despite differences in hardware design and implementation.

With a single API interface, a single application can be reused without any changes across all supported MC9S08DZ chips. Such an application, however, must either restrict itself to a core set of features supported by all MC9S08DZ chips, or detect at runtime which MC9S08DZ60 chip is installed before performing any MC9S08DZ60-specific operations.

This chapter describes the requirements, design, implementation, and client API that is provided for accessing hardware. The MC9S08DZ60 driver handles all low-level communications between many other Linux device drivers. The MC9S08DZ60 driver uses I^2C buses to communicate with the CPU.

NOTE

The MC9S08DZ60 driver is intended only for use with the CPU core and the Linux OS.

1.1 Key MC9S08DZ60 Driver Features and Capabilities

The driver typically provides hardware to support the following functions for Freescale i.MX-based platforms:

- Power supply control and power management support
- Real-time clock (RTC) support
- Event notification through the use of hardware interrupts
- GPIO control

1.1.1 MC9S08DZ60 Driver Register Access and Arbitration

The main purpose of the MC9S08DZ60 driver is to provide the necessary read/write access to the MC9S08DZ60 control registers using the I^2C bus interfaces to support all of the higher-level MC9S08DZ60 client drivers.

- Access to the control registers of the MC9S08DZ60 is implemented via I^2C bus interface. MC9S08DZ60 is acting as I^2C client device. The lower part of the protocol driver registers the two chips (MC9S08DZ60 and MC13892) as two I^2C clients respectively and provides APIs for accessing the actual registers of the two chips. The upper part of the protocol driver implements a universal pseudo register space and maps these registers to the actual registers of MC9S08DZ60.

Thus the protocol driver as a whole can provide a set of common APIs for accessing the MC9S08DZ60 module.

- Arbitration of accessing MC9S08DZ60 registers is implemented inside the I^2C host driver.

1.1.2 Interrupt Notification

All interrupts inside (such as keypad, touch screen, RTC) and outside (such as GPS, on/off Key press) are integrated into one interrupt and sent to the i.MX device. Every interrupt source has an enable bit (default disabled). When the interrupt is enabled, and an interrupt occurs, a high level is generated on GPIO, thus the i.MX device detects the interrupt and reads the interrupt flags by I^2C. After the interrupt is read, the flag is cleared manually by I^2C access. After all the flags are cleared, the high level return to low level.

The interrupt notification system is implemented in a pseudo IRQ driver on the board level, which is based on the MC9S08DZ60 driver but not included in the driver.

1.2 Driver Requirements

The MC9S08DZ60 Driver module (also called the core driver in <ltib_dir>/rpm/BUILD/linux/drivers/mxc/mcu_pmic) is responsible for providing two types of services for all of the MC9S08DZ60 client driver components:

- Control Services. This is implemented by providing read/write APIs to get/set the status of GPIO pins of the MC9S08DZ60 hardware
- Event Notification Services. This is implemented through pseudo IRQ driver.

The MC9S08DZ60 driver is 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 MC9S08DZ60 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 MC9S08DZ60 Driver be properly loaded and fully operational.

1.3 Driver Software Operation

The MC9S08DZ60 driver controls the MC9S08DZ60 by reading and writing the MC9S08DZ60 hardware control registers. Both read and write access to the MC9S08DZ60 hardware control registers is done through the I^2C driver. Figure 1-1 shows the relationship between the MC9S08DZ60 driver and all of the other related device drivers in the system as well as the interaction between them.

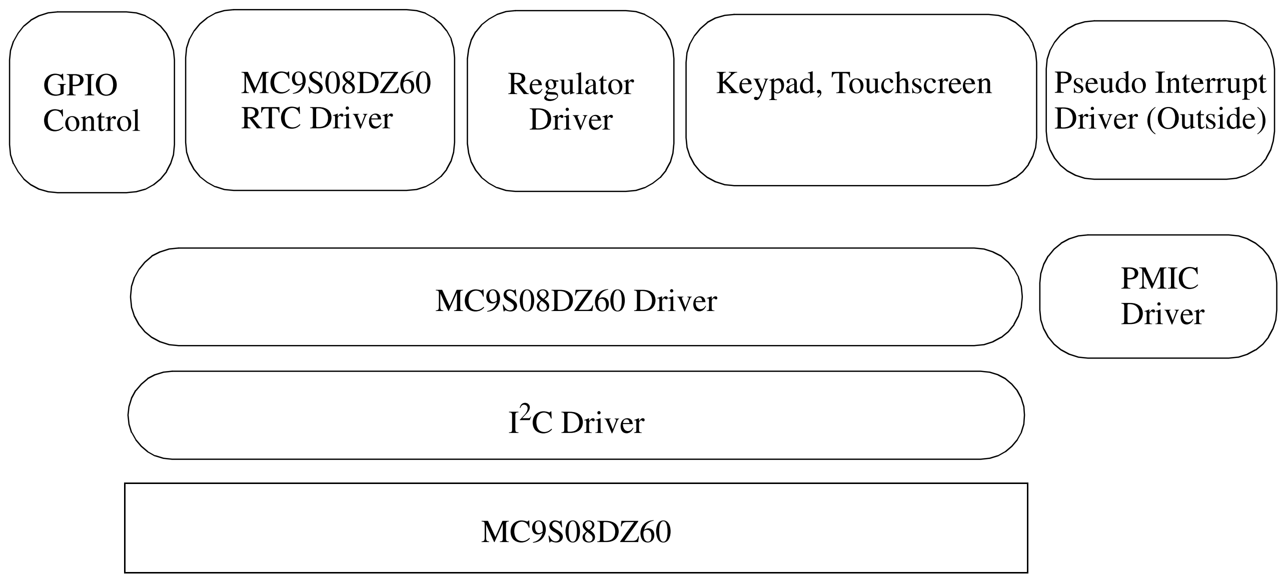


Figure 1-1. MC9S08DZ60 Device Driver

The hardware interrupt signal that can be generated by the MC9S08DZ60 is received and handled by the board level pseudo interrupt driver, that read and writes the MC9S08DZ60 registers to mask/unmask/judge/acknowledge events through the MC9S08DZ60 driver.

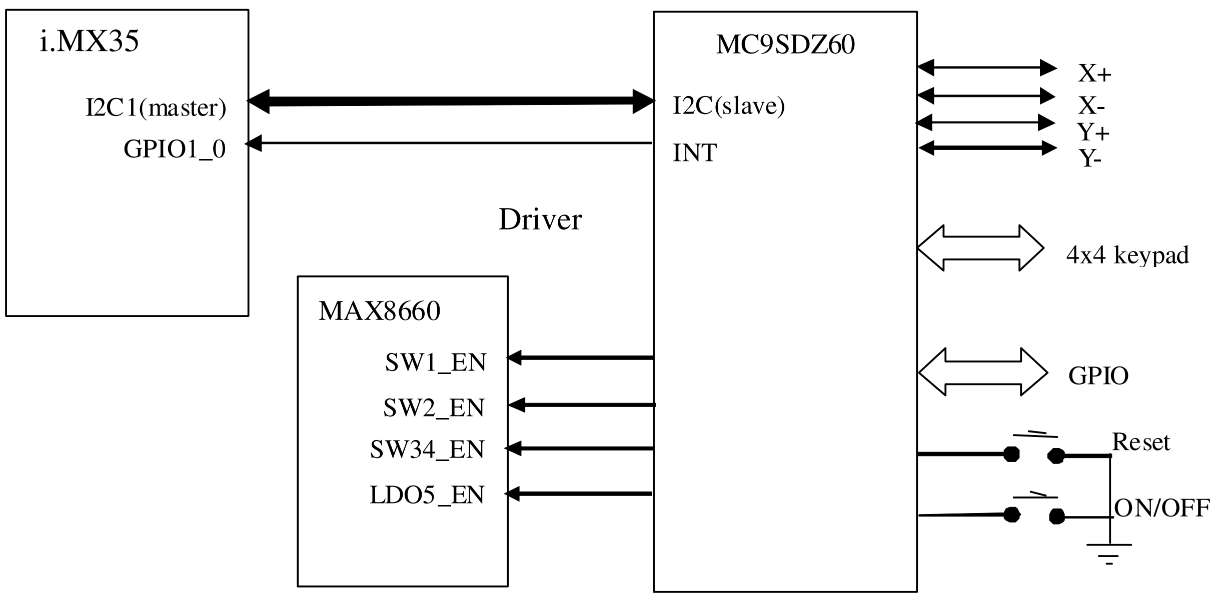


Figure 1-2. MC9S08DZ60 Connection With PMIC (i.MX35 3Stack v1.x)

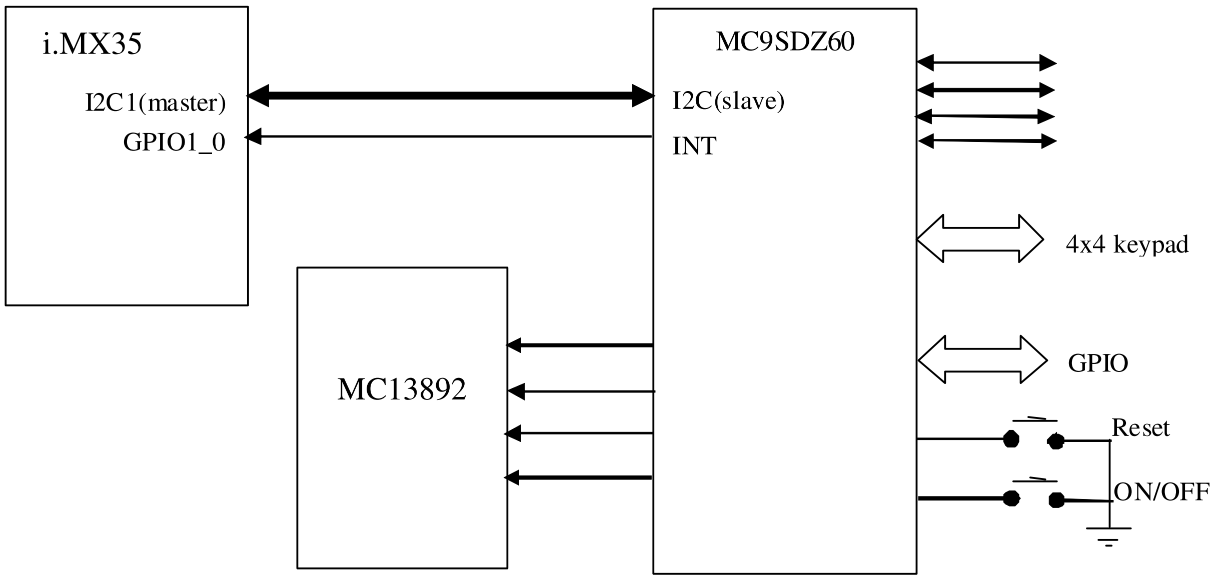


Figure 1-3. MC9S08DZ60 Connection With PMIC (i.MX35 3Stack v2.x)

1.4 Driver Implementation Details

This section describes implementation-specific details associated with the MC9S08DZ60 driver. The device driver source files can be consulted to fully understand the implementation of the MC9S08DZ60 driver. The I^{2}C driver documentation and sources can also be consulted if required.

1.4.1 Driver Initialization

The MC9S08DZ60 driver performs the following operations when it is first loaded/initialized:

- Registers MC9S08DZ60 as I^{2}C client.
- Initializes all driver-specific global variables.

1.4.2 Driver Unloading

The following operations are performed when unloading/deinitializing the MC9S08DZ60mc9s08dz60 driver:

- Unregisters MC9S08DZ60 and MAX8660 I^{2}C clients.

1.4.3 MC9S08DZ60 for i.MX35 3-Stack Version 1 and Version 2

Generally the MC9S08DZ60 is used for CPU interrupt source extension and GPIO extension. According to different board hardware designs, the interrupt source and GPIO pins may be assigned for different purposes.

- For the interrupt sources assignment, there are no changes from version 1 to version 2

- GPIO control pins are mainly used for peripheral enable/disable controls, reset controls, power on/off controls and mux controls for device selection. From board version 1 to version 2, the hardware design changed some GPIO pins assignment, which should be noticed by the drivers that use the MC9S08DZ60 APIs.

1.5 Driver Source Code Structure

The source files for the MC9S08DZ60 driver are available in the drivers directory, `<ltib_dir>/rpm/BUILD/linux/drivers/mxc/mcu_pmic/`.

Table 1-1 provides a brief description of each of the device driver source files.

Table 1-1. MC9S08DZ60 Driver Sources File List

File	Description
`mcu_pmic_core.c`	Main function of the module, register access function
`mc9s08dz60.h`	Declaration of all the functions whose implementation differs from MC9S08DZ60 chip to MC9S08DZ60 chip
`mc9s08dz60.c`	Low level driver for mc9s08dz60
`mcu_pmic_gpio.c`	mc9s08dz60 GPIO driver

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 1.6, “Driver Configuration”) and a `Makefile` that is used during the Linux kernel image build process.

1.6 Driver Configuration

The MC9S08DZ60 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 MC9S08DZ60 driver.

The following Linux kernel configuration options are provided for the MC9S08DZ60 driver. To enter the configuration screens, use the following command. You should be located in the ltib directory.

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

In the screen select Configure Kernel, exit, and a new screen appears. Choose the following to have MC9S08DZ60 driver support:

Device Drivers > MXC Support Drivers > MC9SDZ60 PMIC

By default, this option is Y for the i.MX35.

Chapter 6
MC13892 Regulator Driver

The MC13892 regulator driver provides the low-level control of the power supply regulators, selection of voltage levels, and enabling/disabling of regulators. This device driver makes use of the PMIC protocol driver (Chapter 1, "MC9S08DZ60-PMIC Protocol Driver") to access the PMIC hardware control registers.

6.1 Hardware Operation

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

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

6.2 Driver Features

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

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

6.3 Software Operation

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

Some of the PMIC power management operations depend on the system design and configuration. For example, if the system is powered by a power source other than the PMIC, then turning off or adjusting the PMIC voltage regulators has no effect. Conversely, if the system is powered by the PMIC, then any changes that use the power management driver and the regulator client driver can affect the operation or stability of the entire system.

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 provide status information to user space applications through a sysfs interface. The intention is to allow systems to dynamically control regulator output to save power and prolong battery life. This applies to both voltage regulators (where voltage output is controllable) and current sinks (where current output is controllable).

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

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

- regulator_get—lookup and obtain a reference to a regulator
  ```
  struct regulator *regulator_get(struct device *dev, const char *id);
  ```
- regulator_put—free the regulator source
  ```
  void regulator_put(struct regulator *regulator, struct device *dev);
  ```
- regulator_enable—enable regulator output
  ```
  int regulator_enable(struct regulator *regulator);
  ```
- regulator_disable—disable regulator output
  ```
  int regulator_disable(struct regulator *regulator);
  ```
- regulator_is_enabled—is the regulator output enabled
  ```
  int regulator_is_enabled(struct regulator *regulator);
  ```
- regulator_set_voltage—set regulator output voltage
  ```
  int regulator_set_voltage(struct regulator *regulator, int uV);
  ```
- regulator_get_voltage—get regulator output voltage
  ```
  int regulator_get_voltage(struct regulator *regulator);
  ```

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

6.5 Driver Architecture

Figure 6-1 shows the basic architecture of the MC13892 regulator driver.

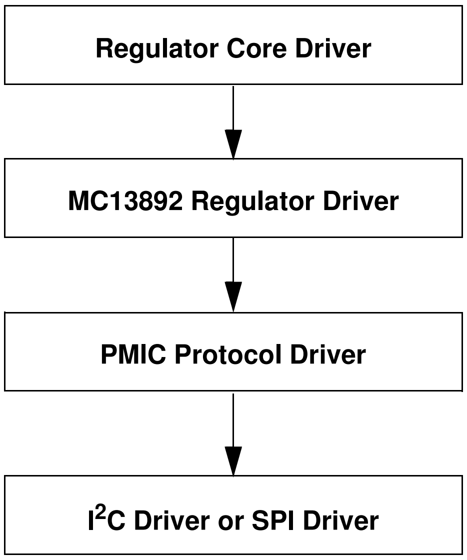


Figure 6-1. MC13892 Regulator Driver Architecture

6.6 Driver Interface Details

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

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

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

6.7 Source Code Structure

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

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

Table 6-1. MC13892 Power Management Driver Files

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

6.8 Menu Configuration Options

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

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

Chapter 7
MC13892 Digitizer Driver

This chapter describes the Linux PMIC Digitizer Driver that provides low-level access to the PMIC analog-to-digital converters (ADC). This capability includes taking measurements of the X-Y coordinates and contact pressure from an attached touch panel. This device driver uses the PMIC protocol driver (see Chapter 1, "MC9S08DZ60-PMIC Protocol Driver") to access the PMIC hardware control registers that are associated with the ADC.

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

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

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

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

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

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

7.1 Driver Features

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

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

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

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

7.2 Software Operation

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

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

The PMIC ADC conversion can take a significant amount of time. The delay between a start of conversion request and a conversion completed event may even be open ended, if the conversion is not started until the appropriate external trigger signal is received. Therefore, all ADC conversion requests must be placed in a wait queue until the conversion is complete. Once the ADC conversion has completed, the calling thread can be removed from the wait queue and reawakened.

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

7.3 Source Code Structure

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

```
<ltib_dir>/rpm/BUILD/linux/drivers/mxc/pmic/mc13892/pmic_adc.c
<ltib_dir>/rpm/BUILD/linux/include/linux/pmic_adc.h
<ltib_dir>/rpm/BUILD/linux/drivers/input/touchscreen/mxc_ts.c
```

Table 7-1. MC13892 Digitizer Driver Files

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

7.4 Menu Configuration Options

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

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

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

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

Chapter 1
MC13892 RTC Driver

The Linux MC13892 RTC driver provides access to the MC13892 RTC control circuits. This device driver makes use of the MC13892 protocol driver (see Chapter 1, "MC9S08DZ60-PMIC Protocol Driver") to access the MC13892 hardware control registers. The MC13892 device is used for real-time clock control and wait alarm events.

1.1 Driver Features

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

1.2 Software Operation

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

1.3 Driver Implementation Details

Configuring the MC13892 RTC driver includes the following parameters:

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

1.3.1 Driver Access and Control

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

1.4 Source Code Structure

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

Table 1-1. MC9S08DZ60 RTC Driver Files

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

1.5 Menu Configuration Options

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

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

Chapter 2
i.MX35 Low-level Power Management Driver

This chapter describes the design of the low-level power management drivers for the i.MX35 platform. This driver implements dynamic voltage and frequency scaling (DVFS) techniques and lower power modes.

2.1 Overview

DVFS is used to reduce active power consumption by scaling voltage and frequency accordingly to reduce MIPs. This is done when the system is in RUN mode to conserve power. Lower power modes, such as WAIT, DOZE, and STOP are implemented to save power. In all these cases, power consumption is achieved by reducing the frequency and increasing the severity of clock gating.

2.2 Hardware Operation

2.2.1 DVFS

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

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

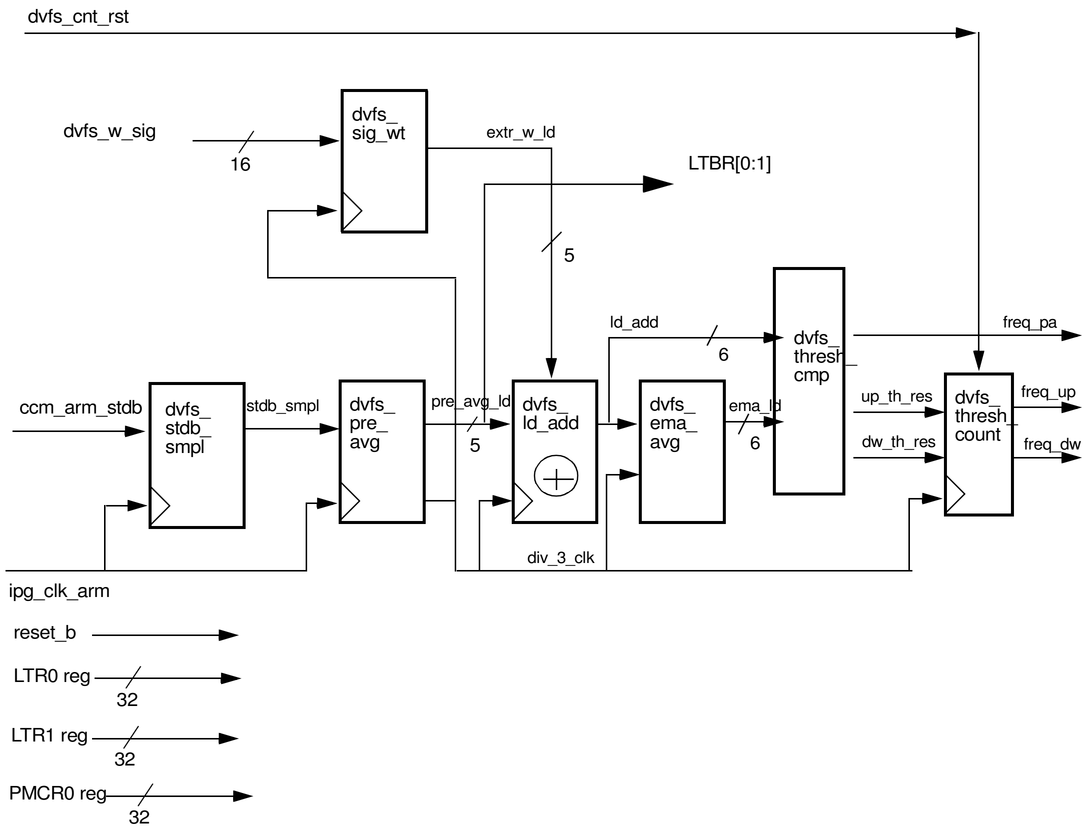


Figure 2-1. DVFS Load Tracking Module Block Diagram

2.2.2 Lower Power Mode

i.MX35 supports a versatile definition of power modes (see Table 2-1), including power and clock domains status and applied power techniques.

Table 2-1. Low Power Modes

Mode	Core	ARM11P MAX	Modules	MPLL	PPLL	Osc24M	Osc Audio	Perclk
RUN	Active	Active	Active, Idle or Disable	On	On/off	On	On/off	Some On
WAIT	Disable	Active	Active, Idle or Disable	On	On/off	On	On/off	Some On
DOZE	Disable	Disable	Active, Idle or Disable	On	On/off	On	On/off	Some On
STOP	Disable	Disable	Disable	Off	Off	On/Off	Off	Off

If logic core supply is configured to lower down from 1.1 V to 1.0 V in stop mode, this specific stop mode is called state-retention mode. If Osc24M is powered off in stop mode and all clocks are off including CKIL, this special stop mode is called static stop mode.

2.3 Software Operations

2.3.1 DVFS

The DVFS Linux driver is designed to monitor and control the DVFS hardware module, and perform the transitions between device working points. It provides SYSFS interface to the user space to enable and disable DVFS module. If DVFS is enabled, the driver sets DVFS load tracking registers according to different working points. When ARM DVFS interrupt is received, the driver performs the following operations:

- Config dvfs_start bit and mask interrupt. Put the work to change DVFS in global work queue.
- In work task, check FSVAI and DVSUP to determine whether frequency and voltage needs to be changed.
- Write 0 to DPVCR to disable DPTC voltage change.
- If frequency needs to be increased, increase voltage firstly. And then increase frequency.
- If frequency needs to be decreased, decrease frequency firstly. And then decrease voltage.
- Update DVSUP to new frequency/voltage level. And move to new working points.
- Unmask DVFS interrupt and config dvfs_update_finish bit.

If DVFS is disabled, the driver restores working point to previous status when entering DVFS. The i.MX35 DVFS driver only supports two working points (399 MHz and 533 MHz) for TO1 consumer path and TO2. This is because the voltages of all frequencies under 399 MHz required 1.2 V. The power consumption is increased if working with lower frequency and the same voltage.

2.3.2 Lower Power Modes

Following is the program flow to enter and exit low power mode:

1. Setup wakeup interrupt before entering lower power mode.
2. Program LPMD bits and program interrupt holdoff bits in CCM. For Stop mode, config STANDBY exit source according to hardware design.
3. Call `cpu_do_idle` to execute WFI pending instructions.
4. Generate wakeup interrupt and exit low power mode.

Currently the i.MX35 PM driver maps `stop` mode to support the bellow kernel power management states accordingly:

- `standby`: This state offers minimal power saving, while providing a very low-latency transition back to a working system.
- `mem` (suspend to RAM): This state offers significant power saving as everything in the system is put into a low-power state, except for memory, which is placed in self-refresh mode to retain its contents

PM driver also maps `Wait` mode for system idle function.

NOTE

Because the i.MX35 TO1 3-stack does not support external clock connected to CKIL, static mode is not currently supported. Although the VSTBY signal is connected to MCU-PMIC, MCU-PMIC does nothing after this signal is asserted. Therefore, VSTBY bit in CCM is programmed as “0” to disable state-retention mode. State-retention mode is supported after MCU firmware is updated to support VSTBY signal.

i.MX35 TO2 adds the support for VSTBY signal support. The core voltage in standby/mem state is reduced to 1.0 V.

2.4 Source Code Structure

Table 2-2 lists the DVFS and lower power mode driver source files. These files are available in the directory, `<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx35/`.

Table 2-2. Source Code for i.MX35 PM Drivers

File	Description
dvfs.c	Source file for DVFS driver
pm.c	Source file to support suspend operation
system.c	Source file to support lower power modes

2.5 Configuration

The following Linux kernel configurations are provided for power management drivers. To get to the power management 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_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 for the i.MX35 platform.
- `CONFIG_SUSPEND:` Build support for suspend. In menuconfig, this option is available under Power management options > Suspend to RAM and standby. By default, this option is Y for i.MX35 platform.

2.6 Programming Interface

mxc_cpu_lp_set is provided for low-power modes. This implements all the steps required to put the system under WAIT/DOZE/STOP mode.

Chapter 3
Image Processing Unit (IPU) Drivers

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

Typical logical channels include:

- CSI direct to memory
- CSI to viewfinder pre-processing to memory or ADC
- Memory to viewfinder pre-processing to memory or ADC
- Memory to viewfinder rotation to memory
- CSI to encoder pre-processing to memory
- Memory to encoder pre-processing to memory
- Memory to encoder rotation to memory
- Memory to post-processing to memory or ADC
- Memory to post-processing rotation to memory
- Memory to post filter (Y buffer) to memory
- Memory to post filter (U buffer) to memory
- Memory to post filter (V buffer) to memory
- Memory to synchronous frame buffer background
- Memory to synchronous frame buffer foreground
- Memory to synchronous frame buffer mask
- Memory to ADC system channel 1
- Memory to ADC System channel 2

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

- Synchronous frame buffer functions
  - Panel interface initialization
  - Set foreground and background plane positions

- Set local/global alpha and color key
- Set backlight level

- CSI functions
  - Sensor interface initialization
  - Set sensor clock
  - Set capture size
- ADC Functions
  - Panel interface initialization
  - Send commands to panel

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

3.1 Hardware Operation

The detailed hardware operation of the IPU is discussed in the *MCIMX35 Multimedia Applications Processors Reference Manual*, (*MCIMX35RM*). Figure 3-1 shows the IPU hardware modules.

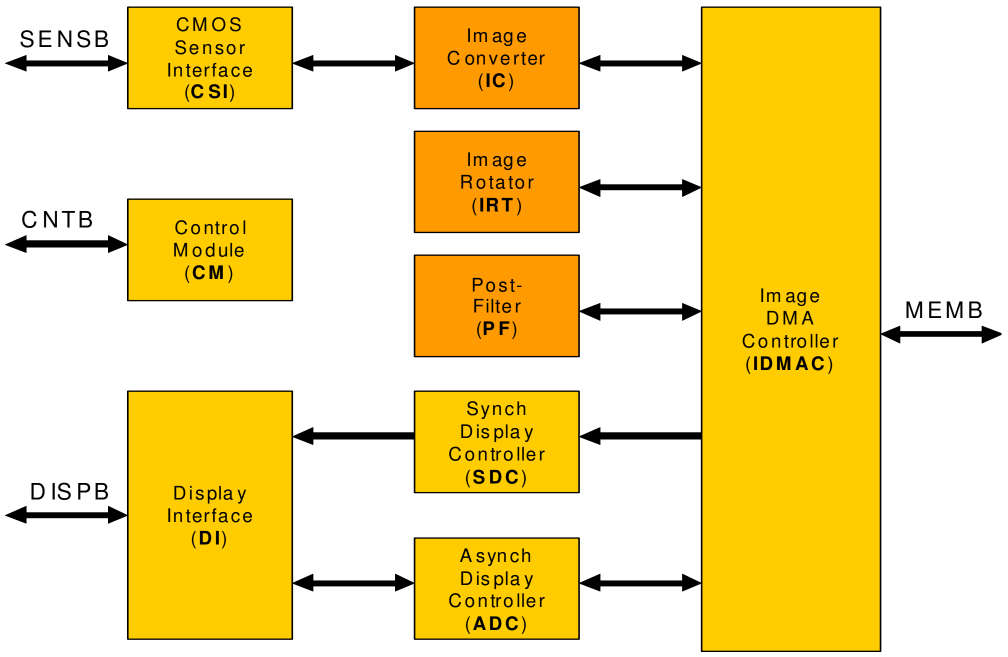

SENSB
CMOS Sensor Interface (CSI)
Image Converter (IC)
Image Rotator (IRT)
CNTB
Control Module (CM)
Post-Filter (PF)
Image DMA Controller (IDMAC)
MEMB
Synch Display Controller (SDC)
DISPB
Display Interface (DI)
Asynch Display Controller (ADC)

Figure 3-1. IPUv1 hardware module overview

3.2 Software Operation

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

- Synchronous frame buffer driver

- Asynchronous frame buffer driver
- Display Interface (DI)
- Image DMA Controller (IDMAC)
- CMOS Sensor Interface (CSI)
- Image Converter (IC)
- Post-Filter (PF)

Figure 3-2 shows the interaction between the different graphics/video drivers and the IPU.

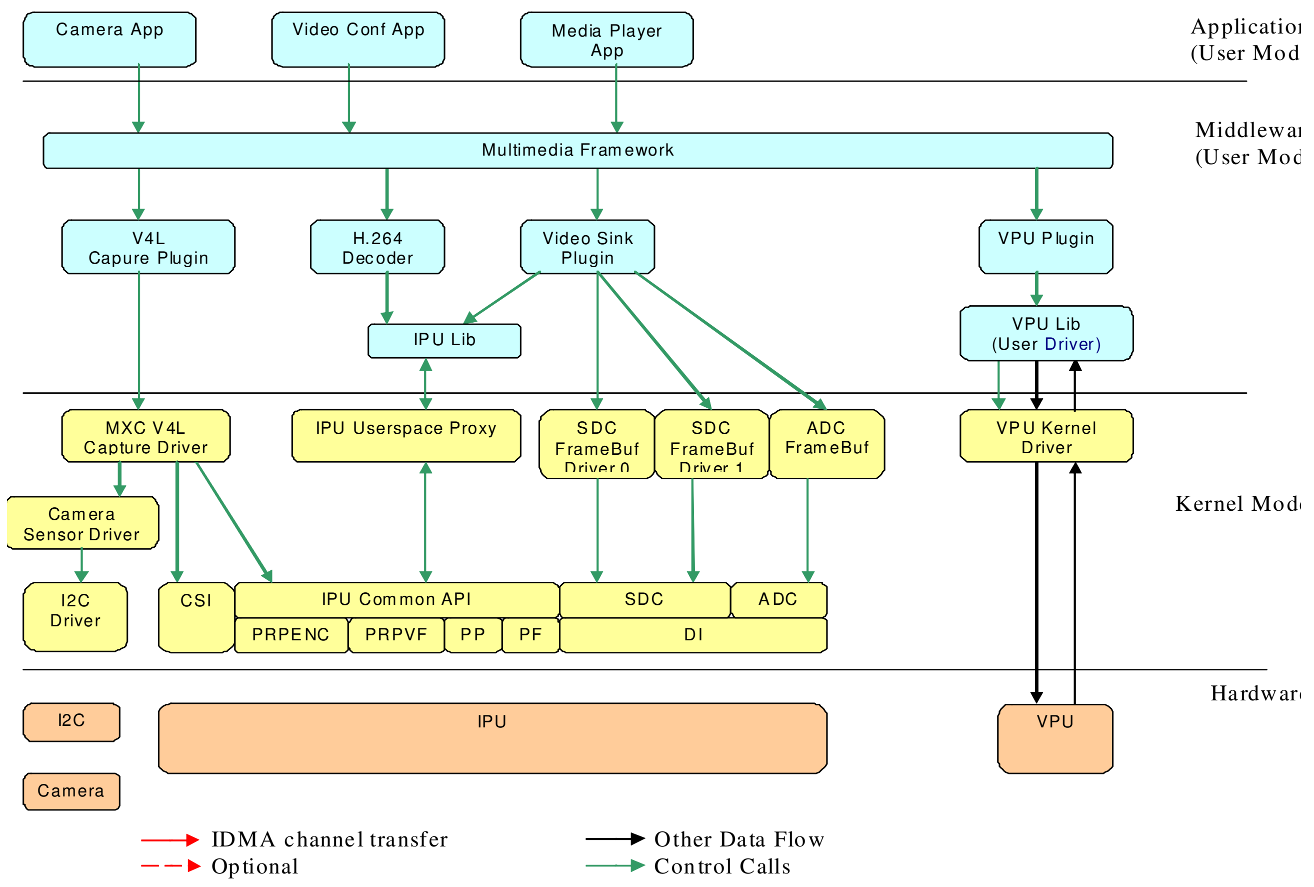


Figure 3-2. Graphics/Video Drivers Software Interaction

The IPU drivers are sub-divided as follows:

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

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

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

  `<ltib_dir>/rpm/BUILD/linux/drivers/media/video` directory of the Linux kernel.

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

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

3.2.1 IPU Frame Buffer Drivers Overview

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

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

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

3.2.1.1 IPU Frame Buffer Hardware Operation

The frame buffer interacts with the IPU hardware driver module.

3.2.1.2 IPU Frame Buffer Software Operation

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

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

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

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

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

3.2.1.3 Synchronous Frame Buffer Driver

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

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

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

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

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

The following features are supported:

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

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

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

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

3.2.1.4 Asynchronous Display Controller (ADC) Frame Buffer Driver

The asynchronous display controller frame buffer screen driver implements a Linux standard frame buffer driver API for asynchronous or smart LCD panels. The asynchronous frame buffer screen driver is the top level kernel video driver that interacts with the kernel and user level applications. This is enabled by selecting the corresponding frame buffer option under the graphics parameters in the kernel configuration. To supplement the Frame buffer driver, the kernel builder may also include support for fonts and a startup logo. This depends on the VT console for switching from serial to graphics mode.

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

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

The following features are supported:

- Configurable RGB 16, 24 or 32 bits per pixel frame buffer
- Palette/color conversion management
- Power management
- LCD power off/on

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

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

3.2.2 IPU Backlight Driver

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

3.2.3 MPEG4/H.264 Post Filter Driver

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

- Support for MPEG4 dering and/or deblock
- Support for H264 deblock
- Support for intra-frame pause and resume (H.264 only)
- Synchronous and asynchronous operation
- Support for driver-allocated or user-allocated buffers

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

3.3 Source Code Structure

Table 3-1 lists the source files associated with the IPU, Sensor, V4L2 and Panel drivers. These files are available in the following directories:

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

Table 3-1. IPU Driver Files

File	Description
ipu_adc.c	ADC configuration driver
ipu_common.c	Configuration functions for ADC and SDC
ipu_csi.c	CMOS sensor interface functions
ipu_ic.c	IPU library functions
ipu_sdc.c	SDC configuration driver
ipu_device.c	IPU driver device interface and fops functions
/pf/mxc_pf.c	Post filtering driver
mxcfb.c	Driver for synchronous frame buffer
mxcfb_epson.c	Driver for asynchronous frame buffer
mxcfb_epson_vga.c	Driver for synchronous framebuffer for VGA
mxcfb_claa_wvga.c	Driver for synchronous frame buffer for WVGA
mxcfb_modedb.c	Parameter settings for Framebuffer devices
mxc_ipu_bl.c	IPU backlight control driver

Table 3-2 lists the global header files associated with the IPU and Panel drivers. These files are available in the following directories:

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

Table 3-2. IPU Global Header Files

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

3.4 Menu Configuration Options

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

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

  Device Drivers > MXC support drivers > Image Processing Unit Driver

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

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

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

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

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

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

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

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

  Use case driver for dumb sensor or

  CSI > IC(PRP VF) > MEM

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

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

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

  Use case driver for the rotation

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

  or for smart sensors

CSI > IC > ADC.

In `menuconfig`, this option is available under: (Asynchronous Panels must be selected under Graphics support for PP VF ADC library to appear)

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

By default, this option is M for all.

- CONFIG_MXC_IPU_PRP_ENC—Option for the IPU:

  Use case driver for dumb sensors

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

  or for smart sensors

  CSI > IC(PRP ENC) > MEM.

  In menuconfig, this option is available under:

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

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

- CONFIG_MXC_IPU_CSI_ENC—Option for the IPU:

  Use case driver for dumb sensors

  CSI > MEM

  In `menuconfig`, this option is available under:

  Device Drivers > Multimedia Devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > IPU CSI Encoder library

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

- CONFIG_MXC_IPU_PF—This is configuration option for MXC MPEG4/H.264 Post Filter Driver. This option is dependent on the MXC_IPU option. In `menuconfig`, this option is available under:

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

  By default, this option is Y for all.

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

  VIDEO_DEV && MXC_IPU && MXC_IPU_PRP_VF_SDC && MXC_IPU_PRP_ENC

  In menuconfig, this option is available under:

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

  By default, this option is M for all.

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

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

  By default, this option is Y for all.

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

  Device Drivers > Graphics support > Support for frame buffer devices

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

  Device Drivers > Graphics support > MXC Framebuffer support

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

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

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

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

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

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

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > FS453 TV Out Encoder
- CONFIG_FB_MXC_ASYNC_PANEL—This configuration option selects the asynchronous panel framebuffer. This option is dependent on CONFIG_FB_MXC option. In `menuconfig`, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Asynchronous Panels

  By default, this option is N for all architectures.
- CONFIG_FB_MXC_EPSON_PANEL—This configuration option selects the Epson panel. This option is dependent on CONFIG_FB_MXC_ASYNC_PANEL option. In `menuconfig`, this option is available under

Device Drivers > Graphics support > MXC Framebuffer support > Asynchronous Panels > Asynchronous Panel Type > Epson 176x220 Panel

By default this option is N for all architectures.

3.5 Programming Interface

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

Chapter 4
Video for Linux Two (V4L2) Driver

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

The features supported by the V4L2 driver are as follows:

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

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

4.1 V4L2 Capture Device

The V4L2 capture device includes two interfaces:

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

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

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

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

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

4.1.1 V4L2 Capture IOCTLs

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

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

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

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

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

Figure 4-1 shows a block diagram of V4L2 Capture API interaction.

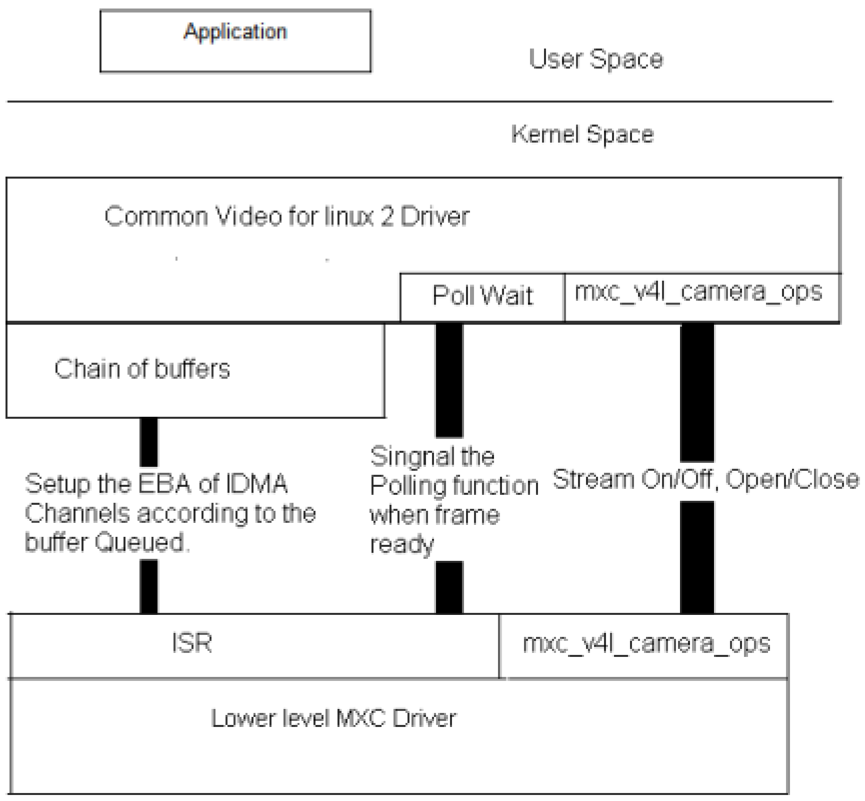

Application
User Space
Kernel Space
Common Video for linux 2 Driver
Poll Wait
mxc_v4l_camera_ops
Chain of buffers
Setup the EBA of IDMA Channels according to the buffer Queued.
Singnal the Polling function when frame ready
Stream On/Off, Open/Close
ISR
mxc_v4l_camera_ops
Lower level MXC Driver

Figure 4-1. Video4Linux2 Capture API Interaction

4.1.2 Use of the V4L2 Capture APIs

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

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

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

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

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

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

4.2 V4L2 Output Device

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

4.2.1 V4L2 Output IOCTLs

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

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

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

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

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

4.2.2 Use of the V4L2 Output APIs

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

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

4.3 Source Code Structure

Table 4-1 lists the source and header files associated with the V4L2 drivers. These files are available in the following directory:

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

Table 4-1. V2L2 Driver Files

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

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

4.4 Menu Configuration Options

The Linux kernel configuration options are provided in the chapter on the IPU module. See Section 3.4, "Menu Configuration Options."

4.5 V4L2 Programming Interface

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

Chapter 5
TV Decoder (TV-In) Driver

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

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

5.1 Hardware Operation

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

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

5.2 Software Operation

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

5.3 Source Code Structure Configuration

Table 5-1 describes the source files associated with the TVIN driver, which are available in the directory `<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture`.

Table 5-1. TV-In Driver Source File

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

5.4 Linux Menu Configuration Options

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

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

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

NOTE

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

Chapter 6
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.

6.1 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.MX35 Multimedia Applications Processor Reference Manual* for more information on CSI and IPU (IPU is not supported on all platforms).

6.2 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 stream mode
- Capture still mode
- Rotation
- Resize

The supported picture formats are:

- RGB565
- RGB24
- BGR24
- RGB32
- BGR32

- YUV422P
- UYVY
- YUV420

6.3 Source Code Structure

Table 6-1 shows the camera driver source files available in the directory
`<ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/capture.`

Table 6-1. Camera Driver Files

File	Description
ov2640.c	Camera driver implementation

6.4 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 7
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.

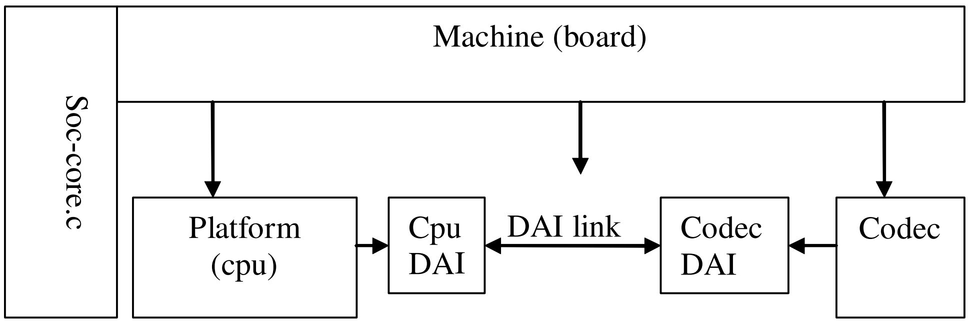


Figure 7-1. ALSA SoC Software Architecture

The ALSA SoC driver has the following components as seen in Figure 7-1:

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

7.1 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 Bluetooth codec is only supported on the i.MX35 platform.

7.1.1 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

7.1.2 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

7.1.3 Bluetooth Codec Features

- Supported sample rate for Playback/Capture: 8 KHz
- Supported channels:
  - Playback: supports two channels.
  - Capture: supports two channels.
- Supported audio formats:
  - Playback: SNDRV_PCM_FMTBIT_S16_LE

— Capture: SNDRV_PCM_FMTBIT_S16_LE

7.1.4 Sound Card Information

The registered sound card information can be listed as follows by the command `aplay -l` and `arecord -l`. For example, the stereo sound card is registered as card 0, the 5.1 sound card is registered as card 1, and the Bluetooth card is registered as card 2.

```
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
      card 2: bluetooth [imx-3stack], device 0: bluetooth bluetooth-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
      card 2: bluetooth [imx-3stack], device 0: bluetooth bluetooth-PCM-0 []
        Subdevices: 1/1
        Subdevice #0: subdevice #0
```

7.2 ASoC Driver Source Architecture

As illustrated in Figure 7-2, `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 Bluetooth 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. `bluetooth.c` registers the Bluetooth codec and Bluetooth DAI drivers. The direct hardware operations on the codec are in `bluetooth.c`. `imx-3stack-bt.c` is the machine layer code which creates the driver device and registers the sound card.

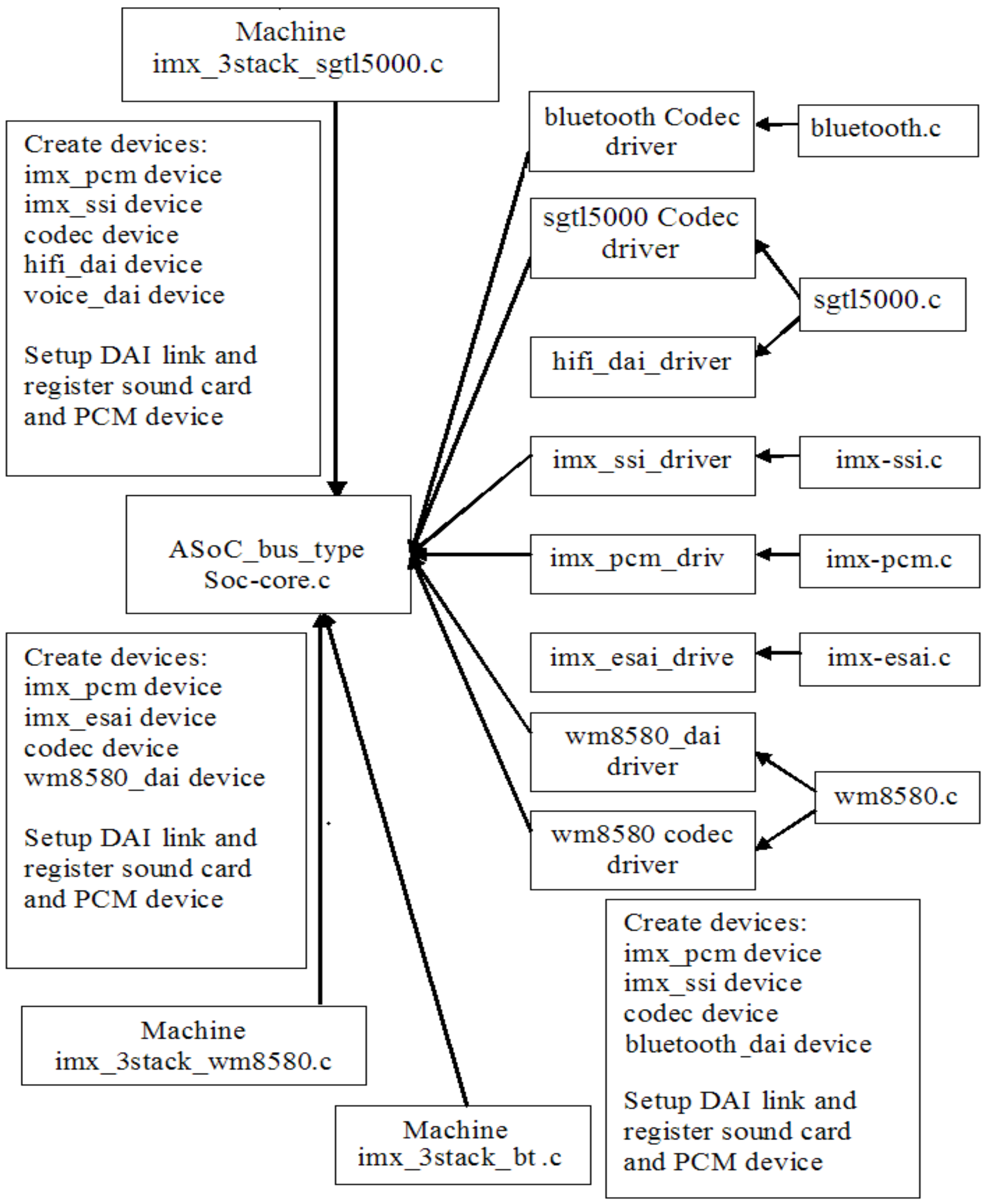


Figure 7-2. ALSA SoC Source File Relationship

Table 7-1 shows the stereo codec SoC driver source files. These files are under the `<ltib_dir>/rpm/BUILD/linux/sound/soc` directory.

Table 7-1. 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 7-1. 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 7-2 shows the 5.1 codec SoC driver source files. These files are also under the `<ltib_dir>/rpm/BUILD/linux/sound/soc` directory.

Table 7-2. 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 7-3 lists the Bluetooth codec SoC driver source files. These files are under the `<ltib_dir>/rpm/BUILD/linux/sound/soc` directory.

Table 7-3. Bluetooth Codec SoC Driver Source Files

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

7.3 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 - BLUETOOTH

7.4 Hardware Operation

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

7.4.1 Stereo Audio Codec

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

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

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

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

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

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

7.4.2 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.

7.4.3 Bluetooth Codec

The Bluetooth codec is a virtual codec, it only has a PCM interface connected to the Bluetooth device. The audio data is transferred from the user data buffer to or 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 as it provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate.

7.5 Software Operation

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

7.5.1 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

7.5.2 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 8
The Sony/Philips Digital Interface (S/PDIF) Rx/Tx Driver

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

8.1 S/PDIF Overview

Figure 8-1 shows the block diagram of the S/PDIF interface.

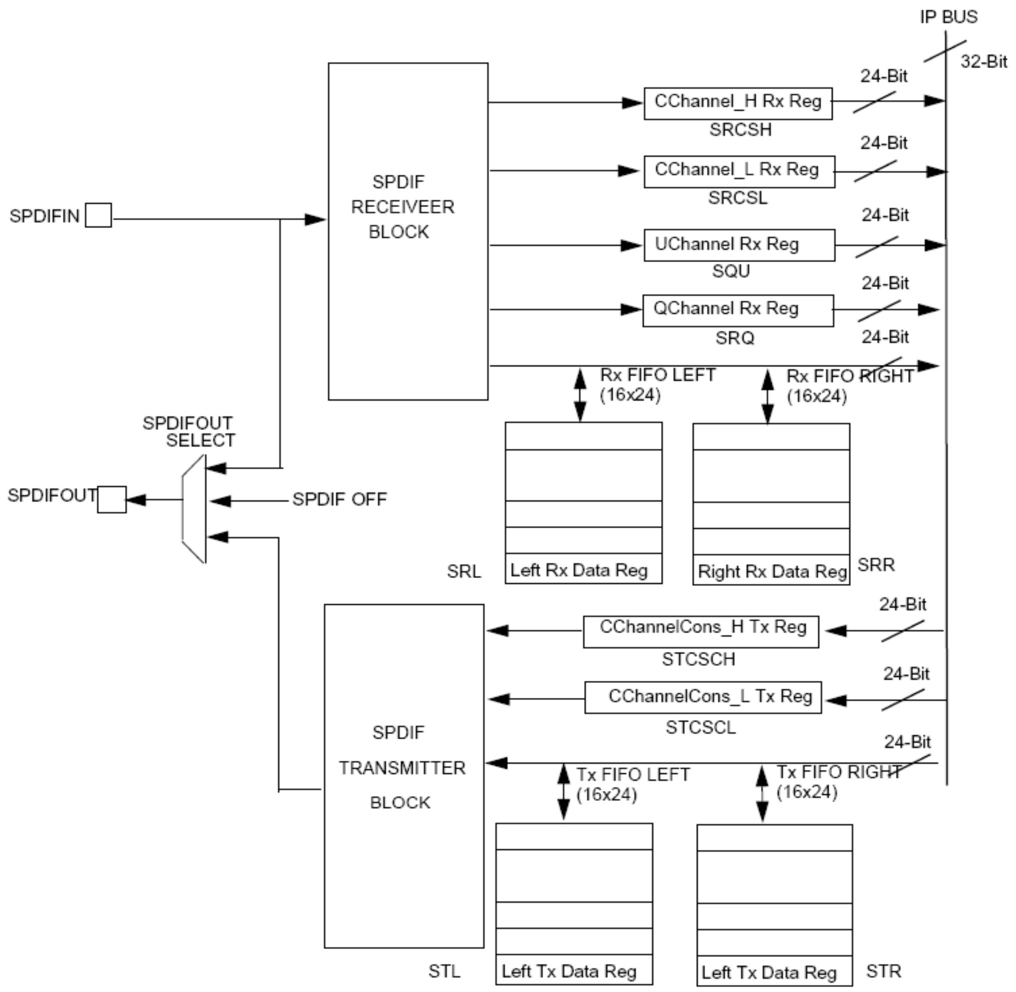


Figure 8-1. S/PDIF Transceiver Data Interface Block Diagram

8.1.1 Hardware Overview

The S/PDIF is composed of two parts:

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

In the S/PDIF transmitter, the IEC958 biphase bit stream is generated on both edges of the S/PDIF Transmit clock. The S/PDIF Transmit clock is generated by the S/PDIF internal clock generate module and the sources are from outside of the S/PDIF block. For the S/PDIF receiver, it can recover the S/PDIF Rx clock. Figure 8-1 shows the clock structure of the S/PDIF transceiver. i.MX35 supports the S/PDIF transceiver.

8.1.2 Software Overview

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

8.2 S/PDIF Tx Driver

The S/PDIF Tx driver supports the following features:

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

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

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

  ```
  #modprobe snd-spdif
  ```

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

  — Get card ID and name

    an example on i.MX35 3-Stack platform:

    ```
    #cat /proc/asound/cards
    ```

```
0 [imx3stack      ]: imx_3stack - imx_3stack
imx_3stack (ak4647)
1 [imx3stack_1    ]: imx_3stack - imx_3stack
imx_3stack (wm8580)
2 [TXRX           ]: MXC_SPDIF - MXC SPDIF TX/RX
MXC Freescale with SPDIF
```

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

— Get Playback PCM device info

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

- Software operation

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

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

NOTE

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

8.2.1 Driver Design

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

8.2.2 Provided User Interface

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

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

8.3 S/PDIF Rx Driver

The S/PDIF Rx driver supports the following features:

- 16, 32, 44.1, 48, 64 and 96 KHz receiving sample rate
- Signed 24-bit little endian sample format. Due to S/PDIF SDMA feature, each channel bit length in PCM recorded frame is 32 bits, and only the 24 LSBs are valid

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

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

  ```
  #modprobe snd-spdif
  ```

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

  — Get Card ID and name

  For example:

  ```
  #cat /proc/asound/cards
  0 [imx3stack      ]: imx_3stack - imx_3stack
            imx_3stack (ak4647)
  1 [imx3stack_1    ]: imx_3stack - imx_3stack
  imx_3stack (wm8580)
  2 [TXRX           ]: MXC_SPDIF - MXC SPDIF TX/RX
      MXC Freescale with SPDIF
  ```

  For example:

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

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

  — Get capture PCM device info

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

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

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

NOTE

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

8.3.1 Driver Design

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

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

For user bits reception, there are two modes for User Channel reception: CD and non-CD. The mode is determined by the USyncMode (bit 1 of CDText_Control register). User can call the sound control interface to set the mode (see Figure 8-1), but no matter what the mode is, the driver handles the user bits in the same way. For the S/PDIF Rx, the hardware block copies the Q bits from the user bits to the QChannel registers and puts the user bits in UChannel registers. The driver allocates two queue buffers for both U bits and Q bits. The U bits queue buffer is 96×2 bytes in size, the Q bits queue buffer is 12×2 bytes in size, and queue buffers are filled in the U/Q Full, Err and Sync interrupt handlers. This means that the user can get the previous ready U/Q bits while S/PDIF driver is reading new U/Q bits.

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

8.3.2 Provided User Interface

The S/PDIF Rx driver provides interfaces for user application as shown in Table 8-1.

Table 8-1. S/PDIF Rx Driver Interfaces

Interface	Type	Mode[1]	Parameter	Comment
Common PCM	PCM	—	—	PCM open/close prepare/trigger hw_params/sw_params
Rx Sample Rate	Sound Control[2]	r	Integer Range: [16000, 96000]	Get sample rate. It is not accurate due to DPLL frequency measure module. So the user application must do a correction to the get value.
USyncMode	Sound Control	rw	Boolean Value: 0 or 1	Set 1 for CD mode Set 0 for non-CD mode
Channel Status	Sound Control	r	struct snd_aes_iec958 Only status [6] array member is used	—
User bit	Sound Control	r	Byte array 96 bytes for U bits 12 bytes for Q bits	—
No good V bit	Sound Control	r	Boolean Value: 0 or 1	An interrupt is associated with the valid flag. (interrupt 16 - SPDIFValNoGood). This interrupt is set every time a frame is seen on the SPDIF interface with the valid bit set to invalid.

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

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

The user application can follow the program flow from Figure 8-2 to use the S/PDIF Rx driver. First the application opens the S/PDIF Rx PCM device, waits for the DPLL to lock the input bit stream, and gets the input sample rate. If the USyncMode needs to be set, set it before reading the U/Q bits. Next, set the hardware parameters, including channel number, format and capture sample rate which is obtained from the driver. Then call prepare and trigger to startup S/PDIF Rx stream read. Finally call the read function to get the data. During the reading process, applications can read the U/Q bits and channel status from the driver and valid the no good bit.

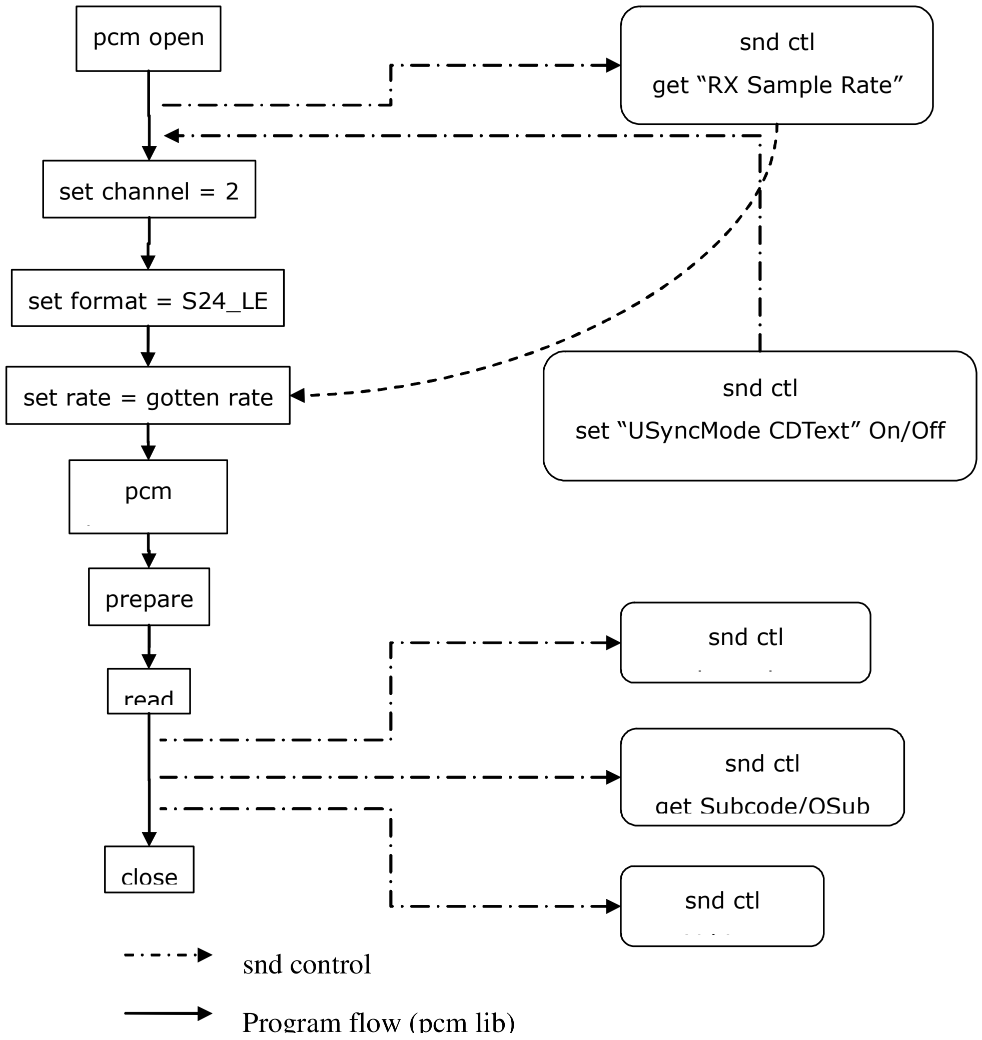


Figure 8-2. S/PDIF Rx Application Program Flow

8.4 Interrupts and Exceptions

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

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

8.5 Source Code Structure

Table 8-2 lists the source file that is available in the directory:

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

Table 8-2. S/PDIF Driver Files

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

8.6 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

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

  Device Drivers > Sound card support > Advanced Linux Sound Architecture

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

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

  By default, this option is M.

Chapter 9
NOR Flash Memory Technology Device (MTD) Driver

The NOR Flash Memory Technology Device (MTD) driver supports the Intel StrataFlash28F256L18 Flash and the Spansion S29WS256N Flash. Both devices are Common Flash Interface (CFI) compliant. For CFI-compliant devices, only a map driver is needed to provide the MTD mapping information. Other functionality, such as Flash read, write, and erase is provided by other parts of the Linux MTD subsystem. As NOR Flash is off-chip memory, and connected to the microprocessor as an external device, the map driver must be board specific.

By default, the NOR Flash MTD driver creates static MTD partitions to support either the Intel Strata Flash or the Spansion Flash on the board. If RedBoot partitions exist, they have higher priority than static partitions, and the MTD partitions can be created from the RedBoot partitions.

9.1 Hardware Operation

NOR Flash is non-volatile storage for embedded systems. Reading NOR Flash is identical to reading RAM. Software can run directly from NOR Flash, and this is called Execute-In-Place or XIP.

Writing to NOR Flash is very different from writing to conventional RAM. For NOR Flash, a sequence of steps is needed to initiate a write of data. A write almost always involves an erase cycle on some part of the Flash. The minimal unit to be erased is called a sector or block.

The board contains either one Spansion Flash (16-bit S29WS256N) chip or one Intel Strata Flash (16-bit 28F256L18) chip. In both cases, the total size of the Flash is 32 MBytes. The Spansion NOR Flash device is the default device used on the memory daughter cards.

NOR Flash memory is on CS0 which is controlled by the EIM module. The EIM module must be set up properly for NOR Flash to work.

For more information, see the data sheet for the selected hardware.

9.2 Software Operation

In a Flash-based embedded Linux system, a number of Linux technologies work together to implement a file system. Figure 9-1 illustrates the relationships between some of the standard components.

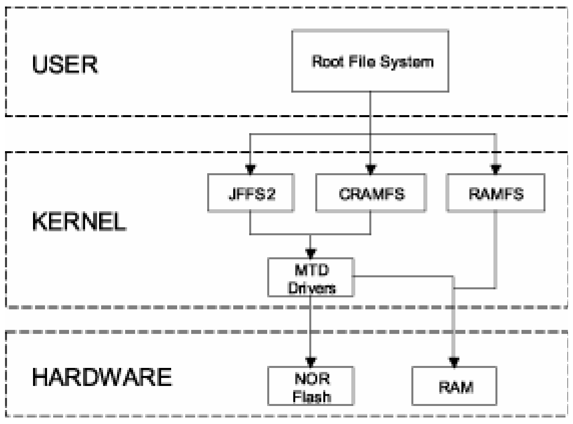


Figure 9-1. Components of a File System in a Flash-Based System

The memory technology device (MTD) subsystem for Linux is a generic interface to memory devices, such as Flash and RAM, providing simple read, write and erase access to physical memory devices. Devices called `mtdblock` devices can be mounted by JFFS, JFFS2 and CRAMFS file systems. The MTD driver provides extensive support for NOR Flash devices that support common Flash interfaces (CFIs), such as Intel, Sharp, AMD and Fujitsu. The width of the Flash bus and number of chips required to implement the bus width can be configured, or they can be automatically detected. The MTD driver layer also supports multiple Flash partitions on one set of Flash devices. The wear-leveling feature is supported by the upper layer file system, such as JFFS2. For more detailed description, see the article *Flash File Systems for Embedded Linux Systems* at http://www.linuxdevices.com/articles/AT7478621147.html.

There are two functions for loading and unloading the module. The initialization function is called when the NOR MTD map module is loaded to return NOR Flash mapping information to the upper layer MTD subsystem. The exit function is called when the module is unloaded.

9.3 Requirements

This NOR MTD implementation meets the following requirements:

- Provides necessary information for the upper layer MTD driver.
- Conforms to the Linux coding standard.

9.4 Source Code Structure

From a porting perspective, only one file (`mxc_nor.c`, which specifies i.MX NOR drivers for MTD partition) is added to provide functions to locate valid MTD partition table and its initialization. The file is available in the `<ltib_dir>/rpm/BUILD/linux/drivers/mtd/maps` directory.

9.5 Linux Menu Configuration Options

The following Linux kernel configurations are enabled for the Linux MTD module:

```
CONFIG_MTD=y
CONFIG_MTD_PARTITIONS=y
CONFIG_MTD_REDBOOT_PARTS=y
CONFIG_MTD_CHAR=y
CONFIG_MTD_BLOCK=y
CONFIG_MTD_CFI=y
CONFIG_MTD_GEN_PROBE=y
CONFIG_MTD_CFI_ADV_OPTIONS=y
CONFIG_MTD_CFI_NOSWAP=y
CONFIG_MTD_CFI_GEOMETRY=y
CONFIG_MTD_CFI_B2=y
CONFIG_MTD_CFI_I1=y
CONFIG_MTD_COMPLEX_MAPPINGS=y
```

For Intel Strata Flash, select:

```
CONFIG_MTD_CFI_INTELEXT=y
```

For Spansion Flash, select:

```
CONFIG_MTD_CFI_AMDSTD=y
```

In the menuconfig, to get to the MTD configuration, use the command ./ltib -c when located in the <ltib dir>. In the screen, select Configure Kernel, exit, and a new screen appears. The following are the options under Device Drivers > Memory Technology Device (MTD) support > RAM/ROM/Flash chip drivers:

- Detect flash chips by Common Flash Interface (CFI) probe
- Detect non-CFI AMD/JEDEC-compatible flash chips
- Support for RAM chips in bus mapping
- Support for ROM chips in bus mapping
- Support for absent chips in bus mapping

9.6 Programming Interface

Only two static functions are provided in this module:

- One initialization function that is called automatically during kernel starts
- A cleanup function that is called when the module is unloaded.

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

Chapter 10
NAND Flash Memory Technology Device (MTD) Driver

10.1 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.

10.1.1 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.

10.1.2 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/.

10.2 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

10.3 Source Code Structure

Table 10-1 shows the source files available for the NAND MTD driver. These files are under the `<ltib_dir>/rpm/BUILD/linux/drivers/mtd/nand` directory.

10.4 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:

10.5 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 11
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.

11.1 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.

11.2 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

11.3 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

11.4 Source Code Structure

Table 11-1 shows the source files available in the

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

Table 11-1. Ethernet Driver Files

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

11.5 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 12
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).

12.1 Hardware Operation

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

A brief overview of the device functionality is provided here. For details see the FEC chapter of the *i.MX35 Multimedia Applications Processor Reference Manual*.

In MII mode, there are 18 signals defined by the IEEE 802.3 standard and supported by the EMAC. SNI mode uses a subset of the 18 signals. These signals are listed in Table 12-1.

Table 12-1. Pin Usage in MII and SNI Modes

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

Table 12-1. Pin Usage in MII and SNI Modes (continued)

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

The MII management interface consists of two pins, FEC_MDIO and FEC_MDC. These pins are configured through the GPIO settings. The FEC hardware operation can be divided in the following parts. For detailed information consult the *i.MX35 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.

12.2 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

12.3 Source Code Structure

Table 12-2 shows the source files available in the `<ltib_dir>/rpm/BUILD/linux/drivers/net directory.`

Table 12-2. 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.

12.4 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.

12.5 Programming Interface

Table 12-2 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.

12.5.1 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;
```

12.5.2 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)
```

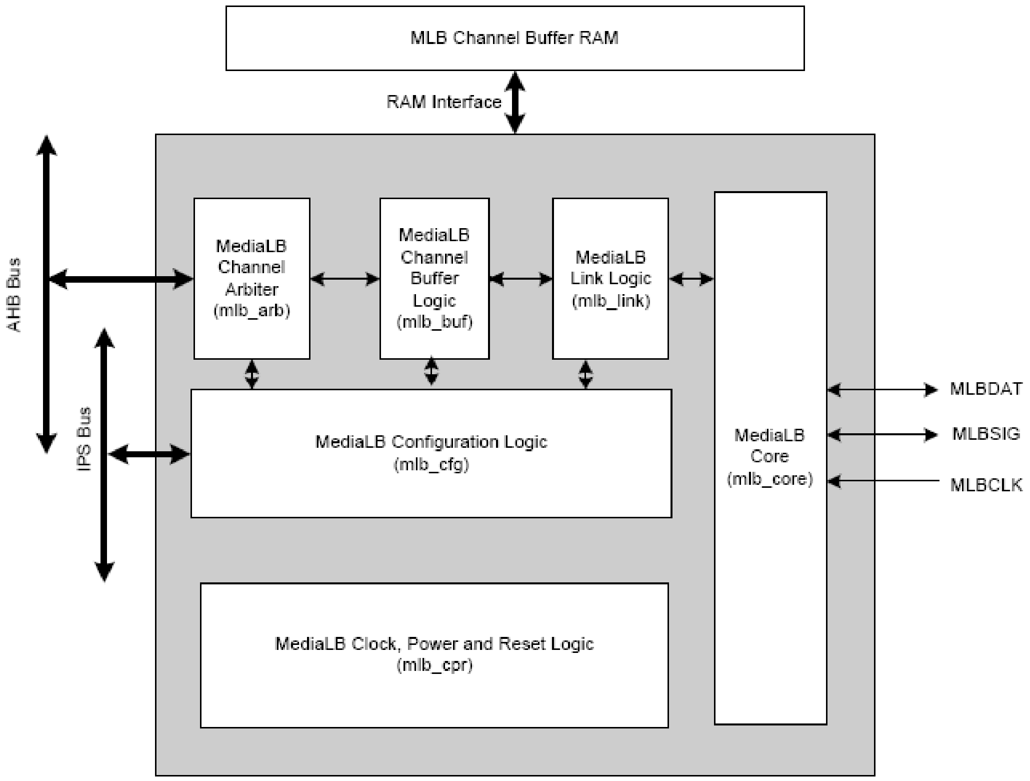
If the MAC address is not programmed, the driver sets the MAC address to "0x00:0x00:0x00:0x00:0x00:0x00:", which is not an acceptable address. The MAC address can also be set by the REDBOOT command `fconfig`.

Chapter 13
Media Local Bus Driver

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

13.1 Overview

13.1.1 MLB Device Module



MLB Channel Buffer RAM
RAM Interface
AHB Bus
IPS Bus
MediaLB Channel Arbiter (mlb_arb)
MediaLB Channel Buffer Logic (mlb_buf)
MediaLB Link Logic (mlb_link)
MediaLB Configuration Logic (mlb_cfg)
MediaLB Clock, Power and Reset Logic (mlb_cpr)
MediaLB Core (mlb_core)
MLBDAT
MLBSIG
MLBCLK

Figure 13-1. MLB Device Top-Level Block Diagram

The MediaLB module implements the Physical Layer and Link Layer of the MediaLB specification, interfacing the i.MX to the MediaLB controller. The MLB implements the 3-pin MediaLB mode and can run at speeds up to 1024Fs. It does not implement MediaLB controller functionality. All MediaLB devices support a set of physical channels for sending data over the MediaLB. Each physical channel is 4 bytes in length (quadlet) and grouped into logical channels with one or more physical channels allocated to each logical channel. These logical channels can be any combination of channel type (synchronous, asynchronous, control, or isochronous) and direction (transmit or receive).

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

13.1.1.1 Supported Feature

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

13.1.1.2 Modes of Operation

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

13.1.2 MLB Driver Overview

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

13.2 MLB Driver

13.2.1 Supported Features

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

— Channel address

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

13.2.2 MLB Driver Architecture

The MLB driver is a common linux character driver and the architecture is shown in Figure 13-2.

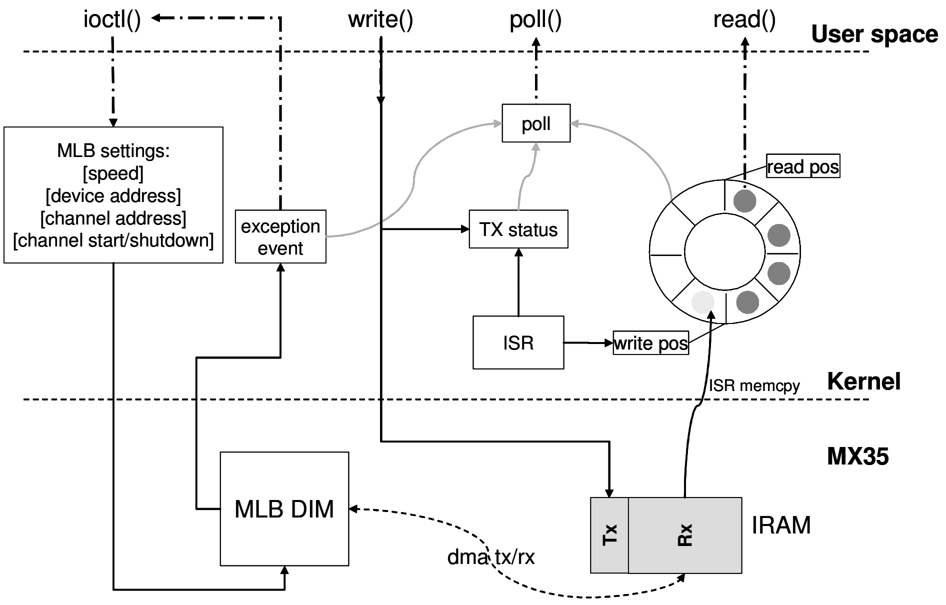


Figure 13-2. MLB Driver Architecture Diagram

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

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

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

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

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

13.2.3 Software Operation

The MLB driver provides a common interface to application.

- Packet read/write–BLOCK and NONBLOCK Packet I/O modes are supported. Only one packet can be read or written at once. The minimum read length must be greater or equal to the received packet length, meanwhile the write length must be shorter than 1024Bytes.
- Polling–The MLB driver provide polling interface which polls for three status, application can use select to get current I/O status:
  - Packet available for read (ready to read)
  - Driver is ready to send next packet (ready to write)
  - Exception event comes (ready to read)
- ioctl–MLB driver provides the following ioctl:
  - `MLB_SET_FPS`

    Argument type: unsigned int

    Set frame rate, the argument must be 256, 512 or 1024.
  - `MLB_GET_VER`

    Argument type: unsigned long

    Get MLB device module version, which is 0x02000202 by default on the i.MX35.
  - `MLB_SET_DEVADDR`

    Argument type: unsigned char

    Set MLB device address, which is used by the system channel MlbScan command.
  - `MLB_CHAN_SETADDR`

    Argument type: unsigned int

    Set the corresponding channel address [8:1] bits. This ioctl combines both tx and rx channel address, the argument format is: tx_ca[8:1] << 16 | rx_ca[8:1]
  - `MLB_CHAN_STARTUP`

    Startup the corresponding type of channel for transmit and reception.
  - `MLB_CHAN_SHUTDOWN`

    Shutdown the corresponding type of channel.
  - `MLB_CHAN_GETEVENT`

    Argument type: unsigned long

    Get exception event from MLB device module, the event is defined as a set of enumeration:

    ```
    MLB_EVT_TX_PROTO_ERR_CUR
    MLB_EVT_TX_BRK_DETECT_CUR
    MLB_EVT_RX_PROTO_ERR_CUR
    MLB_EVT_RX_BRK_DETECT_CUR
    ```

13.3 Driver Files

Table 13-1 lists the source file associated with the MLB driver that are found in the directory `<ltib_dir>/rpm/BUILD/linux/drivers/mxc/mlb/`.

Table 13-1. MLB Driver Source File List

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

13.4 Menu Configuration Options

To get to the MediaLB configuration, use the command `./ltib -c` when located in the `<ltib dir>`. In the screen, select **Configure Kernel**, exit, and a new screen appears. The CONFIG_ARCH_MX35 Linux kernel configuration is provided for this module. This option is available under:

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

Chapter 14
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

14.1 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

The I^2C slave mode is supported by a separate driver as described in Chapter 15, "I2C Slave Driver".

14.2 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.

14.3 Hardware Operation

The I^{2}C module provides the functionality of a standard I^{2}C master and slave. It is designed to be compatible with the standard Philips I^{2}C 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

14.4 Software Operation

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

14.4.1 I^{2}C Bus Driver Software Operation

The I^{2}C 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^{2}C bus. The algorithm structure contains a pointer to a function that is called whenever the I^{2}C chip driver wants to communicate with an I^{2}C device.

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

After transmitting each packet, the I^{2}C 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^{2}C bus driver uses wait queues for its operation, other device drivers should be careful not to call the I^{2}C API methods from an interrupt mode.

14.4.2 I^{2}C Device Driver Software Operation

The I^{2}C driver controls an individual I^{2}C device on the I^{2}C bus. A structure, `i2c_driver`, describes the I^{2}C 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^{2}C devices, after the driver is loaded. The `attach_adapter` callback function is called whenever a new I^{2}C bus driver is loaded in the system. When the I^{2}C 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.

14.5 Driver Features

The I^2C driver supports the following features:

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

14.6 Source Code Structure

Table 14-1 shows the I^2C bus driver source files available in the directory:

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

Table 14-1. I^2C Bus Driver Files

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

14.7 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.

14.8 Programming Interface

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

14.9 Interrupt Requirements

The I^2C module generates many kinds of interrupts. The highest interrupt rate is associated with the transfer complete interrupt as shown in Table 14-2.

Table 14-2. I^2C Interrupt Requirements

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

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

Chapter 15
I^2C Slave 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 MXC I^2C slave driver is divided into two layers: the I^2C slave core and the I^2C slave chip driver. The I^2C core driver stays at the top level and is the interface for the registered I^2C slave device to the Linux device driver model. The I^2C slave chip driver handles the low level hardware operation.

15.1 I^2C Slave Core Overview

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

15.2 I^2C Slave Chip Driver Overview

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

15.3 Hardware Operation

The I^2C module provides the functionality of a standard I^2C master and slave. It is designed to be compatible with the standard Philips I^2C bus protocol. The module supports up to 64 different clock frequencies that can be programmed by setting a value to the frequency divider register (IFDR). It also generates an interrupt when one of the following occurs:

- One byte transfer is completed.
- An address is received that matches its own specific address in slave-receive mode.
- Arbitration is lost.

For more details, see the device specification.

15.4 Software Operation

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

15.5 Requirements

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

- Supports the I^{2}C communication protocol.
- Supports the I^{2}C slave mode of operation.

15.6 Source Code Structure

Table 15-1 lists the I^{2}C bus driver source files available in the directory, `<ltib_dir>/rpm/BUILD/linux/drivers/i2c-slave`.

Table 15-1. I^{2}C Bus Driver Files

File	Description
i2c_slave_core.c	I^{2}C slave core file
i2c_slave_device.c	Interface between I^{2}C slave core and chip driver
i2c_slave_ring_buffer.c	Ring buffer source file
mxc_i2c_slave.c	I^{2}C slave chip driver for i.MX35 I^{2}C slave
`i2c_slave_client.c`	I^{2}C master device driver which makes the I^{2}C master on the board get access to I^{2}C slave
mxc_i2c_slave.h	I^{2}C slave device definition
mxc_i2c_slave_reg.h	I^{2}C registers definition

15.7 Linux Menu Configuration Options

To get to the I^{2}C configuration, use the command `./ltib -c` when located in the <`ltib dir`>. In the screen, select `Configure Kernel`, exit, and a new screen appears.

The following Linux kernel configurations are provided for this module:

- CONFIG_I2C_SLAVE–This option is available under Device Drivers > I2C Slave support.
- CONFIG_I2C_SLAVE_CORE–This option is available under Device Drivers > I2C Slave support > I2C SLAVE CORE.
- CONFIG_MXC_I2C_SLAVE–This option is available under Device Drivers > I2C Slave support > MXC I2C SLAVE.
- CONFIG_I2C_SLAVE_CLIENT–This option is available under Device Drivers > I2C Slave support > I2C SLAVE CLIENT.

15.8 Programming Interface

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

Chapter 16
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

16.1 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)

16.2 Software Operation

The following sections describe the CSPI software operation.

16.2.1 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 16-1 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.

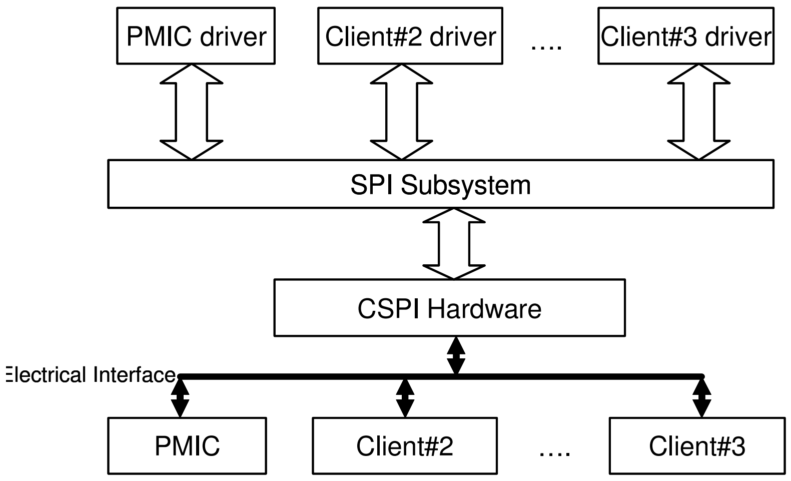


Figure 16-1. SPI Subsystem

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 16-2 shows how the different SPI drivers are layered in the SPI subsystem.

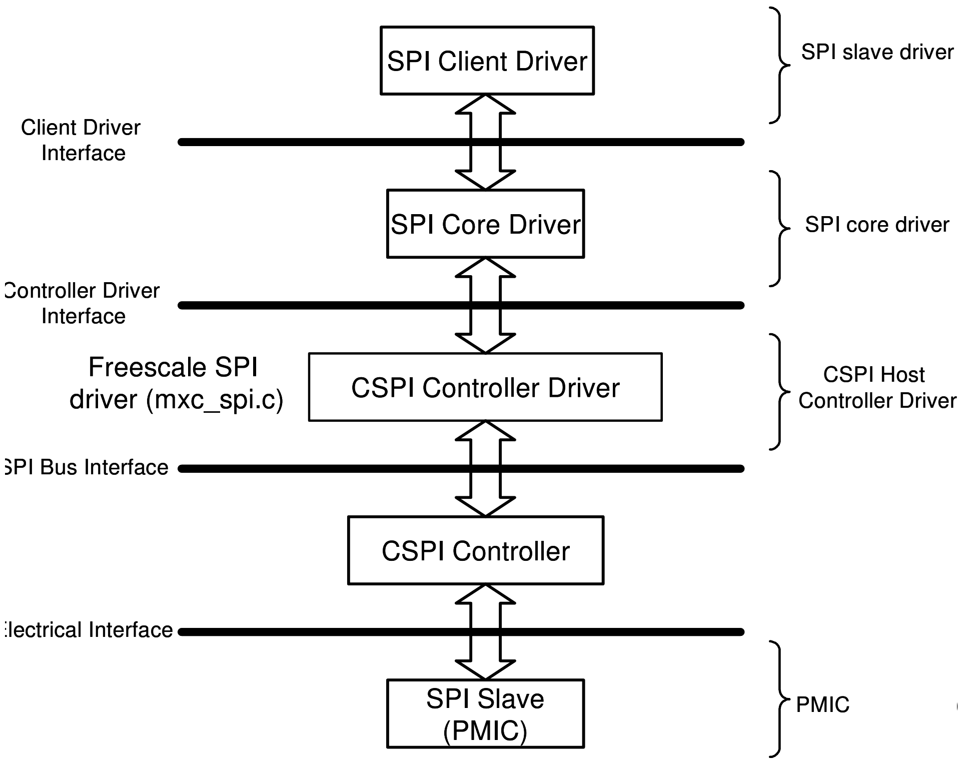


Figure 16-2. Layering of SPI Drivers in SPI Subsystem

16.2.2 Software Limitations

The CSPI driver limitations are as follows:

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

16.2.3 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.

16.2.4 CSPI Synchronous Operation

Figure 16-3 shows how the CSPI provides synchronous read/write operations.

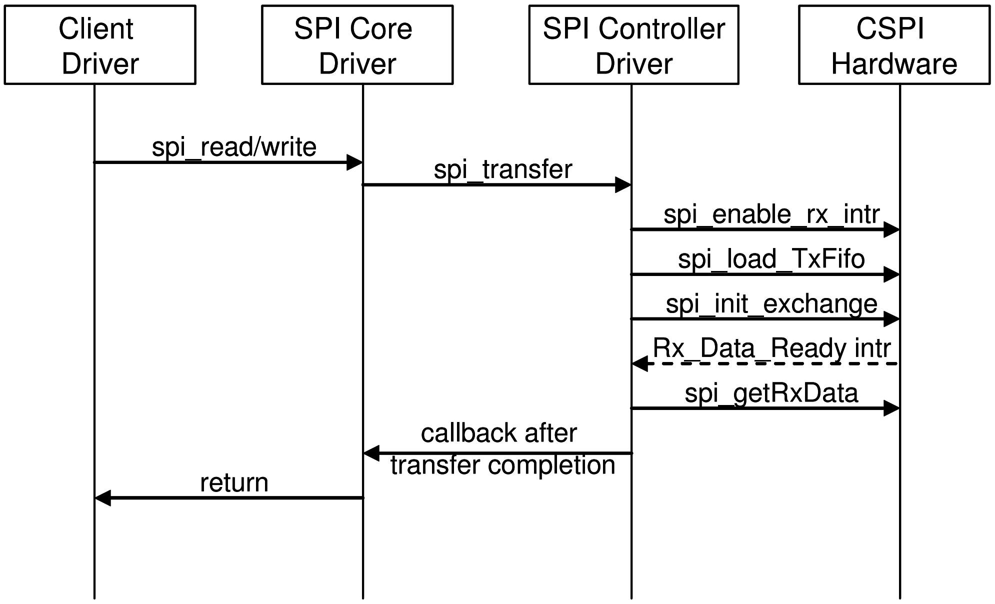


Figure 16-3. CSPI Synchronous Operation

16.3 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

16.4 Source Code Structure

Table 16-1 shows the source files available in the devices directory:

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

Table 16-1. CSPI Driver Files

File	Description
mxc_spi.c	SPI Master Controller driver

16.5 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.

16.6 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).

16.7 Interrupt Requirements

The SPI interface generates interrupts. CSPI interrupt requirements are listed in Table 16-2.

Table 16-2. 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 17 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.

17.1 Hardware Operation

Refer to the *i.MX35 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.

17.2 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.

17.3 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

17.4 Source Code Structure

Table 17-1 shows the UART driver source files that are available in the directory:

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

Table 17-1. 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 17-2 shows the header files associated with the UART driver.

Table 17-2. 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-mx25/board-mx25_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-mx35`. The source file contains UART configuration data and calls to register the device with the platform bus.

17.5 Configuration

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

17.5.1 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.

17.5.2 Source Code Configuration Options

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

17.5.2.1 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 17-5.

17.5.2.2 Board Configuration Options

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

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx35/board-mx35_.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

17.6 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.

17.7 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 17-3.

Table 17-3. 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.

17.8 Device Specific Information

17.8.1 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 17-4.

17.8.2 Board Setup Configuration

Table 17-4. UART General Configuration

Platform	Number of UARTs	Max Baudrate
i.MX35	3	1500000 (1.5 Mbps)

Table 17-5. UART Active/Inactive Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	1	1	1	--	--	--

Table 17-6. UART IRDA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	NO_IRDA	NO_IRDA	NO_IRDA	--	--	--

Table 17-7. UART Mode Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	MODE_DCE	MODE_DTE	MODE_DTE	--	--	--

Table 17-8. UART Shared Peripheral Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	-1	-1	SPBA_UART3	--	--	--

Table 17-9. UART Hardware Flow Control Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	1	0	1	--	--	--

Table 17-10. UART DMA Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	0	1	1	--	--	--

Table 17-11. UART DMA RX Buffer Size Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	1024	1024	1024	--	--	--

Table 17-12. UART UCR4_CTSTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	16	-1	16	--	--	--

Table 17-13. UART UFCR_RXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	16	16	16	--	--	--

Table 17-14. UART UFCR_TXTL Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	16	16	16	--	--	--

Table 17-15. UART Interrupt Mux Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	INTS_MUXED	INTS_MUXED	INTS_MUXED	--	--	--

Table 17-16. UART Interrupt 1 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	INT_UART1	INT_UART2	INT_UART3	--	--	--

Table 17-17. UART Interrupt 2 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	-1	-1	-1	--	--	--

Table 17-18. UART interrupt 3 Configuration

Platform	UART1	UART2	UART3	UART4	UART5	UART6
i.MX35	-1	-1	-1	--	--	--

17.9 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 18
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
- Supports 1/4/8-bit operations

18.1 Hardware Operation

The MMC communication is based on an advanced 7-pin serial bus designed to operate in a low voltage range. The eSDHC module support MMC along with SD memory and I/O functions. The eSDHC controls the MMC, SD memory, and I/O cards by sending commands to cards and performing data accesses to and from the cards. The SD memory card system defines two alternative communication protocols: SD and SPI. The eSDHC only support the SD bus protocol.

The eSDHC command transfer type and eSDHC command argument registers allow a command to be issued to the card. The eSDHC command, system control and protocol control registers allow the users to specify the format of the data and response and to control the read wait cycle.

The block length register defines the number of bytes in a block (block size). As the Stream mode of MMC is not supported, the block length must be set for every transfer.

There are four 32-bit registers used to store the response from the card in the eSDHC. The eSDHC reads these four registers to get the command response directly. The eSDHC uses a fully configurable 128×32-bit FIFO for read and write. The buffer is used as temporary storage for data being transferred between the host system and the card, and vice versa. The eSDHC data buffer access register bits hold 32-bit data upon a read or write transfer.

For receiving data, the steps are as follows:

1. The eSDHC controller generates a DMA request when there are more words received in the buffer than the amount set in the RD_WML register
2. Upon receiving this request, DMA engine starts transferring data from the eSDHC FIFO to system memory by reading the data buffer access register

To transmitting data, the steps are as follows:

1. The eSDHC controller generates a DMA request whenever the amount of the buffer space exceeds the value set in the WR_WML register
2. Upon receiving this request, the DMA engine starts moving data from the system memory to the eSDHC FIFO by writing to the Data Buffer Access Register for a number of pre-defined bytes

The read-only eSDHC Present State and Interrupt Status Registers provide eSDHC operations status, application FIFO status, error conditions, and interrupt status.

When certain events occur, the module has the ability to generate interrupts as well as set the corresponding Status Register bits. The eSDHC interrupt status enable and signal enable registers allow the user to control if these interrupts occur.

18.2 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 18-1 shows how the MMC-related drivers are layered.

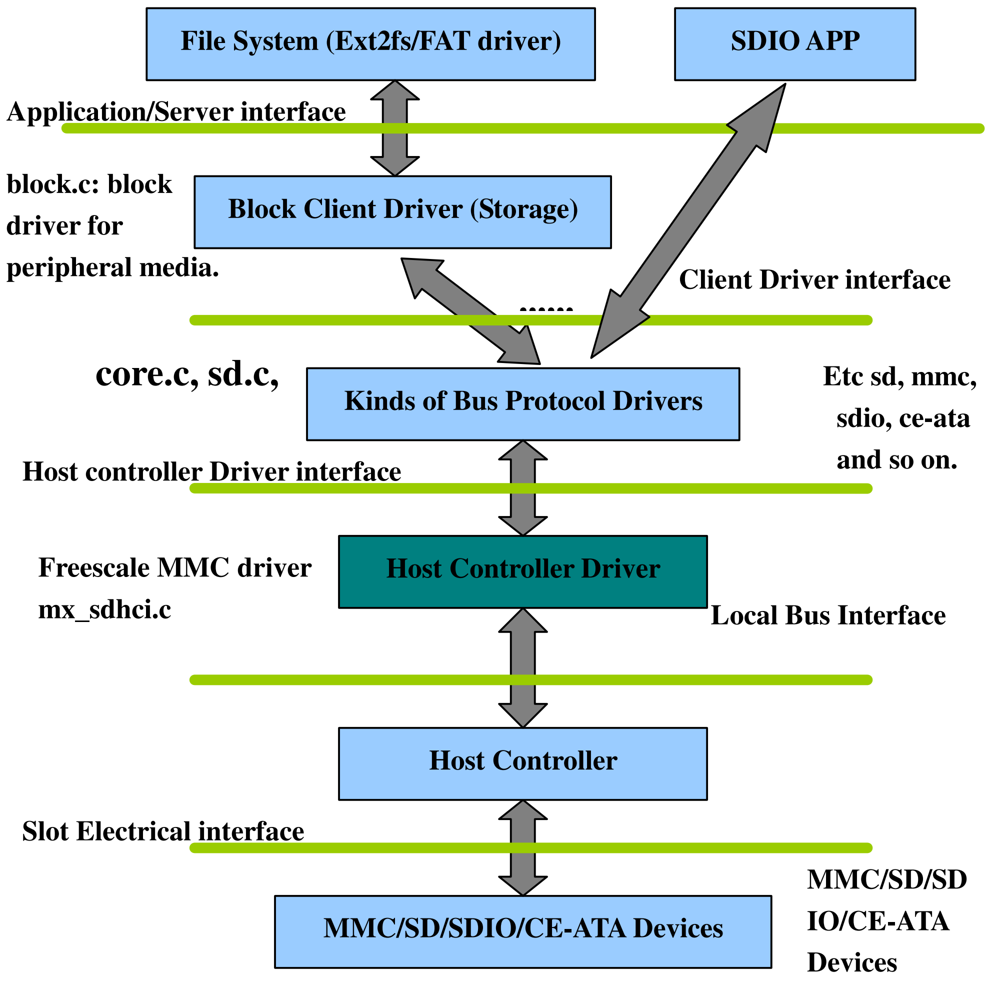

File System (Ext2fs/FAT driver)
SDIO APP
Application/Server interface
block.c: block driver for peripheral media.
Block Client Driver (Storage)
Client Driver interface
......
core.c, sd.c,
Kinds of Bus Protocol Drivers
Etc sd, mmc, sdio, ce-ata and so on.
Host controller Driver interface
Freescale MMC 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 18-1. MMC Drivers Layering

18.3 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

18.4 Source Code Structure

Table 18-1 shows the eSDHC source files available in the source directory: `<ltib_dir>/rpm/BUILD/linux/drivers/mmc/host/`.

Table 18-1. eSDHC Driver FilesMMC/SD Driver Files

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

18.5 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

18.6 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 19
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

19.1 Architectural Overview

A USB host system is composed of a number of hardware and software layers. Figure 19-1 shows a conceptual block diagram of the building block layers in a host system that support USB 2.0.

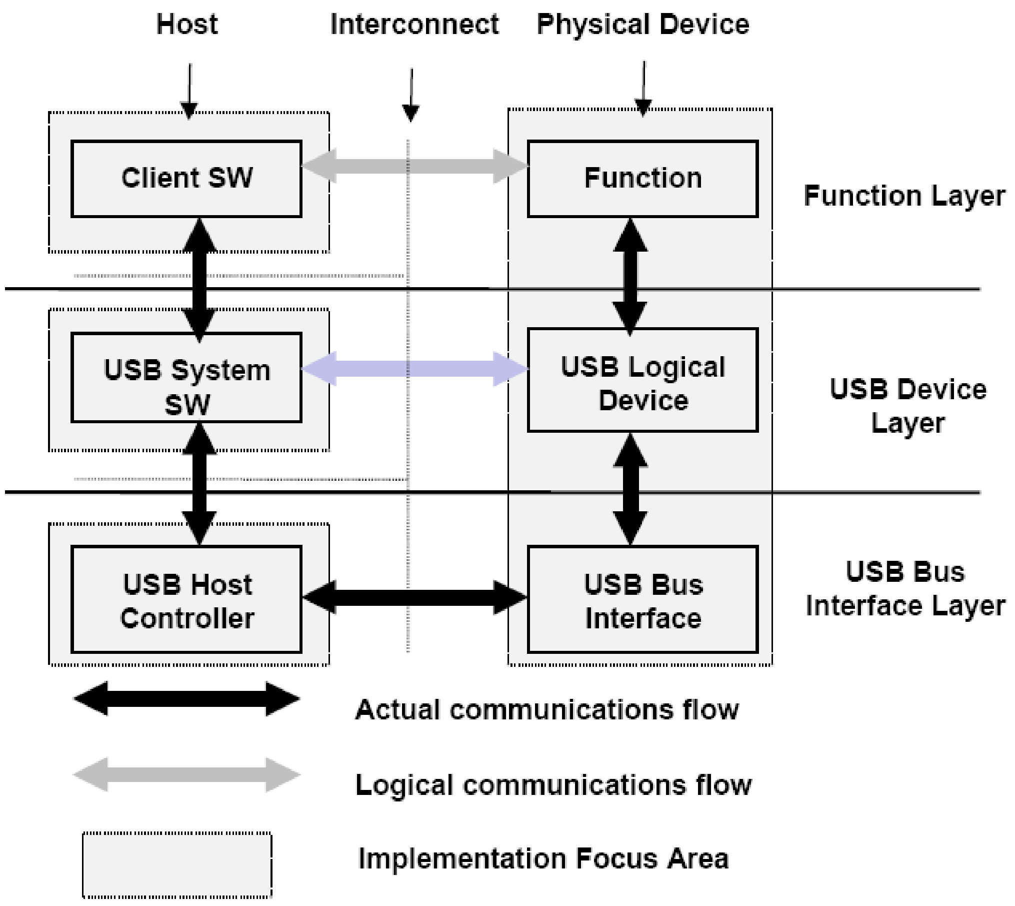


Figure 19-1. USB Block Diagram

19.2 Hardware Operation

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

19.3 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.

19.4 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

19.5 Source Code Structure

Table 19-1 shows the source files available in the source directory, `<ltib_dir>/rpm/BUILD/linux/drivers/usb`.

Table 19-1. 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 19-2 shows the platform related source files.

Table 19-2. 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 19-3 shows the platform-related source files in the directory:

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx35/`

Table 19-3. USB Platform Header Files

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

Table 19-4 shows the common platform source files in the directory:

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

Table 19-4. USB Common Platform Files

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

19.6 Menu Configuration Options

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

- CONFIG_USB—Build support for USB
- CONFIG_USB_EHCI_HCD—Build support for USB host driver. In menuconfig, this option is available under

  Device drivers > USB support > EHCI HCD (USB 2.0) support.

  By default, this option is M.
- CONFIG_USB_EHCI_ARC—Build support for selecting the ARC EHCI host. In menuconfig, this option is available under

  Device drivers > USB support > Support for Freescale controller.

  By default, this option is Y.
- CONFIG_USB_EHCI_ARC_H1—Build support for selecting the USB Host1. In menuconfig, this option is available under

  Device drivers > USB support > Support for Host1 port on Freescale controller. By default, this option is N.
- CONFIG_USB_EHCI_ARC_OTG—Build support for selecting the ARC EHCI OTG host. In menuconfig, this option is available under

  Device drivers > USB support > Support for Host-side USB > EHCI HCD (USB 2.0) support > Support for Freescale controller.

  By default, this option is Y.
- CONFIG_USB_STATIC_IRAM—Build support for selecting the IRAM usage for host. In menuconfig, this option is available under

  Device drivers > USB support > Use IRAM for USB.

  By default, this option is N.
- CONFIG_USB_EHCI_ROOT_HUB_TT—Build support for OHCI or UHCI companion. In menuconfig, this option is available under

  Device drivers > USB support > Root Hub Transaction Translators.

  By default, this option is Y selected by USB_EHCI_FSL && USB_SUPPORT.

- CONFIG_USB_STORAGE—Build support for USB mass storage devices. In menuconfig, this option is available under

  Device drivers > USB support > USB Mass Storage support.

  By default, this option is Y.
- CONFIG_USB_HID—Build support for all USB HID devices. In menuconfig, this option is available under

  Device drivers > HID Devices > USB Human Interface Device (full HID) support.

  By default, this option is M.
- CONFIG_USB_HIDINPUT—Build support for USB HID input devices. In menuconfig, this option is available under

  Device drivers > HID devices.

  By default, this option is Y.
- CONFIG_USB_GADGET—Build support for USB gadget. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support.

  By default, this option is M.
- CONFIG_USB_GADGET_ARC—Build support for ARC USB gadget. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support > USB Peripheral Controller (Freescale USB Device Controller).

  By default, this option is Y.
- CONFIG_USB_GADGET_ARC_OTG—Build support for the USB OTG port in HS/FS peripheral mode. In menuconfig, this option is available under

  Device Drivers > USB support > USB Gadget Support.

  By default, this option is Y.
- CONFIG_USB_OTG—OTG Support, support dual role with ID pin detection.

  By default, this option is N.
- CONFIG_UTMI_MXC_OTG—USB OTG pin detect support for UTMI PHY, enable UTMI PHY for OTG support.

  By default, this option is N.
- CONFIG_USB_ETH—Build support for Ethernet gadget. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support > Ethernet Gadget (with CDC Ethernet Support).

  By default, this option is M.
- CONFIG_USB_ETH_RNDIS—Build support for Ethernet RNDIS protocol. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support > Ethernet Gadget (with CDC Ethernet Support) > RNDIS support.

By default, this option is Y.

- CONFIG_USB_FILE_STORAGE—Build support for Mass Storage gadget. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support > File-backed Storage Gadget.

  By default, this option is M.
- CONFIG_USB_G_SERIAL—Build support for ACM gadget. In menuconfig, this option is available under

  Device drivers > USB support > USB Gadget Support > Serial Gadget (with CDC ACM support).

  By default, this option is M.

19.7 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.

19.8 Default USB Settings

Table 19-5 shows the default USB settings.

Table 19-5. Default USB Settings

Platform	OTG HS	OTG FS	Host1	Host2(HS)	Host2(FS)
i.MX35 3-Stack	enabled	N/A	N/A	enabled	enabled

Chapter 20 FlexCAN Driver

20.1 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.

20.2 Hardware Operation

For the information on hardware operations, see the *i.MX35 Multimedia Applications Processor Reference Manual*.

20.3 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`

20.4 Source Code Structure

Table 20-1 shows the driver source file available in the directory, `<ltib_dir>/rpm/BUILD/linux/drivers/net/can/flexcan.`

Table 20-1. 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

20.5 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 21 Real Time Clock (RTC) Driver

The i.MX processor includes an integrated real time clock (RTC) module. The RTC is used to keep the time and date while the system is turned off. The driver can also:

- Provide periodic interrupts at certain frequencies (PIE)
- Wake up the system by providing an alarm feature (AIE)

21.1 Hardware Operation

The RTC prescaler converts the incoming crystal reference clock to a 1 Hz signal, which is used to increment seconds, minutes, hours, and days Time-Of-Day (TOD) counters. The alarm functions, when enabled, generate RTC interrupts when the TOD settings reach programmed values. The sampling timer generates fixed-frequency interrupts, and the minutes stopwatch allows efficient interrupts on minute boundaries.

21.2 Software Operation

The RTC module software implementation is through the RTC driver. Besides the initialization function, it provides IOCTL functions to set up the RTC timer, interrupt, and so on. The periodic interrupt is supported at fixed frequencies of 2, 4, 8, 16, 32, 64, 128, 256, and 512 Hz given the clock input of 32.768 KHz (Other clock input frequencies are not supported by the driver). The 1 Hz periodic interrupt is also called the update interrupt (UIE). See the Linux documentation in `<ltib_dir>/rpm/BUILD/linux/Documentation/rtc.txt` for information on the RTC API.

NOTE

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

21.3 Source Code Structure

The RTC module is implemented in the `<ltib_dir>/rpm/BUILD/linux/drivers/rtc` directory. Table 21-1 shows the RTC module files. The source file for the RTC specifies the RTC function implementations.

Table 21-1. RTC Driver File List

File	Description
rtc-mxc.c	RTC driver implementation file

21.4 Programming Interface

All the Linux RTC functions are based on `rtclib`. The `include/linux/rtc.h` file specifies all the IOCTLs for the RTC. shows the IOCTLs that are listed in `include/linux/rtc.h` and which are supported by the RTC driver.

API documentation for the programming interface is in the doxygen folder of the documents package.

Chapter 22
ATA Driver

The ATA module is an AT attachment host interface mainly used to interface with hard disk devices. The ATA driver is compliant to the ATA-6 standard, and supports the following protocols:

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

22.1 Hardware Operation

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

22.2 Software Operation

22.2.1 ATA Driver Architecture

Figure 22-1 shows ATA driver architecture. File systems are built upon the block device. The integrated internal DMA engine handles the DMA transmission through the ATA Controller driver.

The DMA engine used depends on chip capability. See Table 22-1 for detailed information.

Table 22-1. DMA Engine

DMA Engine Type	Platform
Internal DMA	i.MX35

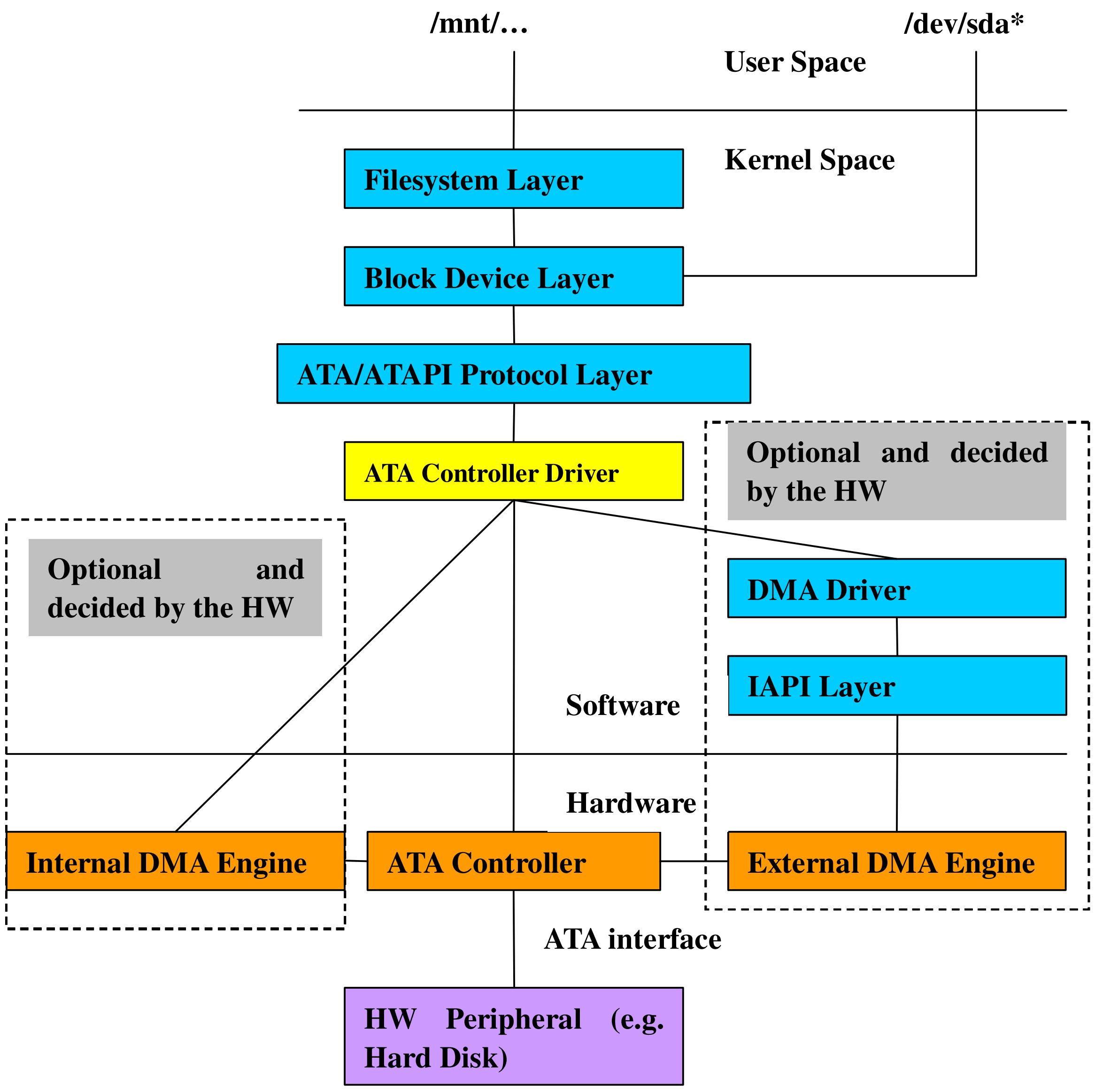


Figure 22-1. ATA Driver Layers

22.2.2 LibATA Driver

LibATA is a library used inside the Linux kernel to support ATA host controllers and devices. libATA provides an ATA driver API, class transports for ATA and ATAPI devices, and SCSI <-> ATA translation for ATA devices according to the T10 SAT specification driver. The hard disk is exposed to the application in user space by the /dev/sda interface.

22.3 Source Code Structure Configuration

22.3.1 LibATADriver

Table 22-2 lists the source file available in the directory, `<ltib_dir>/rpm/BUILD/linux/drivers/ata`

Table 22-2. LibATA Driver File List

File	Description
pata_fsl.c	ATA Driver Implementation file

22.4 Linux Menu Configuration Option

Enable these kernel configuration options as either modules (M) or built-in to the kernel (Y). These options are all under Device Drivers > Serial ATA (prod) and Parallel ATA (experimental) drivers > Freescale on-chip PATA support:

For ATA device support, enable these options: Device Drivers > SCSI device support > SCSI disk support.

For ATAPI device support, enable these options: Device Drivers > SCSI device support > SCSI CDROM support and File systems > CD-ROM/DVD File systems > ISO 9660 CDROM file system support

22.5 Board Configuration Options

Table 22-3 lists the hardware configurations for 3-Stack boards:

Table 22-3. Hardware Configuration for 3-Stack Boards

Platform	Hardware Configuration
i.MX35	• Ensure R189 is removed and R190 is populated • The ATA connector is at the back of the Personality card. • For ATAPI device support, one CD-ROM adaptor is needed on the 3-Stack board, an additional +5V power supply should be supplied to CDROM devices.

Chapter 23
Asynchronous Sample Rate Converter (ASRC) Driver

The Asynchronous Sample Rate Converter (ASRC) converts the sampling rate of a signal associated to an input clock into a signal associated to a different output clock. The ASRC supports concurrent sample rate conversion of up to 10 channels. The sample rate conversion of each channel is associated to a pair of incoming and outgoing sampling rates. The ASRC supports up to three sampling rate pairs simultaneously.

23.1 Hardware Operation

ASRC includes the following features:

- Supports ratio (Fsin/Fsout) range between 1/24 to 8.
- Designed for rate conversion between 44.1 KHz, 32 KHz, 48 KHz, and 96 KHz.
- Other input sampling rates in the range of 8 KHz to 100 KHz are also supported, but with less performance (see IC spec for more details).
- Other output sampling rates in the range of 30 KHz to 100 KHz are also supported, but with less performance.
- Automatic accommodation to slow variations in the incoming and outgoing sampling rates.
- Tolerant to sample clock jitter.
- Designed mainly for real-time streaming audio usage. Can be used for non-realtime streaming audio usage when the input sampling clocks are not available.
- In any usage case, the output sampling clocks must be activated.
- In case of real-time streaming audio, both input and output clocks need to be available and activated.
- In case of non-realtime streaming audio, the input sampling rate clocks can be avoided by setting ideal-ratio values into ASRC interface registers.

The ASRC supports polling, interrupt and DMA modes, but only DMA mode is used in the platform for better performance. The ASRC supports following DMA channels:

- Peripheral to peripheral, for example: ASRC to SPDIF
- Memory to peripheral, for example: memory to ASRC
- Peripheral to memory, for example: ASRC to memory

For more information, see the chapter on ASRC in the *Multimedia Applications Processor* documentation.

23.2 Software Operation

As an assistant component in the audio system, the ASRC driver implementation depends on the use cases in the platform. Currently ASRC is used in following two scenarios.

- Memory > ASRC > Memory, ASRC is controlled by user application or ALSA plug-in.
- Memory > ASRC > peripheral, ASRC is controlled directly by other ALSA driver.

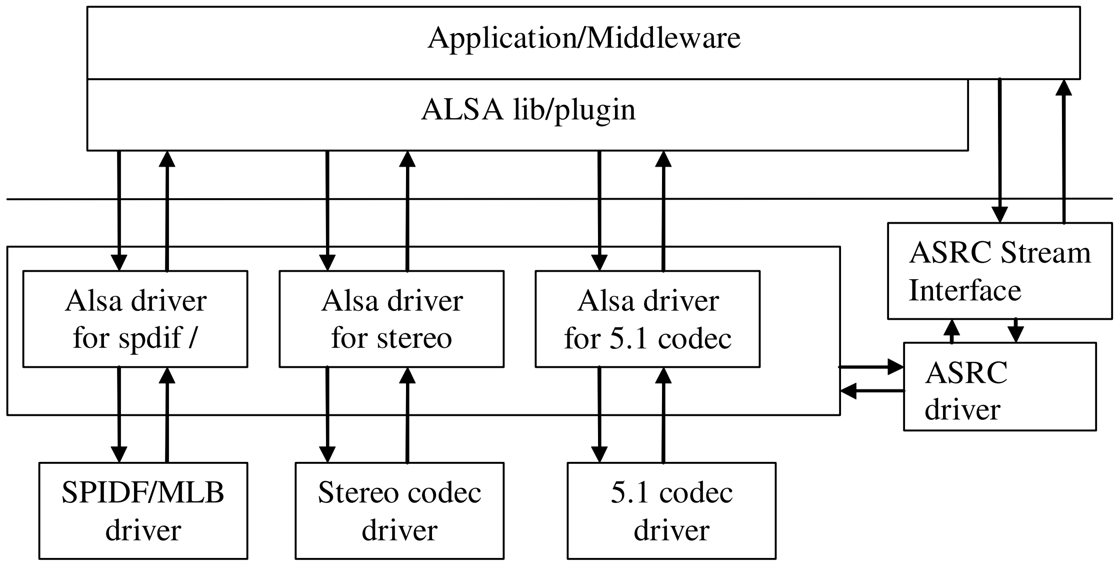


Figure 23-1. Audio Driver Interactions

As illustrated in Figure 23-1, the ASRC stream interface provides the interface for the user space. The ASRC registers itself under `/dev/mxc_asrc` and creates proc file `/proc/driver/asrc` when the module is inserted. proc is used to track the channel number for each pair. If all the pairs are not used, users can adjust the channel number through the proc file. The total channels number should equal five, or else the adjusted value cannot be saved properly. The minimum unit here is one (two channels), for example three represents six channels (three pairs).

23.2.1 Sequence for Memory to ASRC to Memory

- Open `/dev/mxc_asrc` device
- Request ASRC pair - ASRC_REQ_PAIR
- Configure ASRC pair - ASRC_CONIFG_PAIR
- Query DMA buffer info - ASRC_QUERYBUF
- Write the raw audio data (to be converted) into the DMA input buffer. Then, put the DMA input buffer in the input buffer queue (ASRC_Q_INBUF) and put the DMA output buffer in the output buffer queue (ASRC_Q_OUTBUF), so that the ASRC driver can get the buffers. To avoid an underrun, it would be better to queue in three buffers to both input and output.
- Start ASRC - ASRC_START_CONV
- Dequeue ASRC output and input buffer, copy the converted audio data for further process from output buffer and write new audio data to input buffer.
- Repeat enqueue and dequeue steps until conversion is complete
- Stop ASRC conversion - ASRC_STOP_CONV

- Release ASRC pair - ASRC_RELEASE_PAIR
- Close `/dev/mxc_asrc` device

23.2.2 Sequence for Memory to ASRC to Peripheral

- The sound card has been registered and start to enable the DMA channel in ALSA driver
- Request ASRC pair - asrc_req_pair
- Configure ASRC pair - asrc_config_pair
- Enable the DMA channel from Memory to ASRC and from ASRC to Memory
- Start DMA channel and start ASRC conversion - asrc_start_conv
- When audio data playback complete, stop DMA channel and ASRC - asrc_stop_conv
- Release ASRC pair - asrc_release_pair

23.3 Source Code Structure

Table 23-1 lists the source files available in the devices directory:

```
<ltib_dir>/rpm/BUILD/linux/drivers/mxc/asrc
<ltib_dir>/rpm/BUILD/linux/include/linux/
```

Table 23-1. ASRC Source File List

File	Description
`mxc_asrc.h`	ASRC register definitions and export function declarations
`mxc_asrc.c`	ASRC driver implementation codes including stream interface

23.3.1 Linux Menu Configuration Options

Device drivers > MXC support drivers > MXC Asynchronous Sample Rate Converter support > ASRC support

23.4 Programming Interface (Exported API and IOCTLs)

The ASRC Exported API allows the ALSA driver to use ASRC services. The ASRC IOCTLs in Table 23-2 are used for user space applications.

Table 23-2. ASRC Exported Functions

Function/ioctl	Description
`int asrc_req_pair(int chn_num, enum asrc_pair_index *index)`	Request the ASRC pair
`void asrc_release_pair (enum asrc_pair_index index)`	Release ASRC pair
`int asrc_config_pair (struct asrc_config *config)`	Configure the ASRC hardware

Table 23-2. ASRC Exported Functions (continued)

Function/ioctl	Description
`void asrc_start_conv (enum asrc_pair_index index)`	Start ASRC conversion for specific pair
`void asrc_stop_conv (enum asrc_pair_index index)`	Stop ASRC conversion
`ASRC_REQ_PAIR`	Request ASRC pair
`ASRC_CONIFG_PAIR`	Configure ASRC pair and allocate DMA buffer
`ASRC_RELEASE_PAIR`	Release ASRC pair and release DMA buffer
`ASRC_QUERYBUF`	Query buffer information
`ASRC_Q_INBUF`	Queue ASRC input buffer (audio data to be converted)
`ASRC_DQ_INBUF`	Dequeue ASRC input buffer (audio processed)
`ASRC_Q_OUTBUF`	Queue ASRC output buffer (empty buffer to hold the converted buffer)
`ASRC_DQ_OUTBUF`	Dequeue ASRC output buffer (buffer hold converted audio data)
`ASRC_START_CONV`	Start ASRC conversion
`ASRC_STOP_CONV`	Stop ASRC conversion

•

— 0 -- INCLK_NONE
— 1 -- INCLK_ESAI_RX
— 2 -- INCLK_SSI1_RX
— 3 -- INCLK_SSI2_RX
— 4 -- INCLK_SPDIF_RX
— 5 -- INCLK_MLB_CLK
— 6 -- INCLK_ESAI_TX
— 7 -- INCLK_SSI1_TX
— 8 -- INCLK_SSI2_TX
— 9 -- INCLK_SPDIF_TX
— 10 -- INCLK_ASRCK1_CLK
— default option for input clock source is 0, using the INCLK_NONE option.
— 0 -- OUTCLK_NONE
— 1 -- OUTCLK_ESAI_TX
— 2 -- OUTCLK_SSI1_TX
— 3 -- OUTCLK_SSI2_TX
— 4 -- OUTCLK_SPDIF_TX
— 5 -- OUTCLK_MLB_CLK
— 6 -- OUTCLK_ESAI_RX

- 7 -- OUTCLK_SSI1_RX
- 8 -- OUTCLK_SSI2_RX
- 9 -- OUTCLK_SPDIF_RX
- 10 -- OUTCLK_ASRCK1_CLK
- default option for output clock source is 10, using the OUTCLK_ASRCK1_CLK option.

Chapter 24
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.

24.1 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.

24.2 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.

24.3 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.

24.3.1 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

24.3.2 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.

24.3.3 Source Code Structure

Table 24-1 shows the source files for WDOG drivers that are in the following directory:

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

Table 24-1. 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`

24.3.4 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 25
FM Driver

Si4702 is used as the FM chip on the board. The Si4702 extends Silicon Laboratories Si4700 FM tuner family and further increases the ease and attractiveness of adding FM radio reception to mobile devices through small size and board area, minimum component count, and flexible programmability. A headset cable is used for antenna on the board.

25.1 FM Overview

The Si4702 device offers significant programmability and caters to the subjective nature of FM listeners and variable FM broadcast environments world-wide through a simplified programming interface and mature functionality.

Power management is also simplified with an integrated regulator allowing direct connection to a 2.7–5.5 V battery. The features of the FM module are as follows:

- Worldwide FM band support (76-108 MHz)
- Digital low-IF receiver
- Seek tuning
- Automatic frequency control (AFC)
- Automatic gain control (AGC)
- Signal strength measurement
- Adaptive noise suppression
- Volume control
- 32.768 KHz reference clock
- 2-wire and 3-wire control interface
- 2.7 to 5.5 V supply voltage
- Integrated LDO regulator allows direct connection to battery
- Integrated crystal oscillator

25.1.1 Hardware Operation

Si4702 supports both three-wire control and two-wire control. Two-wire control is chosen by driving SEN pin high during boot up.

For two-wire operation, a transfer begins with the START condition. The control word is latched internally on rising SCLK edges and is eight bits in length: a seven bit device address equal to 0010000b and a read/write bit (write = 0 and read = 1). The device acknowledges the address by setting SDIO low on the next falling SCLK edge.

For write operations, the device acknowledge is followed by an eight bit data word latched internally on rising edges of SCLK. The device always acknowledges the data by setting SDIO low on the next falling SCLK edge. An internal address counter automatically increments to allow continuous data byte writes, starting with the upper byte of register 02h, followed by the lower byte of register 02h, and onward until the lower byte of the last register is reached. The internal address counter then automatically wraps around to the upper byte of register 00h and proceeds from there until continuous writes cease. Data transfer ceases with the STOP command. After every STOP command, the internal address counter is reset.

For read operations, the device acknowledge is followed by an eight bit data word shifted out on falling SCLK edges. An internal address counter automatically increments to allow continuous data byte reads, starting with the upper byte of register 0Ah, followed by the lower byte of register 0Ah, and onward until the lower byte of the last register is reached. The internal address counter then automatically wraps around to the upper byte of register 00h and proceeds from there until continuous reads cease. After each byte of data is read, the controller IC returns an acknowledge if an additional byte of data is requested. Data transfer ceases with the STOP command. After every STOP command, the internal address counter is reset.

FM analog signals connect directly to the audio chip which routes them out to the headset.

25.1.2 Software Operation

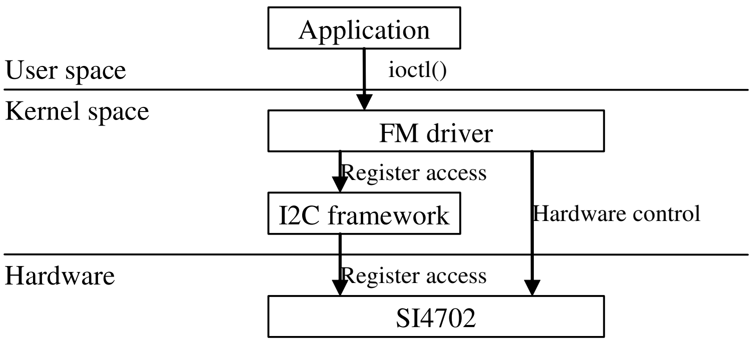


Figure 25-1. FM Driver Software Operation

The FM driver serves as an interface between kernel and user space. The driver can control hardware directly (for example, a reset operation) but most of the functional operation comes from the I^{2}C framework, which is especially convenient in the 2-wire control case.

The main software operation is as follows:

1. In initialization stage, register device in character sub-system, and then register it to I^{2}C framework.
2. In open operation, reset the chip, and initialize the register on the chip.
3. In release operation, shutdown the chip.
4. In ioctl operation, handle all the commands from user space, execute them and then feed back information if there is any.

25.2 Source Code Structure Configuration

Table 25-1 lists the source files associated with the FM driver that are available in the directory,

```
<ltib_dir>/rpm/BUILD/linux/drivers/char/mxc_si4702.c
<ltib_dir>/rpm/BUILD/linux/include/linux/mxc_si4702.h
```

Table 25-1. FM Driver Source and Header File List

File	Description
`mxc_si4702.c`	Source file for SI4702 FM driver
`mxc_si4702.h`	Header file for SI4702 FM driver

25.3 Linux Menu Configuration Options

The Linux kernel configurations are provided for this module. CONFIG_FM_SI4702 is the configuration option for the FM driver. By default, this option is M.

To load the FM drivers use the command:

```
insmod mxc_si4702.ko
```

It is located in `/lib/modules/2.6.31-203-gee1fdae/kernel/drivers/char.`

To open the FM audio, audio loopback mode should be set:

```
amixer cset numid=6 1
```

Chapter 26
Graphics Processing Unit (GPU)

The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D graphics applications. The GPU2D (2D graphics processing unit) is based on the AMD Z160 core, which is an embedded 2D and vector graphics accelerator targeting the OpenVG 1.1 graphics API and feature set. The GPU driver is delivered as binary only.

26.1 Driver Features

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

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

26.2 Hardware Operation

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

26.3 Software Operation

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

- EGL 1.3 API
- OpenVG 1.1 API

26.4 Source Code Structure

Table 26-1 lists the modules and libraries associated with GPU.

Table 26-1. GPU Related File List

File	Description
libbb2d.so libc2d.so libcsi.so libegl13.so libgsl.so libOpenVG.so libpanel2.so libres.so	GPU related libraries that are part of the Linux BSP filesystem. These libraries are located in /usr/lib/
gpu_z160.ko	Kernel level modules for the GPU driver and the display driver. These modules are located in /lib/modules/2.6.31-203-gee1fdae/kernel/drivers/char/

26.5 GPU Driver API Reference

Refer to the following web sites for detailed specifications:

- EGL 1.3 API: http://www.khronos.org/egl/ for detailed specifications
- OpenVG 1.1 API: http://www.khronos.org/openvg/

26.6 Menu Configuration Options

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

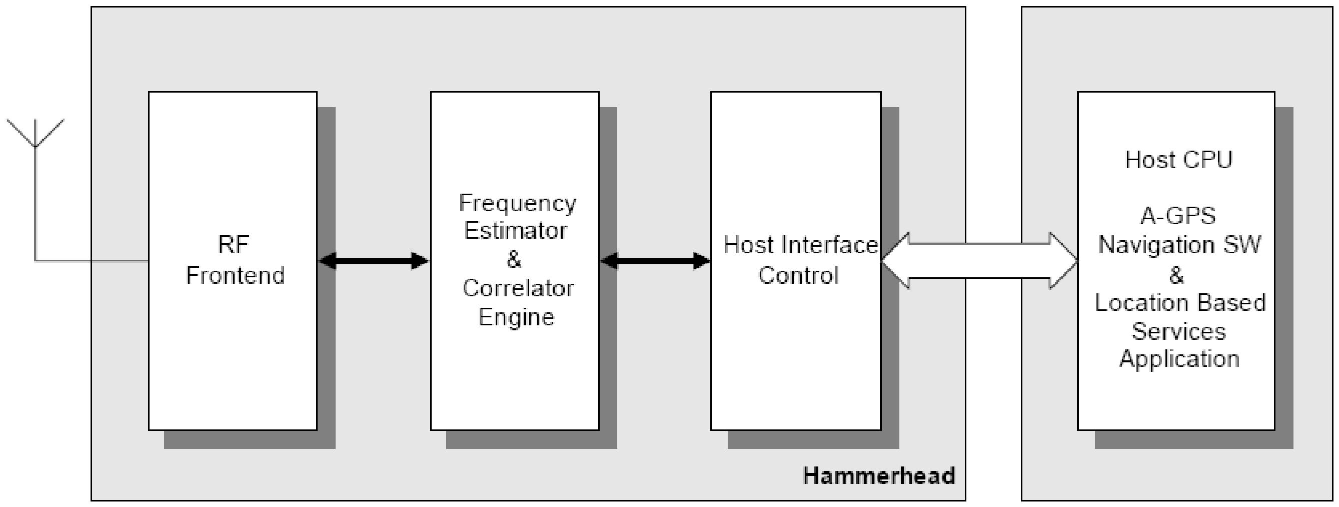
- Package list > amd-gpu-bin-mx35

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

Chapter 27
Global Positioning System (GPS) Driver

27.1 GPS Driver Overview

An external global positioning system (GPS) module is supported through the serial port and necessary GPIO resources. Currently, Broadcom's Barracuda Single Chip A-GPS Solution is supported. Since this chip set features a host-based architecture, several software components need to be loaded on the platform to enable full operation. Figure 27-1 shows a coarse block diagram of the complete GPS system architecture consisting of the BCM4750 Barracuda GPS IC and the host CPU.



RF Frontend
Frequency Estimator & Correlator Engine
Host Interface Control
Hammerhead
Host CPU
A-GPS Navigation SW & Location Based Services Application

Figure 27-1. Barracuda GPS Coarse System Architecture Including Host CPU

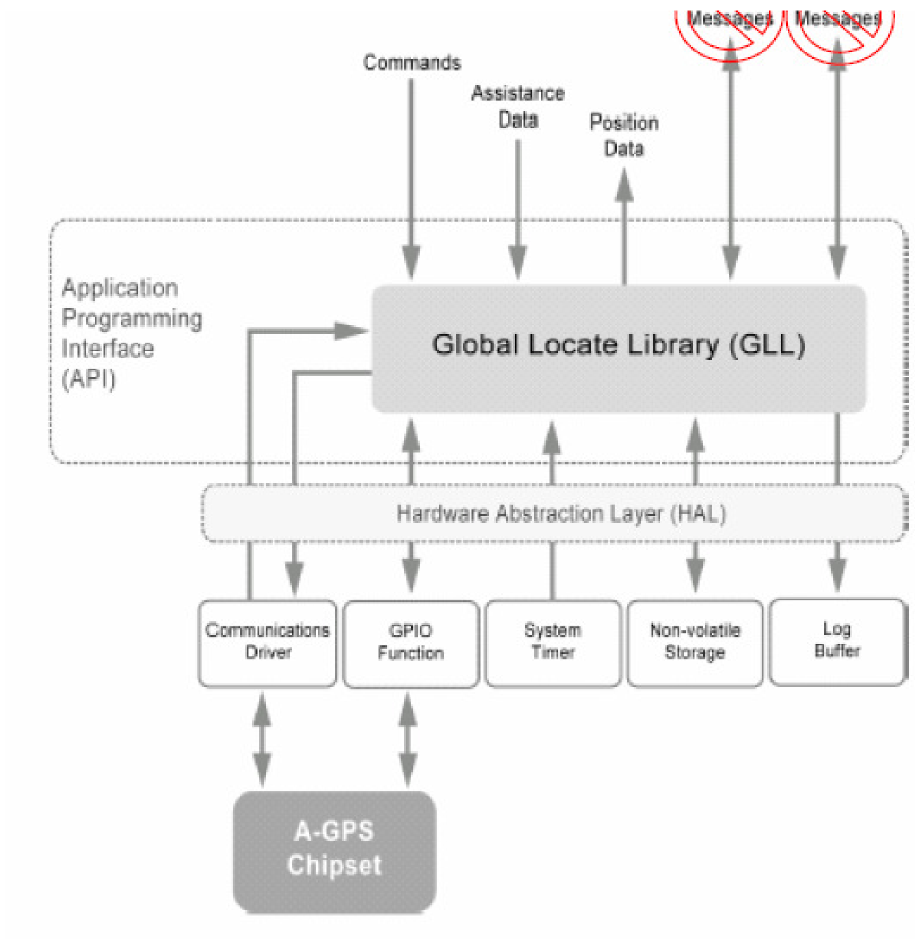


Figure 27-2. GL GPS SW Architecture

Figure 27-2 shows the GPS software architecture on the host. The communications driver provides a serial interface to the core driver and communicates with the external GPS chip set. In the current platform, the UART acts as this serial interface. The GPIO function controls the power and reset functions of the GPS chip set. The system timer, that is the GPT timer, provides an accurate timer to the GPS core driver. Non-volatile storage is used to store assistance data and useful information for the last GPS position information received, which can accelerate the next position fix. A log buffer is generated by the GLL lib and tracks problems when integrating the GL GPS solution. The hardware abstraction layer (HAL) is used to abstract specific hardware platforms, which makes the GPS core driver hardware-independent. The core driver processes GPS data and controls the work flow with the GPS device. It accepts application requests and sends the GPS position information to the upper layer.

Most of the GPS software modules are provided in binary format only. BSP source code is available for the driver that handles the concrete hardware access, additional source code controls the GPS data flow when GPS is running.

Generally, the functionality of the GPS module is segmented into two parts: control driver and core driver. The control driver is mainly for hardware-specific IO settings. The core driver (glgps_freescaleLinux) manages the GPS system, gets and parses data from the Barracuda chip, and generates position data and sends it to a pipe which connects to the application.

27.2 Hardware Operation

The GPS daughter board connects to the UART2 slot of the platform hardware. Since GPS is very sensitive to timing accuracy, the platform provides a high-accuracy, non-drifting millisecond timer function to the GPS core drivers.

The GPS Control driver manages hardware-specific IOs to regulate power usage on the platform, power on/off and reset GPIO pins setting when the GPS is supported. Before GPS power up, the reset/init sequence is done. The reset pin is asserted for a few milliseconds, then de-asserted. The reset sequence is complete when the gps_gpiodrv.ko is loaded. When GPS is launched, the power pin is asserted.

27.2.1 UART Port

The GPS module uses UART2 to communicate with the host. The device name in the Linux system is shown in Table .

Table 27-1. UART Port

Platform	UART	Device Name
i.MX35	UART3	/dev/ttymxc2

27.2.2 GPIO Control

GPIO pins are used to control the GPS module as shown in Table 27-2.

Table 27-2. GPIO Control Signals

GPIO Name Pin	Value	Description
MCU_GPIO_REG_RESET_1	Bit 5	0: Reset of GPS module is asserted 1: Reset of GPS module is de-asserted
MCU_GPIO_REG_GPIO_CONTROL_2	Bit 0	1: GPS module is power on 0: GPS module is power off

27.2.3 Hardware Dependent Parameters

The TCXO clock is a hardware parameter that must be set in the XML file in order for the GPS to work properly.

Table 27-3. Hardware Dependent Parameters

Parameter	Value Description
Frequency Plan	The TCXO has to be accurate +/- 2.0 ppm. The number after FRQ_PLAN_ describes the type of TCXO used, for example, FRQ_PLAN_13MHZ_2PPM is a 13MHz reference clock. FRQ_PLAN_13MHZ_2PPM FRQ_PLAN_16_8MHZ_2PPM FRQ_PLAN_26MHZ_2PPM FRQ_PLAN_10MHZ_2PPM_10MHZ_50PPB FRQ_PLAN_20000_2PPM_13MHZ_50PPB FRQ_PLAN_27456_2PPM_26MHZ_50PPB FRQ_PLAN_33600_2PPM_26MHZ_50PPB FRQ_PLAN_19200_2PPM_26MHZ_100PPB

27.3 Software Operation

Broadcom provides several software components to drive the GPS hardware. Broadcom's software architecture allows the LTO feature to be enabled or disabled as needed, which is a way to get the assistance data shown in Table 27-2.

Software applications communicate with the GPS module through a NMEA pipe where the incoming NMEA data is read. The GPS application creates a NMEA pipe then receives the NMEA sentences from this pipe.

27.3.1 GLGPS Configuration

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

Example 27-1 shows the XML file.

Example 27-1. GLGPS Configuration

```
<?xml version=”1.0” encoding=”utf-8”?>
<glgps xmlns=”http://www.glpals.com/” xmlns:xsi=”http://www.w3.org/2001/XMLSchema-instance”
xsi:schemaLocation=”http://www.glpals.com/ glconfig.xsd” >

  <!--HAL Confguration  -->
  <hal acPortName=”/dev/ttymxc2” lBaudRate=”115200” cLogEnabled=”true” acLogDirectory=”./log”
ltoFileName=”lto.dat”
       bPrintToConsole=”false” />

  <!-- Parameters passed to GlEngine   -->
  <gll FrqPlan=”FRQ_PLAN_26MHZ_2PPM” RfType=”GL_RF_BARRACUDA”
       LogPriMask=”LOG_DEBUG”
      LogFacMask=”LOG_GLLAPI | LOG_DEVIA | LOG_NMEA | LOG_RAWDATA “
  />
  <!-- List of jobs can be performed by the GPS controller    -->
      <!-- The default job all parameters are set to default values  -->
    <job id="normal">
      <task>
          <req_pos />
      </task>
    </job>

    <job id="cold">
      <task>
            <!-- Instructs GLL to ignore all elements stored in NVRAM listed below -->
            <startup ignore_time="true" ignore_osc="true" ignore_pos="true" ignore_nav="true"
ignore_ram_alm="true"  />
          <req_pos />
      </task>
    </job>
</glgps>
```

The `<hal>` tag defines the glhal settings (serial COM settings, enable hal log, some other paths). The parameters are explained in Table 27-4.

Table 27-4. hal Attributes

hal Attributes	Description	Comment
acPortName	Serial COM port	
lBaudRate	Baud rate	9600, 19200, 38400, 57600, 115200
acLogDirectory	Directory where the logs are placed	

The `<gll>` tag defines the gll specific parameters, as explained in Table 27-5.

Table 27-5. gll Attributes

gll attributes	Description	Comment
FrqPlan	Type of TCXO	For GPS/B it is FRQ_PLAN_26MHZ_2PPM
RfType	Type of RF chip	Should be GL_RF_BARRACUDA

Different tasks are configured in Example 27-1. The task named *normal* makes an autonomous fix every second and the task named cold starts autonomous fixes with no assisted data.

27.3.2 Driver Configuration

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

27.3.2.1 Linux Menu Configuration Options

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

The CONFIG_GPS_IOCTRL Linux kernel configuration option is provided for the GPS GPIO control driver. It is the build option for GPS GPIO control driver support. In `menuconfig`, it can be located under Device Drivers > MXC Support Drivers > Broadcom GPS ioctrl support > GPS ioctrl support. By default, this option is M for all architectures.

27.3.3 Source Code

Table 27-6 lists the source files available in the source directory

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

Table 27-6. GPS Driver Source Code

File	Description
agpsgpiodev.c	Main file for GPIO kernel module
agpsgpiodev.h	Head file of simple character device interface for AGPS kernel module

27.3.4 LTO Feature (Optional)

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

The reference board should be connected through a NFS server. Put the LTO file in the folder pointed to by the GPS driver. It is the responsibility of the user to implement the method used to get a TCP/IP connection on the platform (Bluetooth, IP over USB).

27.3.4.1 Enabling The LTO Feature on the Platform

Contact Broadcom to obtain the necessary license for enabling this feature on the Freescale reference platform. Submit the platform name and platform type to obtain the license. Also, ensure that the LTO feature is enabled through the different input parameters of the GPS driver.

27.3.5 Power Management

The GPS driver automatically manages the power management of the Broadcom chip set by toggling the different IOs of the chip set. The GPS chip set enters low power standby mode when the GPS driver is stopped.

27.3.6 irm Commands

While the GLGPS is running, commands can be sent to the process. There is a special FIFO file in `/var/run/glgpsctrl` where the commands are sent. Commands are ASCII strings with the `$pglirm, prefix + command name` format.

- `$pglirm,req_pos,name,period,N1,fixcount,N2,validfix,N3,duration_sec,N4`

  Creates a periodic position request.
  – Name – Unique request identifier, it also used by GLHAL to output NMEA data
  – Period – Update period in milliseconds
  – Fixcount – Stop after that number of fixes (valid or invalid) reported
  – Validfix – Stop after that number of valid fixes reported
  – Duration_sec – Stop after that many seconds
- `$pglirm,req_pos_single,name,acc,timeout,N1`

  Creates a single shot request.
  – Name – Unique request identifier, it also used by GLHAL to output NMEA data
  – Acc – Accuracy QoS parameter
  – Timeout – Time-out QoS parameter
- `$pglirm,req_aid,name`

  Queries for what assistance data is missing.
  – Name – Unique request identifier, it also used by GLHAL to output NMEA data
- `$pglirm,factory,test,prn,17,timeout,16,GL_FACT_TEST_MODE,GL_FACT_TEST_CONT,GL_FACT_TEST_ITEMS,GL_FACT_TEST_WER`

  Creates a factory request.
  – Test – Unique request identifier, it also used by GLHAL to output NMEA data
  – Prn – PRN number
  – Timeout – How long to run test for
  – GL_FACT_TEST_MODE – Test mode [GL_FACT_TEST_ONCE|GL_FACT_TEST_CONT]
  – GL_FACT_TEST_ITEMS – What to test GL_FACT_TEST_CN0|GL_FACT_TEST_FRQ|GL_FACT_TEST_WER
- `$pglirm,startup,{[ignore_osc|ignore_rom_alm|ignore_rom_alm|ignore_pos|ignore_ram_alm|ignore_time],[true|false]}`

  Tells the GLCT to ignore specified elements of the data previously stored in nonvolatile storage.
- `$pglirm,stop,name`

  Stops an ongoing request with a name "name"; The name "all" is reserved to stop all ongoing requests.
- `$pglirm,quit`

  Causes GLCT to exit.
- `$pglirm,pwm,[on|off]`

Turns auto power management on or off

As an example, to send quit command in a shell, do the following:

```
echo ‘$pglirm,quit’ >/var/run/glgpsctrl
```

Chapter 28 OProfile

OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead. OProfile is released under the GNU GPL. It consists of a kernel driver, a daemon for collecting sample data, and several post-profiling tools for turning data into information.

28.1 Overview

OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling. All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

28.2 Features

The features of the OProfile are as follows:

- Unobtrusive—No special recompilations or wrapper libraries are necessary. Even debug symbols (-g option to `gcc`) are not necessary unless users want to produce annotated source. No kernel patch is needed; just insert the module.
- System-wide profiling—All code running on the system is profiled, enabling analysis of system performance.
- Performance counter support—Enables collection of various low-level data and association for particular sections of code.
- Call-graph support—With an 2.6 kernel, OProfile can provide gprof-style call-graph profiling data.
- Low overhead—OProfile has a typical overhead of 1–8% depending on the sampling frequency and workload.
- Post-profile analysis—Profile data can be produced on the function-level or instruction-level detail. Source trees, annotated with profile information, can be created. A hit list of applications and functions that utilize the most CPU time across the whole system can be produced.
- System support—Works with almost any 2.2, 2.4 and 2.6 kernels, and works on ARM11based platforms.

28.3 Hardware Operation

OProfile is a statistical continuous profiler. In other words, profiles are generated by regularly sampling the current registers on each CPU (from an interrupt handler, the saved PC value at the time of interrupt is stored), and converting that runtime PC value into something meaningful to the programmer.

OProfile achieves this by taking the stream of sampled PC values, along with the detail of which task was running at the time of the interrupt, and converting the values into a file offset against a particular binary file. Each PC value is thus converted into a tuple (group or set) of binary-image offset. The userspace tools can use this data to reconstruct where the code came from, including the particular assembly instructions, symbol, and source line (through the binary debug information if present).

Regularly sampling the PC value like this approximates what actually was executed and how often and more often than not, this statistical approximation is good enough to reflect reality. In common operation, the time between each sample interrupt is regulated by a fixed number of clock cycles. This implies that the results reflect where the CPU is spending the most time. This is a very useful information source for performance analysis.

The ARM CPU provides hardware performance counters capable of measuring these events at the hardware level. Typically, these counters increment once per each event and generate an interrupt on reaching some pre-defined number of events. OProfile can use these interrupts to generate samples and the profile results are a statistical approximation of which code caused how many instances of the given event.

The ARM1136 Platform Event Monitor (EVTMON) is a 32-bit IP-bus peripheral used for monitoring the Level 2 Cache Controller (L2CC) events through the L2CC event bus interface. The EVTMON contains six event counters (EMC0-EMC5) which may be used to count six different events selected from a list of ten possible external events (through the L2CC event bus) or seven internal events (overflows or clock edges).

28.4 Software Operation

28.4.1 Architecture Specific Components

If OProfile supports the hardware performance counters available on a particular architecture. Code for managing the details of setting up and managing these counters can be located in the kernel source tree in the relevant `<ltib_dir>/rpm/BUILD/linux/arch/arm/oprofile` directory. The architecture-specific implementation operates through filling in the `oprofile_operations` structure at initialization. This provides a set of operations, such as setup(), start(), stop(), and so on, that manage the hardware-specific details the performance counter registers.

The other important facility available to the architecture code is `oprofile_add_sample`(). This is where a particular sample taken at interrupt time is fed into the generic OProfile driver code.

28.4.2 oprofilefs Pseudo Filesystem

OProfile implements a pseudo-filesystem known as oprofilefs, which is mounted from userspace at `/dev/oprofile`. This consists of small files for reporting and receiving configuration from userspace, as well as the actual character device that the OProfile userspace receives samples from. At setup() time, the architecture-specific code may add further configuration files related to the details of the performance counters. The filesystem also contains a `stats` directory with a number of useful counters for various OProfile events.

28.4.3 Generic Kernel Driver

The generic kernel driver resides in `<ltib_dir>/rpm/BUILD/linux/drivers/oprofile/`, and forms the core of how OProfile operates in the kernel. The generic kernel driver takes samples delivered from the architecture-specific code (through `oprofile_add_sample`()), and buffers this data (in a transformed configuration) until releasing the data to the userspace daemon through the `/dev/oprofile/buffer` character device.

28.4.4 OProfile Daemon

The OProfile userspace daemon takes the raw data provided by the kernel and writes it to the disk. It takes the single data stream from the kernel and logs sample data against a number of sample files (available in `/var/lib/oprofile/samples/current/`). For the benefit of the separate functionality, the names and paths of these sample files are changed to reflect where the samples were from. This can include thread IDs, the binary file path, the event type used, and more.

After this final step from interrupt to disk file, the data is now persistent (that is, changes in the running of the system do not invalidate stored data). This enables the post-profiling tools to run on this data at any time (assuming the original binary files are still available and unchanged).

28.4.5 Post Profiling Tools

The collected data must be presented to the user in a useful form. This is the job of the post-profiling tools. In general, they collate a subset of the available sample files, load and process each one correlated against the relevant binary file, and produce user readable information.

28.5 Requirements

The requirements of OProfile are as follows:

- The user must add Oprofile support with ARM11 PMU events for i.MX platforms. Enter the LTIB configuration screen, use the command ./ltib -c from the <ltib dir>, and select Package List > oprofile.
- The user must add ARM11 L2 Cache EVTMON support to oprofile for i.MX platforms

28.6 Source Code Structure

Oprofile platform-specific source files are available in the directory:

`<ltib_dir>/rpm/BUILD/linux/arch/arm/oprofile/`

Table 28-1. OProfile Source Files

File	Description
op_model_arm11.c	Source file with the implementations required to support HW performance counters on both PMU and EVTMON.
op_arm_model.h	Header File with the register and bit definitions
common.c	Source file with the implementation required for all platforms

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

28.7 Menu Configuration Options

The following Linux kernel configurations are provided for this module. To get to the Oprofile configuration, use the command `./ltib -c` from the <`ltib dir`>. On the screen, first go to Package list and select oprofile. Then return to the first screen and, select **Configure Kernel**, then exit, and a new screen appears.

- CONFIG_OPROFILE—configuration option for the oprofile driver. In the menuconfig this option is available under

  General Setup > Profiling support (EXPERIMENTAL) > OProfile system profiling (EXPERIMENTAL)

28.8 Programming Interface

This driver implements all the methods required to configure and control PMU and L2 cache EVTMON counters. Refer to the doxygen documentation for more information (in the doxygen folder of the documentation package).

28.9 Interrupt Requirements

The number of interrupts generated with respect to the OProfile driver are numerous. The latency requirements are not needed. The rate at which interrupts are generated depends on the event.

28.10 Example Usage in ARM11

The Oprofile driver can be used to profile both the user-space and the kernel-space code. The usage scenario for both cases are discussed below. Follow the commands listed below on the target to set up Oprofile (run 'opcontrol --help' command for more information).

For user space:

```
$opcontrol --no-vmlinux
```

For kernel space:

```
$opcontrol --vmlinux=path_to_vmlinux
$opcontrol --init
$opcontrol --event=event_name:count --event=event_name:count
$opcontrol --start
$opcontrol --dump
$opreport
$opcontrol --stop
```

Using opreport, opannotate, oparchive and opgprof commands on the target you can get profiled information.

For ARM11 platform with L2 cache EVTMON support:

As the EVTMON is supported using ECT workaround, follow the configuration below to monitor L2 cache events:

NOTE

Only one L2 cache event may be configured.

Set ETMEXTOUT[0] PMU event along with the required L2 cache event to be monitored as shown below. Event count for ETMEXTOUT[0] event should be the L2 cache event count, configured as below.

```
root$ opcontrol --event=ETMEXTOUT[0]:l2cache_event_count --event=DREQ:1000.
```

For Timer interrupt mode support:

When using the timer interrupt, the event is always TIMER and there is no configuration possible. Follow below commands to setup timer interrupt mode.

```
$ opcontrol --no-vmlinux
$ opcontrol --init
$ ophelp
Using timer interrupt.
$ opcontrol --start
Using 2.6+ OProfile kernel interface.
Using log file /var/lib/oprofile/oprofiled.log
Daemon started.
Profiler running.
$ opcontrol --dump
$ opreport
CPU: CPU with timer interrupt, speed 0 MHz (estimated)
Profiling through timer interrupt
          TIMER:0|
  samples|      %|
------------------
      705 93.6255 no-vmlinux
       25  3.3201 bash
       13  1.7264 libc-2.5.so
        7  0.9296 ld-2.5.so
        2  0.2656 busybox
        1  0.1328 libm-2.5.so
```

28.11 Device Specific Information

For ARM11 PMU support:

For ARM11 cores with rev r0p2 (for example, i.MX31 ADS) to configure PMU event numbers 0x7 (Instruction count) or 0x22 (event of ETMEXTOUT[0] & ETMEXTOUT[1]), configure the cycle count event with the same event count. Also, the minimum event count for cycle count and increment cycle events must be 5000.

Chapter 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.