i.MX28 EVK Linux

Reference Manual

Document Number: 924-76389
Rev. 2010.08
08/2010

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

ARM POWERED®

freescale™ semiconductor

Contents

About This Book

Chapter 1 Introduction

Chapter 2 Architecture

Chapter 3
Machine Specific Layer (MSL)

Chapter 4
Direct Memory Access Controller (DMAC) API

Chapter 5
Persistent Bits Driver

Chapter 6
Unique ID on Boot Media

Chapter 7
CPU Frequency Scaling (CPUFREQ) Driver

Chapter 8
i.MX28 Static Power Management Driver

Chapter 9
Frame Buffer Driver

Chapter 15
ENET IEEE-1588 Driver

Chapter 16
Programmable 3-Port Ethernet Switch Driver

Chapter 17
Low-Level Keypad Driver

Chapter 18
Touch Screen and ADC Drivers

Chapter 19
Inter-IC (I2C) Driver

Chapter 20
Data Co-Processor (DCP) Driver

Chapter 21
SPI Bus Driver

Chapter 22
MMC/SD/SDIO Host Driver

Chapter 23
Universal Asynchronous Receiver-Transmitter (UART) Driver

Chapter 24
ARC USB Driver

Chapter 25
Real Time Clock (RTC) Driver

Chapter 26
Watchdog (WDOG) Driver

Chapter 27
Battery Charger and Power Source Manager (PSM) Driver

Chapter 28
LED Pulse Width Modulator (PWM) Driver

Chapter 29
Frequently Asked Questions

Tables

Figures

About This Book

The Linux Board Support Package (BSP) represents a porting of the Linux Operating System (OS) to the i.MX processors and its associated reference boards. The BSP supports many hardware features on the platforms and most of the Linux OS features that are 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 knowledge of the Linux 2.6 kernel internals, driver models, and i.MX processors.

Conventions

This document uses the following notational conventions:

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

Definitions, Acronyms, and Abbreviations

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

Definitions and Acronyms

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

Definitions and Acronyms (continued)

Term	Definition
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 be done only 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
ISR	Interrupt Service Routine

Definitions and Acronyms (continued)

Term	Definition
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	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 comes 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 12 Mb/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:

- *PDK Linux Quick Start Guide*
- *BSP API Document (BSP Doxygen Code Documentation)*
- *PDK Linux User's Guide*
- *PDK Hardware User's Guide*
- [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.MX28 Applications Processor

The purpose of this software package is to support Linux on the i.MX family of Integrated Circuits (ICs) and their associated platforms (). It provides the necessary software 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, Graphical User Interface (GUI) components, Java Virtual Machine (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 the 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	Machine Specific Layer (MSL) 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 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
DMAC	Both AHB-to-APBH and AHB-to-APBX DMA support configurable DMA descript chain.	Chapter 4, “Direct Memory Access Controller (DMAC) API”	i.MX28
Power Management Drivers			
Low-level PM Drivers	The low-level power management driver is responsible for implementing hardware-specific operations to meet power requirements and also to conserve power on the development platforms. Driver implementations are often different for different platforms. It is used by the DPM layer.	Chapter 8, “i.MX28 Static Power Management Driver”	i.MX28
CPU Frequency Scaling	The CPU frequency scaling device driver allows the clock speed of the CPUs to be changed on the fly.	Chapter 7, “CPU Frequency Scaling (CPUFREQ) Driver”	i.MX28

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

Feature	Description	Chapter Source	Applicable Platform
Multimedia Drivers			
LCD	The LCD interface driver supports the Samsung LMS430xx 4.3” WQVGA LCD panel.	Chapter 10, “LCD Interface (LCDIF) Driver”	i.MX28
Frame Buffer	The frame buffer driver uses the Linux kernel frame buffer driver framework. It implements the platform driver for a frame buffer device. The implementation uses the LCDIF API for generic LCD low-level operations.	Chapter 9, “Frame Buffer Driver”	i.MX28
Back Light	The LCD backlight driver uses the Linux kernel frame buffer/backlight driver framework.	Chapter 11, “Backlight Driver”	i.MX28
Pixel Pipeline	The Pixel Pipeline (PxP) is a Linux kernel Video4Linux driver.	Chapter 13, “Pixel Pipeline (PxP) Driver”	i.MX28
Sound Drivers			
ALSA Sound	The Advanced Linux Sound Architecture (ALSA) is a sound driver that provides ALSA and OSS compatible applications with the means to perform audio playback and recording functions. ALSA has a user-space component called ALSAlib that can extend the features of audio hardware by emulating the same in software (user space), such as resampling, software mixing, snooping, and so on. The ASoC Sound driver supports stereo codec playback and capture through SSI.	Chapter 12, “Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver”	i.MX28
Memory Drivers			
NAND MTD	The NAND MTD driver interfaces with the integrated NAND controller. It can support various file systems, such as UBI and UBIFSCRAMFS 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 14, “NAND Flash Driver	i.MX28
Keypad	The keypad driver interfaces Linux to the keypad controller (KPP). The software operation of the keypad driver follows the Linux keyboard architecture.	Chapter 17, “Low-Level Keypad Driver”	i.MX28
Touch Screen and ADC	A touch screen and associated add measurement functions to the touch screen.	Chapter 18, “Touch Screen and ADC Drivers”	i.MX28
DCP	The DCP cryptography driver is used to accelerate cryptography operations (AES) in the kernel space and user-space. The DCP support AES EBC encryption and decryption by utilizing the hardware OTP KEY0 which is not readable by software.	Chapter 20, “Data Co-Processor (DCP) Driver”	i.MX28

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

Feature	Description	Chapter Source	Applicable Platform
Bus Drivers			
I^2C	The I^2C bus driver is a low-level interface that is used to interface with the I^2C bus. This driver is invoked by the I^2C chip driver; it is not exposed to the user space. The standard Linux kernel contains a core I^2C module that is used by the chip driver to access the bus driver to transfer data over the I^2C bus. This bus driver supports: • Compatibility with the I^2C bus standard • Bit rates up to 400 Kbps • Standard I^2C master mode • Power management features by suspending and resuming I^2C.	Chapter 19, “Inter-IC (I2C) Driver”	i.MX28
CSPI	The low-level Configurable Serial Peripheral Interface (CSPI) driver interfaces a custom, kernel-space API to both CSPI modules. It supports the following features: • Interrupt-driven transmit/receive of SPI frames • Multi-client management • Priority management between clients • SPI device configuration per client	Chapter 21, “SPI Bus Driver”	i.MX28
MMC/SD/SDIO - SDHC	The MMC/SD/SDIO Host driver is implemented using the i.MX28 SPI component, which supports SD/MMC mode.	Chapter 22, “MMC/SD/SDIO Host Driver”	i.MX28
UART Drivers			
Debug and Application UARTs	These are three serial UARTs. One that has no DMA support and is intended to work as a debug console (debug UART), and two are high-performance UARTs, which are intended to be used by applications (application UART, appUART).	Chapter 23, “Universal Asynchronous Receiver-Transmitter (UART) Driver”	i.MX28
General Drivers			
USB	The USB driver implements a standard Linux driver interface to the ARC USB-OTG controller.	Chapter 24, “ARC USB Driver”	i.MX28
RTC	This is the integrated Real Time Clock (RTC) module. The RTC is used to keep the time and date while the system is turned off. Additionally, it provides the PIE (periodic interrupt at a specific frequency) and AIE (wake up the system by providing an alarm) features.	Chapter 25, “Real Time Clock (RTC) Driver”	i.MX28
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 26, “Watchdog (WDOG) Driver”	i.MX28

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

Feature	Description	Chapter Source	Applicable Platform
Battery Charger	The battery charger device driver for Linux provides support for controlling the battery interface circuits and power source detection.	Chapter 27, “Battery Charger and Power Source Manager (PSM) Driver”	i.MX28
PWM LED	The PWM LED driver provides a standard framework by which to control LEDs attached to PWM interfaces.	Chapter 28, “LED Pulse Width Modulator (PWM) Driver”	i.MX28
Bootloaders			
uBoot	uBoot is an open source boot loader.	See uBoot User guide	i.MX28
GUI			
gnome	gnome is a Network Object Model Environment supported by the GUN.	See Gnome mobile Note	i.MX28

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 that are common to all platforms are referred as i.MX drivers and drivers unique to a specific platform are referred 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, and hardware-specific drivers and functions provided by Freescale for the i.MX processors.

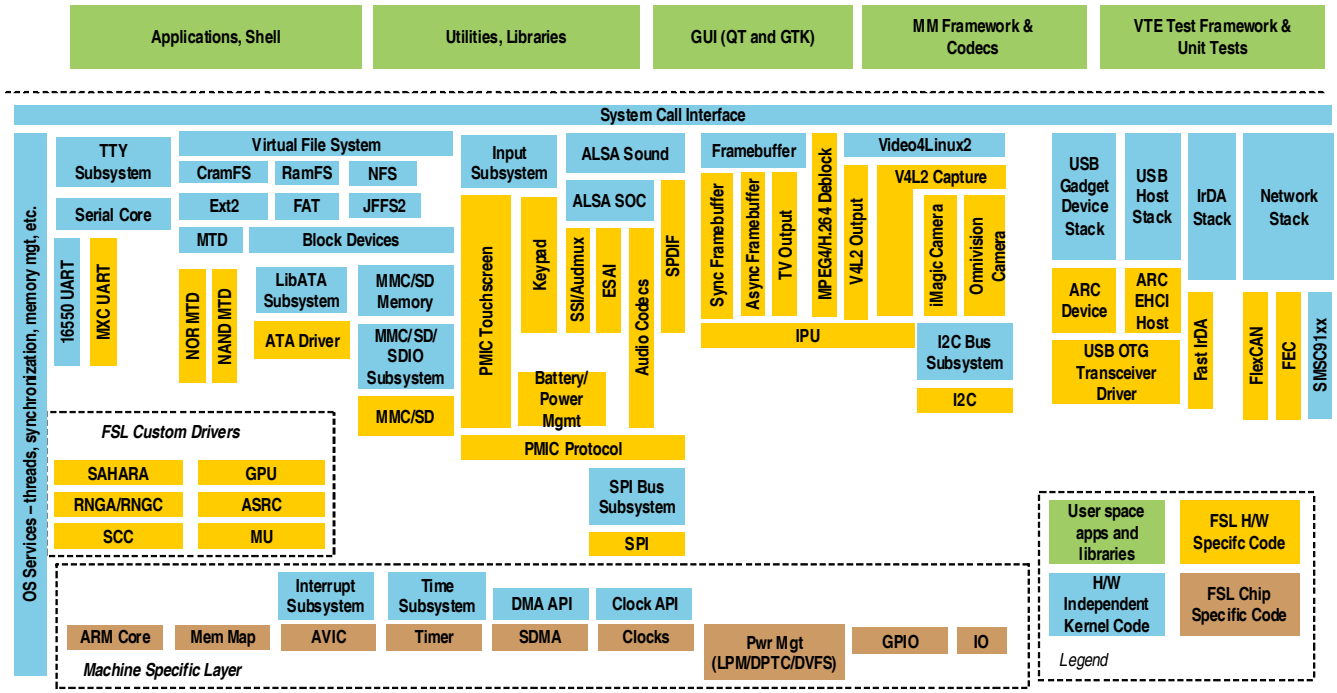

Figure 2-1. BSP Block Diagram

2.2 Kernel

The i.MX Linux port is based on the standard Linux kernel. The kernel supports most of the features available in many 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 MSL implementation.

2.2.1 Kernel Configuration

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

- Embedded mode
- Module loading/unloading
- ARM9
- Supported file formats: 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.MX28	<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx28

See Chapter 3, "Machine Specific Layer (MSL)," for more information.

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 performed through a table structure in the MSL, specific to a particular platform, with each entry specifying a peripheral starting address of virtual addresses, starting address of physical addresses, and the size of the memory region and the type of the region.

2.2.2.2 Interrupts

The standard Linux kernel contains common ARM code for handling interrupts. The MSL contains platform-specific implementations of functions for interfacing the Linux kernel to the ARM9 Interrupt Controller (AITC).

Together, they support the following capabilities:

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

2.2.2.3 General Purpose Timer (GPT)

The GPT is configured to generate an interrupt every 10 ms to provide OS ticks. This timer is also used by the kernel for additional timer events. Linux defines the MSL timer API required for the OS-tick timer and does not expose it beyond the kernel tick implementation. Linux facilitates timer use through various functions for timing delays, measurement, events, and alarms. The GPT is also used as the source to support the high resolution timer feature. The timer tick interrupt is disabled in low-power modes other than idle.

2.2.2.4 DMA API

The i.MX28 device is equipped with two AHB-to-APBH/AHB-to-APBX bridges with built-in DMA capability that allow programmed data transfers between SDRAM and peripheral devices. The DMA is abstracted as a number of channels dedicated to on-chip peripheral devices such as UART, DAC/ADC, GPMI and so on. Each DMA channel is programmed by a set of per-channel registers and special DMA command structure located in memory. A command describes a single DMA transaction and may be chained with other commands. The MSL implements an internal DMA API that allows other drivers to initialize DMA channels and control DMA transfers. The following features are implemented:

- Command structures allocation/de-allocation
- Channel initialization
- Channel execution control: start/stop/freeze a channel
- Channel interrupts control

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, pin multiplexing, 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 Pin Multiplexing

The pin multiplexing component is responsible for setting I/O pin configuration and routing. Each I/O pin is shared between up to three different i.MX28 modules or can be configured as a GPIO pin and controlled

by software. The MSL implements a kernel-space API used by the MSL board specific components to set pins configurations corresponding to a particular board. The following features are implemented:

- Pin resource manager to avoid conflicts on pin use
- Pin voltage control
- Pin strength control
- Pin pull-up resistor control
- Pin group configuration

2.2.2.7 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

Many drivers are 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 Debug Asynchronous Receiver/Transmitter (UART)

The Debug UART driver provides an interface to the i.MX28 Debug UART controller. It provides the standard Linux serial driver API. The following features are supported:

- Interrupt driven transmit/receive of characters
- Standard Linux baud rates up to 115 Kbps
- Receive and transmit FIFOs support
- Transmitting and receiving characters with 5, 6, 7 or 8-bit character lengths
- Odd and even parity
- CTS/RTS hardware flow control
- Send and receive break characters through the standard Linux serial API
- Recognize break and parity errors

- Supports the standard TTY layer IOCTL calls
- Console support required to bring up the command prompt through Debug serial port
- Power management features by suspending and resuming UART ports

Currently, the Debug UART driver is used by default to bring up the console. DMA is not supported by this driver. The Debug UART can be accessed through the `/dev/ttyAM0` device file.

2.3.1.2 Application Asynchronous Receiver/Transmitter (UART)

The Application UART driver provides an interface to the i.MX28 Debug UART controller. It provides the standard Linux serial driver API. The following features are supported:

- Interrupt and DMA driven transmit/receive of characters
- Standard Linux baud rates up to 3 Mb/s
- Transmitting and receiving characters with 5, 6, 7 or 8-bit character lengths
- Odd and even parity
- CTS/RTS hardware flow control
- Send and receive break characters through the standard Linux serial API
- Recognize break and parity errors
- Supports the standard TTY layer IOCTL calls
- Includes console support required to bring up the command prompt through the Debug serial port
- Supports power management features by suspending and resuming UART ports

The application UART can be accessed through the `/dev/ttySP0` device file.

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 DCP

The DCP driver performs AES EBC decryption and encryption using the hardware OTP key that is not accessible from user space. The driver configures the i.MX28 DCP engine to AES 128-bit EBC mode and only supports encrypting/decrypting of a single 128-bit block.

The main purpose of this driver is to implement an interface to the DCP cryptography engine which is necessary for boot stream image verification performed before writing the boot stream to NAND flash. The driver implements a simple IOCTL interface to decrypt and encrypt a single 128-bit block.

2.3.5 i.MX28 Graphics

The graphics component consists of a number of Linux kernel drivers that implement the standard Linux kernel interface to the i.MX28 hardware to manipulate video buffers and output them to an LCD panel or TV screen. The graphic support includes the following components:

- Frame buffer driver
- LCDIF driver
- Pixel Pipeline (PxP) driver
- LCD panel driver

Figure 2-2 shows a block diagram of the i.MX28 Linux kernel graphic components and their relationship to each other.

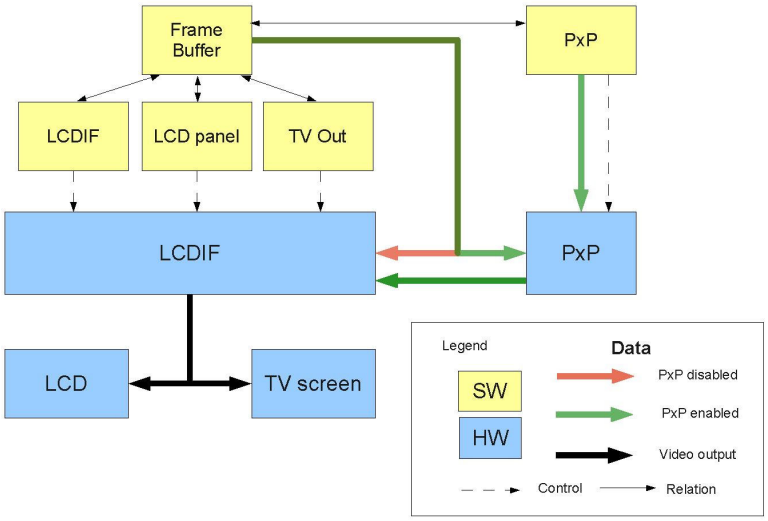

Figure 2-2. i.MX28 Kernel Graphic Components

2.3.5.1 LCDIF Driver

The i.MX28 LCDIF driver implements the Linux kernel-space API for basic LCD interface operations such as initialization, as well as LCD interface DMA abstraction for the callers. The interface is used by other graphics components such as the LCD panel drivers or the Frame buffer driver.

2.3.5.2 LCD Panel Drivers

LCD panel drivers provide an abstraction of a video output device for the Frame buffer driver. The LCD panel driver implements specific LCDIF initialization and exposes a set of API calls to the frame buffer driver so that it can control video output devices and perform dynamic switching between them (for example, run-time switching between the LCD panel and TV-output).

2.3.5.3 Frame Buffer Driver

The Frame buffer driver implements a standard Linux fbdev interface for user space applications and controls dynamic switching between different video outputs per user request.

2.3.5.4 Pixel Pipeline (PXP) Driver

The PxP driver implements a Video for Linux (V4L2) interface to the i.MX28 PxP hardware capable of performing various manipulations with video buffers such as scaling, cropping, rotation, alpha blending and so on. The PxP module handles a video stream received from user space from the V4L interface, then combines it with the frame buffer image and outputs the final image to the LCDIF module.

The graphics components can operate in two modes, with PxP enabled or disabled. Figure 2-2 shows the different video data flows depending on different modes.

2.3.6 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.

2.3.7 Keypad

The keypad driver interfaces Linux to the keypad ladder connected to the i.MX28 LRADC controller. The software operation of the driver follows the Linux keyboard architecture.The driver is driven by interrupts generated by the LRADC controller when changing a signal on the keypad ladder input pin. The driver

reads a current voltage on the LRADC pin, detects which key is being pressed and sends a key code to the upper layer. The driver detects long key presses and reports them as multiple key press events.The keypad driver may be used as a wake-up source for low-power standby mode.

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 shows the MTD architecture.

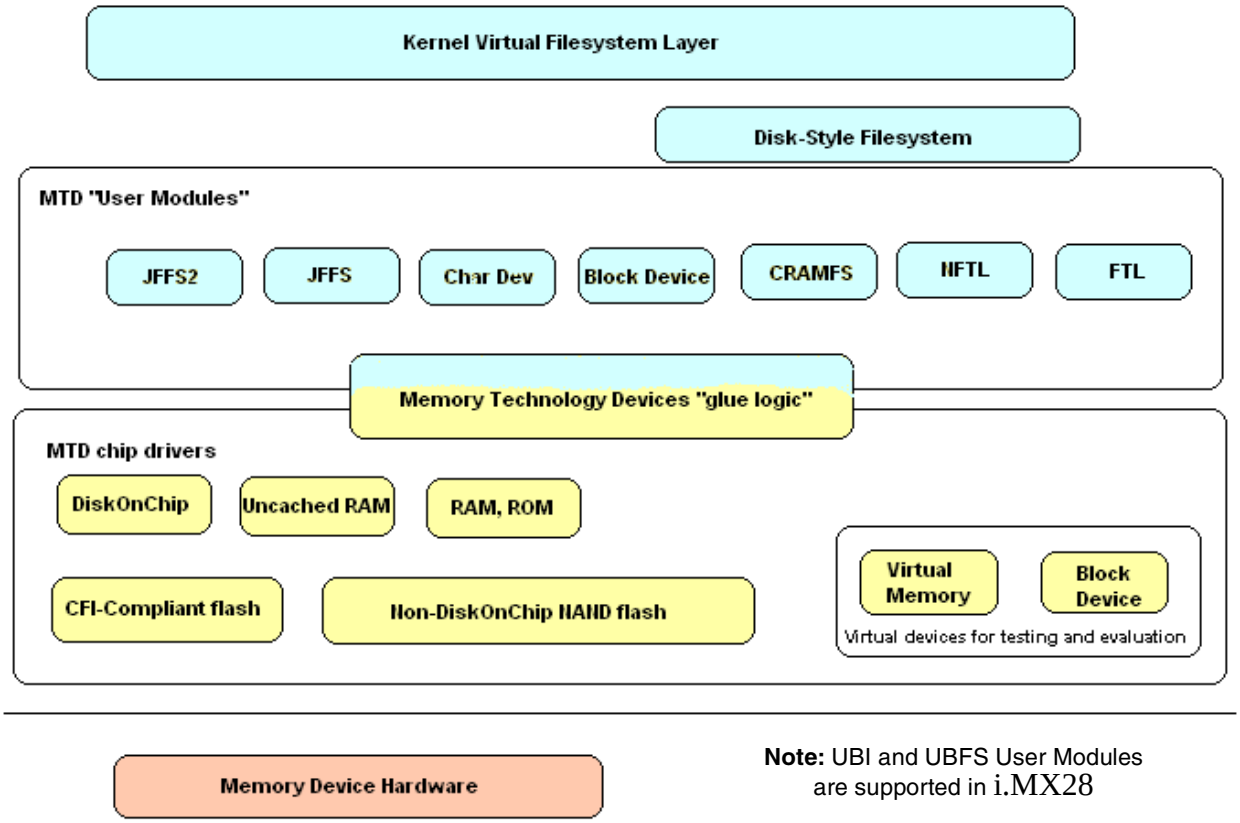

Figure 2-3. MTD Architecture

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 GPMI/NAND

The GPMI/NAND driver interfaces with the i.MX28 GPMI/NAND module that is able to interact with a variety of NAND flash chips with 2 Kbyte and 4 Kbyte page sizes. The driver implements a standard interface for the upper MTD subsystem layer and supports various file systems, such as JFFS2, UBIFS or different commodity file systems (for example, FAT or EXT2) created on top of the UBI FTL.

The GPMI/NAND driver supports the i.MX28 BCH HW Error Correcting Code (ECC) engine that speeds up NAND flash read and write operations

2.3.9 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.

2.3.9.1 USB Host-Side API Model

Within the Linux kernel, host-side drivers for USB devices talk to the usbcore APIs. The 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. See Chapter 2 of http://www.kernel.org/doc/htmldocs/usb.html, for more information.

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.

2.3.9.2 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.

2.3.9.3 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 after the product has been distributed, so driver bugs cannot normally be fixed if they are found after shipment.

Additional changes are required 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 required inside usbcore, so that it can identify OTG-capable devices and respond appropriately to HNP or SRP protocols.

2.3.10 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

2.3.10.1 MMC/SD Host Driver

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

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

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

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

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

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

2.3.10.3 SPI Bus Driver

This low-level SPI module provides an interface to the i.MX28 SSP interface configured to work in SPI master mode. The driver implements standard kernel space API for the Linux SPI core driver that implements a kernel-space interface for other drivers for various SPI devices, such as SPI Ethernet Controller or Power Management Interface Controller (PMIC).

The i.MX28 implements a single DMA channel for SSP interface which does not allow full-duplex bidirectional transfers over the SPI bus. This limitation should be taken into account when developing drivers for SPI devices located on the i.MX28 based boards.

Both DMA and byte-to-byte transfers are supported.

2.3.10.4 Dynamic Power Management (DPM) Driver

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

- Operating points
- Operating states
- Policies
- Policy manager

2.3.10.4.1 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. After a DPM system is initialized and activated, the system is always executing a particular DPM policy.

2.3.10.4.2 Operating Points

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

2.3.10.4.3 Operating States

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

2.3.10.4.4 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 required, then a policy manager must exist in the system to coordinate the activation of different policies.

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

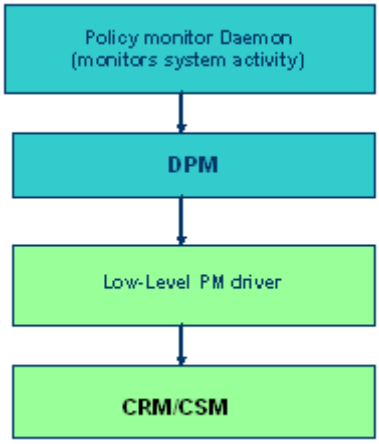

Figure 2-4. DPM High Level Design

Figure 2-5 shows the DPM architecture block diagram.

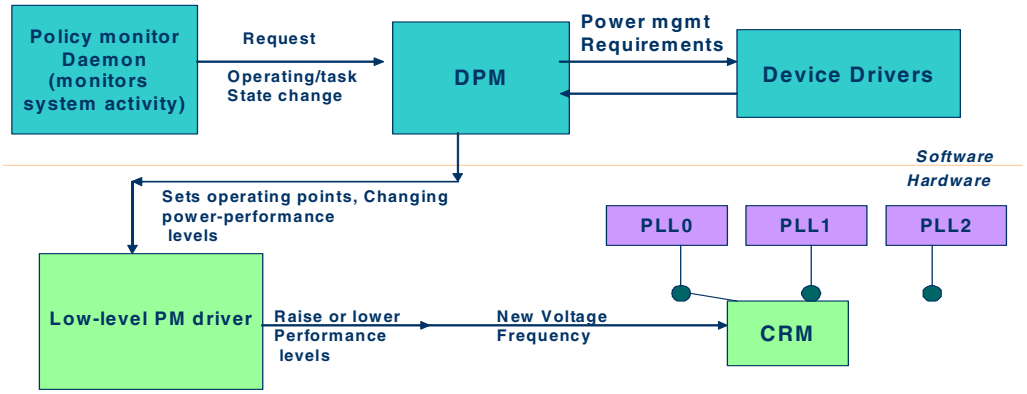

Figure 2-5. DPM Architecture Block Diagram

2.3.10.5 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 performed 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.

2.3.10.6 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.

2.3.10.7 Backlight Driver

The backlight driver implements a standard Linux kernel-space interface for a Linux kernel backlight core driver that, in turn, exposes LCD backlight control interface to user space applications by sysfs.

The backlight driver controls the LCD backlight though the i.MX28 PWM modules connected either directly to the LCD panel backlight LED or to the intermediate backlight controller that sets backlight LED brightness based on input PWM signal. The LCD panel driver implements a LCD specific part of backlight control which is registered with the i.MX28 backlight driver. See Section 2.3.5, “i.MX28 Graphics,” for more details about the LCD panel drivers

2.3.10.8 LED Driver

The LED driver controls on-board LEDs connected to the i.MX28 PWM module. The LED driver implements a standard interface that is exposed to user space applications by sysfs and other kernel drivers though the kernel space API, which may use LEDs to warn about different events, such as timer ticks or MMC data transfers.

2.3.10.9 Power Source Manager and Battery Charger

Power Source Manager and Battery charger drivers controls the i.MX28 power supply module. The i.MX28 may be powered from different power sources that include:

- 5 V wall power supply
- 5 V USB
- Li-Ion 3.7 V battery

Regardless of the power input, the power supply supplies voltage to several output voltage rails intended to power various on-chip and on-board components, such as ARM CPU core, SDRAM, peripheral I/O devices and so on. The way that these output voltages are generated depends on which power source is used. When the device is powered from a 5 V source, it uses internal voltage regulators to convert input voltage. When the device is powered from a battery source, it uses on-chip DC-DC converters. Certain software operations are required during transition from one power source to another, for which the power source manager driver is responsible. Also the power source manager notifies other drivers about power source changes.

Thei.MX28 power supply contains a built-in battery charger module capable of charging Li-Ion batteries. The battery charger driver implements a state machine that controls charging current and protects the battery from damage caused by under or overcharging.

Both drivers are implemented in a single standalone module and do not expose any interfaces to other kernel or userspace components except subscribing for different events detected by the drivers.

2.3.10.10 CPUFreq Driver

The CPUFreq driver is built on top of the voltage regulators and clock framework and implements a set of operating points that define clock speed of CPU, SDRAM and AHB bus along with appropriate CPU voltage value. The CPUFreq driver is plugged into Linux kernel CPUFreq subsystem that, in turn, implements a set of different policies (governors) that control transitions between different operating points.

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:

- 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 i.MX28 Boot Loader

For the i.MX28, some boot loader functionality is delegated to the built-in ROM firmware that is capable of loading a boot stream image containing the Linux kernel from different locations. The boot stream, in turn, implements hardware initialization and an interface to the Linux kernel. Since the i.MX28 built-in ROM is entirely implemented in hardware, it is not described in this document.

The i.MX28 boot image may contain the following bootlets implementing general boot loader functions:

- Boot prep
- Linux prep
- U-boot loader

Figure 2-6 shows block diagrams of two boot stream images.

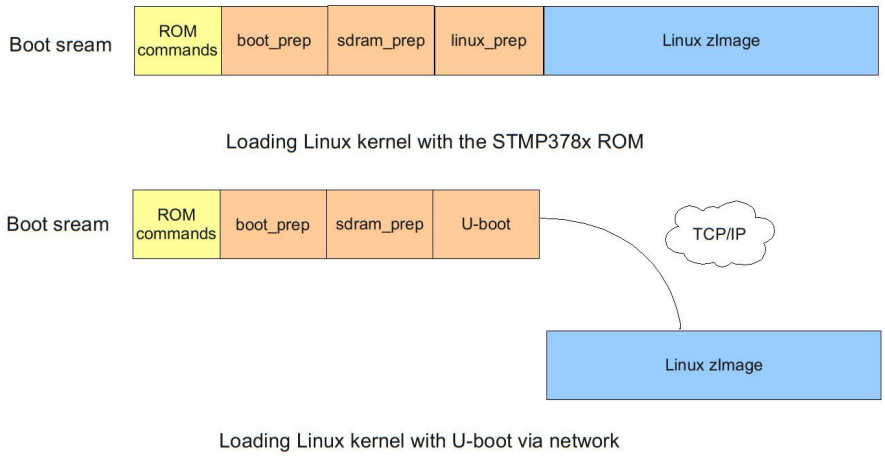

Figure 2-6. i.MX28 Boot Stream

2.4.1.1 Boot Prep

The boot prep bootlet implements basic power supply, EMI controller initialization and clock initialization necessary to start the Linux kernel.

2.4.1.2 Linux Prep

This component provides a standard interface between ARM Linux kernel and boot loader, including:

- Generating a list of ARM tags containing necessary information, such as SDRAM size, ARM CPU and machine identification and Linux kernel command line.
- Jumping to the Linux kernel that has already been downloaded to SDRAM by the i.MX28 ROM firmware.

2.4.1.3 U-boot

U-boot is an open source universal boot loader for various embedded platforms including ARM, PowerPC, MIPS and so on. For the i.MX28, U-boot is used to load Linux kernel image to SDRAM over a network connection because the i.MX28 built-in ROM firmware does not implement a TCP/IP network stack.

The i.MX28 U-boot port implements a driver for the built-in FEC ethernet controller used to transfer data over TCP/IP network.

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

These modules are normally available in the following directory:

```
<litb_dir>/rpm/BUILD/linux/arch/arm/mach-mx28 for imx28 platform
```

The header files are implemented under the following directory:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxs/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. Each of the following sections contains an overview of the hardware operation. For more information, see the corresponding device documentation.

3.1 Interrupts

The i.MX28 uses an Interrupt Collector module. The following sections explain the hardware and software operation for the interrupts.

3.1.1 Interrupt Hardware Operation

The Interrupt Collector module controls and prioritizes a maximum of 128 internal and external interrupt sources. Each source can be enabled and disabled by configuring the ENABLE bit in the dedicated Hardware Interrupt Collector Interrupt register. When an interrupt source is enabled and the corresponding interrupt source is asserted, the Interrupt Collector asserts a normal or a fast interrupt request to the ARM core depending on the ENFIQ bit value in the dedicated Hardware Interrupt Collector Interrupt register.

The Interrupt Collectors interrupt requests are prioritized in the order of fast interrupts and normal interrupts in order of highest priority level. There are four normal interrupt levels, with zero level being the lowest priority. The interrupt levels are configurable through the PRIORITY bits of the Hardware

Interrupt collector Interrupt register. Only in supervisor mode can the Interrupt Collector registers be accessed. A number of IRQ sources can be expanded by using GPIO lines to assert interrupts.

3.1.2 Interrupt Software Operation

In ARM based processors, normal interrupt and fast interrupt are two different exceptions. The exception vector addresses can be configured to start at a low address (0x0) or at a high address (0xFFFF0000). The ARM Linux implementation chooses the high vector address model. The following file has a detailed description about the ARM interrupt architecture:

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

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

3.1.3 Interrupt Source Code Structure

The MSL interrupt layer is implemented in the source files shown in Table 3-1, located in the directories indicated at the beginning of this chapter:

Table 3-1. Interrupt Files List

File	Description
icoll.c	Interrupt manipulation functions
irqs.h	Interrupt source numbers
regs-icoll.h	Interrupt Collector registers
entry-macro.S	Interrupt source detection

3.1.4 Interrupt Programming Interface

The Machine Specific Layer implementation exports a single function that initializes the Interrupt Collector and register interrupt manipulation routines for each interrupt source in the system. This performs with the structures `irq_chip` and `mxs_gpio_chip` of the `irq_chip` type that contain functions to enable, disable, and acknowledge interrupt sources.

The `irq_chip` is associated with i.MX28normal 128 interrupt sources while `mxs_gpio_chi` is used for external GPIO interrupts. Each interrupt source is associated with one of the `irq_chip` structures with the `set_irq_chip` call. After initialization, the interrupt can be used by the drivers through the `request_irq()` and `free_irq()` functions to register device-specific interrupt handlers. Upon receiving the interrupt, the interrupt code uses `get_irqnr_and_base` to detect the interrupt source, acknowledges the interrupt using the registered `irq_chip` structure set by the MSL, and calls the registered device-specific interrupt handler. Depending on the flags passed to the `request_irq` function, the code may disable the interrupt using an `irq_chip` call before executing the device-specific handler.

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). After 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 consists of four 32-bit 32 KHz timers.

3.2.1 Timer Hardware Operation

Each of the four timers consists of a 1632-bit fixed count value and a 1632-bit free-running count value. In most cases, the free-running count decrements to 0. When it decrements to 0, it sets an interrupt status bit associated with the counter, which causes:

- If the RELOAD bit is set to 1, the count is automatically copied to the free-running counter and the count continues
- If the RELOAD bit is not set, the timer stalls when it reaches 0

Each timer has an UPDATE bit that controls whether the free-running-counter is loaded at the same time that the fixed-count register is written from the CPU. The output of each timer's source select has a polarity control that allows the timer to operate on either edge. The timers have multiple clock sources that include the PWM output signals and the on-chip 32 KHz XTAL that, in turn, can be programmed to 32 KHz, 8 KHz, 4 KHz or 1 KHz timer update cycles.

Each of the four times have compare match register. When free-running counter equal match value, it issue a interrupt.

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-mxs/timer-match.c` file.

3.2.5 Timer Programming Interface

The timer module utilizes four hardware timers, to implement clock source and clock event objects. This is done with the mxs_clocksource structure of `struct clocksource` type and mxs_clockevent structure of `struct` mxs_clockevent type. Both structures provide routines required for reading current timer values and scheduling the next timer event. The module implements a timer interrupt routine that services the Linux OS with timer events for the purposes mentioned in the beginning of this chapter.

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-mx28/ mx28evk.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 `mx28evk.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-mxs/include/mach
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-imx
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-
```

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

3.3.5 Memory Map Programming Interface

The Memory Map is implemented in the `mx28evk.c` file to provide the map between physical and virtual addresses. It defines an initialization function to be called during system startup.

3.4 Pin Multiplexing

The i.MX28 implements a flexible pin multiplexing mechanism that permits using the same SoC I/O pins for different purposes depending on the board hardware configuration. The following section describes the Pin Multiplexing software and hardware operation

3.4.1 Pin Multiplexing Hardware Operation

The i.MX28 SoC implements 120 digital interface pins divided into four banks. The first three banks implement multiplexed pins where each pin can be routed up to three different modules or serve as GPIO. The fourth bank implements EMI pins which are not multiplexed.

The pin control interface has the following features:

- All digital pins have selectable output drive strengths
- All EMI pins have 1.8/2.5 V and 3.3 V selects
- Several digital pins can be programmed to enable pull up resistors

3.4.2 Pin Multiplexing Software Operation

The MSL contains board specific files that define I/O pin routing and provide functions for device drivers to set up pin routing during the initialization stage. These mechanisms allow board-independent drivers where all board-specific details are hidden within the MSL. The pin multiplexing implements a pin resource manager intended to prevent conflicting access to shared I/O pins by different device drivers.

3.4.3 Pin Multiplexing Source Code Structure

The MSL Pin Multiplexing layer is implemented in the directories listed at the beginning of this chapter. The files are listed in Table 3-4.

Table 3-4. Pin Multiplexing Source Files

File	Description
mx28_pinsh	I/O pins definitions
pinctrl.c	Pin Multiplexing API implementation

3.4.4 Pin Multiplexing Programming Interface

The MSL Pin Multiplexing module provides a kernel-space internal MSL interface to control I/O pins. This interface is not exposed to other device drivers or kernel components. The interface indirectly sets up pin configuration through driver-specific callbacks implemented by the MSL. Board-specific details are hidden for easier driver migration.

The Pin Multiplexing API defines the following structures and functions:

`enum pin_fun, enum pin_strength, enum pin_voltage`

Define pin routing and configuration.

`struct pin desc, struct pin group`

Describe a group of pins.

`int mxs_request_pin(unsigned id, enum pin_fun fun, char *label)`

Request access to a pin. The label should be used later to configure pin parameters.

`void mxs_release_pin(unsigned id, char *label)`

Release the pin.

`int mxs_request_pin_group(struct pin_group *pin_group, char *label)`

Request access to a group of pins.

`void mxs_release_pin_group(struct pin_group *pin_group, char *label`

Release pin group.

`void mxs_pin_strength(unsigned id, enum pin_strength strength, char *label)`

Set pin output strength.

`void mxs_pin_voltage(unsigned id, enum pin_voltage voltage, char *label)`

Set pin output voltage.

`void mxs_pin_pullup(unsigned id, int enable, char *label)`

Control pull up resistor of a pin.

3.4.5 GPIO With Pin Multiplexing

The Pin Multiplexing module allows routing multiplexed pins to the general purpose input/output module that provides an API to configure pins and a central place to configure GPIO interrupts. Once the i.MX28 pin is routed to the GPIO module, this pin can be manually configured by a set of the pin multiplexing registers dedicated to the GPIO module. These registers allow setting pin direction (input or output), pin output value, and pin configuration as an interrupt source by specifying an interrupt trigger mode (edge or level, high or low).

Each Linux kernel driver or subsystem can request an external pin to be configured as GPIO and then control the pin state using a kernel-space standard Linux GPIO API. The GPIO pins are handled with the standard GPIO API as documented in `Documentation/gpio.txt`. The MSL GPIO module implementation is contained in the `gpio.c` and `gpio.h` files in the directories indicated at the beginning of this chapter.

Chapter 4
Direct Memory Access Controller (DMAC) API

The Direct Memory Access Controller (DMAC) provides 16 channels supporting linear memory, 2D memory, and FIFO transfers to provide support for a wide variety of DMA operations.

4.1 Hardware Operation

The i.MX28 device is equipped with two AHB-to-APBH/AHB-to-APBX bridges with built-in DMA capability that allows programmed data transfers between SDRAM and peripheral devices. The DMA is abstracted as a number of channels dedicated to on-chip peripheral devices such as UART, ADC/DAC, GPMI and so on. Each DMA channel is programmed by a set of per-channel registers and a special DMA command structure located in memory. A command describes a single DMA transaction and can be chained with other commands to set up multiple DMA transfers.

Each DMA channel implements a semaphore used to start and stop the DMA channels. The semaphore may contain values from 0 to 255 that are set by software. The DMA channel starts transferring data on writing a semaphore value greater than zero and continues operation until the semaphore is decremented to zero or an error occurs. The semaphore is decremented after completion of a single DMA transfer if the corresponding flag is set within the command structure.

The DMA channel may generate interrupt events on command completion or on an error. This is configurable through a set of DMA channel registers.

The DMA includes the following features:

- Sixteen channels support linear memory, 2D Memory, and FIFO for both source and destination
- DMA chaining for variable length buffer exchanges and high allowable interrupt latency requirement
- Increment, decrement, and no-change support for source and destination addresses
- Each channel is configurable to response to any of the DMA request signals
- Supports 8, 16, or 32-bit FIFO and memory port size data transfers
- DMA burst length configurable up to a maximum of 16 words, 32 half-words, or 64 bytes for each channel
- Bus utilization control for the channel that is not triggered by a DMA request
- Burst time-out errors terminate the DMA cycle when the burst cannot be completed within a programmed time count
- Buffer overflow error terminates the DMA cycle when the internal buffer receives more than 64 bytes of data
- Transfer error terminates the DMA cycle when a transfer error is detected during a DMA burst

4.2 Software Operation

Prior to using a DMA channel, the driver should register an interrupt handler for interrupts generated by the DMA channel in order to receive DMA error or completion events.

The most used scenario of DMA operation is when a device driver wants to transfer a number of bytes to or from a memory buffer located on SDRAM. First, it allocates and initializes a DMA command structure or a list of command structures for multiple transfers. Then it resets the DMA channel and configures the channel registers to point to a command structure for the first DMA transfer. When all the required initialization is done, the DMA channel is started by setting a DMA channel semaphore.

The module provides an API for other drivers to control DMA channels. The DMA software operations are as follows:

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

4.3 Source Code Structure

The header file, `dmaengine.h`, is available in the directory:

`arch/arm/plat-mxs/include/mach/`

Table 4-1 lists the source files available in the directory, `arch/arm/plat-mxs/`

Table 4-1. DMA API Files

File	Description
`dma-apbh.c, dma-apbx.c`	Parameters of DMA channels
`dmaengine.c`	DMA API functions

4.4 Programming Interface

The module implements custom DMA API. Standard API is not supported. Refer to the doxygen files in the release notes for more information on the methods implemented in the driver.

Chapter 5
Persistent Bits Driver

Persistent bits refers to a small number of registers that persist over power cycles.

5.1 Hardware Operation

The persistent bit block uses persistent storage and resides in a special power domain (crystal domain) that remains powered up even when the rest of the device is in its powered-down state. Six 32-bit persistent bit registers. They are as below:

- HW_RTC_PERSISTENT0—holds bits used to configure various hardware settings
- HW_RTC_PERSISTENT1—holds bits related to the ROM and redundant boot handling
- HW_RTC_PERSISTENT2–5—general purpose use

5.2 Software Operation

The persistent bit support code is implemented as a user-space accessible API, but with the configuration of the bits done by the board setup code. The configuration structures map the name of a bit-field to a part of a 32-bit hardware register; for example:

```
{ .reg = 1, .start = 1, .width = 1, .name = "NAND_SECONDARY_BOOT" }
```

declares that the name NAND_SECONDARY_BOOT is mapped to the HW_RTC_PERSISTENT1 register, starting at bit 1, having a width of 1 bit (a single bit register).

User space accesses the persistent bits by sysfs device attributes in the `/sys/devices/platform/`mxs-persistent.0 directory. Access is done by reading and writing the attribute files.

For example, to read:

```
# cat sys/devices/platform/mxs-persistent.0/NAND_SECONDARY_BOOT
0
#
```

To write:

```
# echo -n 1 > sys/devices/platform/mxs-persistent.0/NAND_SECONDARY_BOOT
#
```

5.3 Source Code Structure

The persistent bit driver code listed in Table 5-1, is located in:

`arch/arm/mach-mx28/include/mach/`

`arch/arm/mach-mx28`

`drivers/misc/`

Table 5-1. Persistent Bits Driver Files

File	Description
`mx28.h`	Device configuration structures
`devices.c`	Device configuration
`mxs-persistent.c`	Driver file

5.4 Menu Configuration Options

The persistent bit driver is unconditionally compiled into the kernel image.

5.5 Programming Interface

The kernel persistent bit API is defined by means of the following structures to facilitate persistent bit configuration.

```
struct mxs_persistent_bit_config {
    int reg;
    int start;
    int width;
    const char *name;
};

struct mxs_platform_persistent_data {
    const struct mxs_persistent_bit_config *bit_config_tab;
    int bit_config_cnt;
};
```

The structure `mxs_persistent_bit_config` defines a single bit that always lies in a single hardware 32-bit register. The structure `mxs_platform_persistent_data` contains all of the persistent bit definitions which are valid for the given board.

Chapter 6
Unique ID on Boot Media

The i.MX28 Unique ID (UID) storage feature allows customers to keep a limited sequence of bytes in a secured place such as:

- One-Time-Programmed (OTP) bits

6.1 Software Operation

The Unique ID module provides a sysfs interface to end users. When the module is started, a new sys entry is created: `/sys/uid`. It contains one or more subdirectories that match the UID provider registered in the system. In turn, each of the subdirectories contains files `id` and `id.bin`. These files can be read from the user space and written to with root privileges. Data that is read from or written to `id` is in human-readable form, while `id.bin` provides access to the raw binary data. The UID provider can enable access to `id`, `id.bin` or both.

Before a UID value can be written, the module must be unlocked. This is achieved by writing 1 to the file `/sys/modules/unique-id/parameters/unlock`. The access is limited to three minutes. After three minutes, the module is locked and must be enabled again.

The nature of UID storage forces some limits and assumptions:

- For OTP—bits can be written only once and the user has access to only three long words ($3 \times 32 = 96$ bits) of data

6.2 Programming Interface

A provider shall register the table of functions using a call to:

```
uid_provider_init(char *name, struct uid_ops *ops, void *context).
```

This function registers the table `ops` as a new UID provider with name `name`. When finished, the provider should be unregistered using a call to:

```
uid_provider_remove(char *name)
```

It completely removes the UID provider from the system.

The structure `uid_ops` contains two pointers to functions `id_show` and `id_store`. Both of these functions follow the conventions for attribute accessors, except for the added first parameter `void *context`, which is passed to `uid_provider_init`.

6.3 Source Code Structure

The Unique ID module code listed in Table 6-1, is located in:

`arch/arm/plat-mxs/include/mach/`

arch/arm/plat-mxs/

Table 6-1. Unique ID Files

File	Description
unique-id.c	Generic UID code
unique-id.h	Header with function prototypes
otp.c	Implementation of OTP UID provider

6.4 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_MXS_UNIQUE_ID = y
- CONFIG_MXS_UNIQUE_ID_OTP = y

Chapter 7
CPU Frequency Scaling (CPUFREQ) Driver

The CPU frequency scaling device driver allows the clock speed of the CPU to be changed on the fly. Once the CPU frequency is changed, the voltages VDDD, VDDD_BO, VDDIO, and VDDA are changed to the voltage value defined in `profiles[]`. This method can reduce power consumption (thus saving battery power), because the CPU uses less power as the clock speed is reduced.

7.1 Software Operation

The CPUFREQ device driver is designed to change the CPU frequency and voltage on the fly. If the frequency is not defined in `profile[]`, the CPUFREQ driver changes the CPU frequency to the nearest frequency in the array. The CPU frequency 64 MHz and below in the `array profiles[]` can be changed only if both USB clock usage and LCD clock usage are zero. The frequencies are manipulated using the clock framework API, while the voltage is set using the regulators API. By default, the userspace CPU frequency governor is used with CPU frequency, which can be changed manually. To change CPU frequency automatically, the conservative CPU frequency governor can be used. Refer to the API document for more information on the functions implemented in the driver (in the doxygen folder of the documentation package).

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

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

To change the CPU frequency to a value that is given by using the command above (for example, to 392.727 MHz) use this command:

```
echo 392727 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

The frequency `392727` is in KHz, which is `392.727` MHz.

The maximum frequency can be checked using this command:

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

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

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

Use the following command to change to conservative CPU frequency governor:

```
echo conservative > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

7.2 Source Code Structure

Table 7-1 shows the source files and headers available in the following directory:

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

Table 7-1. CPUFREQ Driver Files

File	Description
cpufreq.c	CPUFREQ functions

7.3 Menu Configuration Options

The following Linux kernel configuration is provided for this module:

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

7.3.1 Board Configuration Options

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

Chapter 8
i.MX28 Static Power Management Driver

Static Power Management refers to the system power management states set according to the operating mode, as opposed to the dynamic power management where the state is changing according to the given limitations, based on parameters such as system load. Static power management in Linux usually refers to the power saving states. Linux power states are:

- Standby/power-on suspend (standby)
- Suspend-to-RAM (mem)
- Suspend-to-disk (disk)

Refer to `Documentation/power/states.txt` within the Linux kernel source tree for more information on these states. Within the i.MX28 BSP only the standby state is supported.

8.1 Hardware Operation

Standby state, which is also sometimes referred to as Wait for Interrupt (WFI) mode, is entered when the corresponding ARM co-processor instruction (`mcr p15, 0, r0, c7, c0, 4`) is executed. The i.MX28 also has an additional feature for more power saving in WFI mode, called INTERRUPT_WAIT mode. This mode is activated by setting a 1 in the INTERRUPT_WAIT bit of the CLKCTRL_CPU register. This activation should be performed prior to WFI command execution. The coprocessor instruction sequence enables an internal gating signal. This signal triggers the write buffers drain and guarantees that the CPU is in the idle state. With the INTERRUPT_WAIT bit is set, after the WFI command execution, the CPU halts on the `mcr` instruction. When an interrupt or a FIQ occurs, the mcr instruction completes and the IRQ/FIQ handler is entered normally.

8.2 Software Operation

The standby state is implemented within the i.MX28 BSP to minimize the power consumption as much as possible. Before issuing the WFI instruction, the following preparation steps are done:

- Interrupts are disabled except for those that are wakeup sources
- DMA is disabled
- CPU is switched to bypass mode (direct clocking from crystal)
- RAM is switched to bypass mode and put into self-refresh
- PLL is switched off; Xtal oscillator is switched on
- INTERRUPT_WAIT bit is set in the CPU Clock Control register (CLKCTRL_CPU)

The wakeup sources and the system state can be set by the sysfs interface. To activate a wakeup source, write 1 to `/sys/bus/platform/devices/<device>/power/wakeup.`

For example:

```
# echo 1 > /sys/bus/platform/devices/mxs-duart.0/power/wakeup
```

To put the entire system into standby mode, run the following command:

```
# echo standby > /sys/power/state
```

8.3 Source Code Structure

The platform-specific static power management code listed in Table 8-1, is located in `arch/arm/mach-mx28/.`

Table 8-1. Power Management Driver Files

File	Description
`pm.c`	High level code interfacing with the platform-independent static power management API
`sleep.S`	Assembly code implementing the low-level part of standby mode
`sleep.h`	Header file containing definitions and structures

8.4 Menu Configuration Options

The following Linux kernel configurations are provided for this driver:

- CONFIG _PM [=Y]

  Generic configuration option to enable static power management. Once it is enabled, the source files listed above are automatically selected for compilation.

Chapter 9
Frame Buffer Driver

The frame buffer driver is designed using the Linux kernel frame buffer driver framework. It implements the platform driver for a frame buffer device. The implementation uses the LCDIF API for generic LCD low-level operations. The LCD driver is organized in a flexible and extensible manner and is abstracted from any specific LCD panel support. To support a LCD panel, implement the low-level panel handling functions and pass the container structure to the frame buffer driver through `platform_data`. See Chapter 10, "LCD Interface (LCDIF) Driver," for more details on the panel handling interface.

9.1 Hardware Operation

The frame buffer driver uses the LCDIF API to interact with the hardware.

9.2 Software Operation

A frame buffer device is a memory device similar to `/dev/mem` and it has the same features. It can be read from, written to, or some location in it can be seeked and `mmap()`. The difference is that the memory that appears is not the whole memory, but only the frame buffer of the video hardware. The device is accessed through special device nodes, usually located in the `/dev` directory, `/dev/fb*`. `/dev/fb*` also has several IOCTLs which act on it, by which information about the hardware can be queried and set. The color map handling operates through IOCTLs as well. See `linux/fb.h` for more information on what IOCTLs exist and which data structures they use.

The frame buffer driver implementation for i.MX28 is abstracted from the actual hardware. It operates as an arbiter, picking up the panel driver that matches the resolution and bit width selected and calling this panel driver functions. All the panel driver structures are linked into a list that is passed to the frame buffer driver through the `platform_data` parameter of the frame buffer platform device.

The default panel driver is picked up by name (using the `mxs_lcd_iterate_pdata` iterator function) during probing, based on the `lcd_panel` command line parameter passed through the kernel command line (in the case of the frame buffer driver compiled into the kernel) or during module probe (in the case of the frame buffer driver compiled as a module). Later on, if another screen resolution and/or bit width is requested, the frame buffer driver looks through the list of available panel drivers in order to find the one that supports this resolution and bit width. Once found, the frame buffer driver switches to the new panel driver.

9.3 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_FB_MXS_43WVF1G [=Y|N]
  Configuration option to compile support for the SEIKO 4.3' WVGA(800x480) LCD panel into the kernel.

9.4 Source Code Structure

The frame buffer driver source code is in `drivers/video/mxs/mxsfb.c`.

The panel support code is located in `drivers/video/mxs`.

The frame buffer driver includes the source/header files shown in Table 9-1.

Table 9-1. Frame Buffer Driver Files

File	Description
lcd_43wvf1g.c	Supports SEIKO 4.3’ WVF1G 800x480 LCD panel

Chapter 10
LCD Interface (LCDIF) Driver

The LCD Interface (LCDIF) driver supports the SEIKO 4.3” WVF1G LCD panels.

10.1 Hardware Operation

The LCDIF driver includes the following features:

- Bus master and DMA operating modes for LCD writes requiring minimal CPU overhead
- 8/16/18/24-bit LCD data bus support depending on the package size
- Programmable timing and parameters for system, VSYNC and DOTCLK LCD interfaces to support a wide variety of displays
- ITU-R BT.656 mode (Digital Video Interface (DVI) mode) including a progressive-to-interlace feature and RGB to YCbCr 4:2:2 color space conversion to support 525/60 and 625/50 operations

10.2 Software Operation

The LCDIF support code is implemented as a kernel space API. The API routines implements basic LCD interface operations, such as initialization, as well as LCD DMA abstraction for the callers. The interface is meant to be used by kernel entities such as LCD panel drivers and frame buffer drivers.

The LCDIF headers also define the LCD panel structure. This structure encapsulates data and functions to provide flexible support for different LCD panels that can be attached to the LCD interface.

10.3 Source Code Structure

The LCDIF source code, shown in Table 10-1, is located in the following directories:

```
drivers/video/mxs/
arch/arm/mach-mx28/include/mach/
```

Table 10-1. LCDIF Driver Files

File	Description
`lcdif.h`	Header file
`regs-lcdif.h`	Register definitions
`lcdif.c`	Source code file

10.4 Menu Configuration Options

The LCDIF API is unconditionally compiled into the kernel image. The LCD panel can be selected by the kernel command line options:

- `lcd_panel = 43wvf1g` for SEIKO 43WVF1G

NOTE

If there is one panel, then do not need to pass this parameter.

10.5 Programming Interface

The following structures are defined to facilitate flexible LCD panel support:

```
struct mxs_platform_fb_entry {
    char name[16];
    u16 x_res;
    u16 y_res;
    u16 bpp;
    u32 cycle_time_ns;
    int lcd_type;
    int (*init_panel)(struct device *dev,
    dma_addr_t phys, int memsize,
    struct mxs_platform_fb_entry *pentry);
    void (*release_panel)(struct device *dev,
    struct mxs_platform_fb_entry *pentry);
    int (*blank_panel)(int blank);
    void (*run_panel)(void);
    void (*stop_panel)(void);
    int (*pan_display)(dma_addr_t phys);
    int (*update_panel)(void *p,
    struct mxs_platform_fb_entry *pentry);
    struct list_head link;
    struct mxs_platform_bl_data *bl_data;
};

struct mxs_platform_fb_data {
    struct list_head list;
    struct mxs_platform_fb_entry *cur;
};

struct mxs_platform_bl_data {
    struct list_head list;
    int bl_gpio;
    int bl_max_intensity;
    int bl_default_intensity;
    int (*init_bl)(struct mxs_platform_bl_data *data);
    void (*set_bl_intensity)(struct mxs_platform_bl_data *data,
                        struct backlight_device *bd, int suspended);
    void (*free_bl) (struct mxs_platform_bl_data *);
};
```

`mxs_platform_fb_entry` is the structure that completely defines a panel driver. Panel driver entries are then linked into the list in the `platform_data(struct mxs_platform_fb_data)` of the frame buffer platform device.

`mxs_platform_bl_data` is the structure that completely defines a backlight driver. Since the backlight driver is panel-specific, it should be selected based on the panel selection. It is implemented using notifiers described below.

The following functions are defined within the API:

Group 1 functions implement low-level LCD interface handling and are for panel driver usage.

- `void mxs_init_lcdif(void);`
- `int mxs_lcdif_dma_init(struct device *dev, dma_addr_t phys, int memsize);`
- `void mxs_lcdif_dma_release(void);`
- `void mxs_lcdif_run(void);`
- `void mxs_lcdif_stop(void);`
- `int mxs_lcdif_pan_display(dma_addr_t addr);`

Group 2 functions are used to manipulate panel drivers entries, for example to add a platform driver entry to frame buffer platform devices `platform_data` and search through registered panel driver entries in order to find a match. See Chapter 9, “Frame Buffer Driver,” for more information.

- ```
  void mxs_lcd_register_entry(struct mxs_platform_fb_entry *pentry,
                              struct mxs_platform_fb_data *pdata);
  ```
- ```
  int mxs_lcd_iterate_pdata(struct mxs_platform_fb_data *pdata,
                            int (*func)(struct mxs_platform_fb_entry *pentry,
                                        void *data, int ret_prev),
                                        void *data);
  ```
- `void mxs_lcd_set_bl_pdata(struct mxs_platform_bl_data *pdata);`

Group 3 functions are intended for backlight driver usage. The backlight driver registers its notifier client to get updates on the LCD interface mode selected. When the mode selected is DVI mode, an external display is used and the backlight should be turned off. See Chapter 11, “Backlight Driver,” for more information.

- `int mxs_lcdif_register_client(struct notifier_block *nb);`
- `void mxs_lcdif_unregister_client(struct notifier_block *nb);`
- ```
  void mxs_lcdif_notify_clients(unsigned long event,
                                struct mxs_platform_fb_entry *pentry);
  ```
  ```

Chapter 11
Backlight Driver

The LCD backlight driver is designed using the Linux kernel frame buffer/backlight driver framework. It implements the platform driver for a device. The driver is organized in a flexible and extensible manner and is abstracted from any specific LCD panel. To support a backlight, the low-level backlight handling functions are implemented and linked with the panel driver structures for which this backlight driver is valid. See Chapter 10, "LCD Interface (LCDIF) Driver," for more details on the panel handling interface.

11.1 Hardware Operation

The hardware operation for the backlight depends on the actual LCD panel. However, some issues are the same for any implementation. The actual pin that drives the backlight is PWM2, and the PWM is programmed depending on how backlight is implemented on the LCD panel.

11.2 Software Operation

A backlight device is controlled mainly by the sysfs interface. The sysfs backlight control directory (assuming that sysfs is mounted at `/sys`) is `/sys/class/backlight/mxs-bl`. Backlight parameters such as brightness and max_brightness can be queried and set (max_brightness is read-only). See `drivers/video/backlight/backlight.c` and the `bl_device_attributes` array for more information.

The backlight driver is abstracted from the actual hardware. It subscribes to LCD panel change notifications though the LCDIF API. Once the LCD panel is initialized, the driver receives notification which contains the hardware-specific backlight support data and then starts using the data for backlight programming.

11.3 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_BACKLIGHT_MXS [=M|Y]

  Configuration option for the MXS frame buffer driver, which is dependent on the BACKLIGHT CLASS DEVICE option.

  This option can be found under:

  Device Drivers > Graphics Support > Backlight & LCD device support
- CONFIG_FB_MXS_LMS430 [=M|Y]

  Configuration option to compile support for Samsung LMS430 "dot clock" LCD panel and backlight handling into the kernel.

11.4 Source Code Structure

The generic backlight driver source code is located in `drivers/video/backlight/mxs_bl.c`.

The panel support code which includes hardware-dependent backlight handling routines is located in `arch/arm/plat-mxs`.

The backlight driver includes the source/header files shown in Table 11-1.

Table 11-1. Backlight Driver Files

File	Description
`lcd_lms430.c`	Hardware backlight support for the Samsung LMS430 “dot clock” LCD panel

Chapter 12
Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver

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

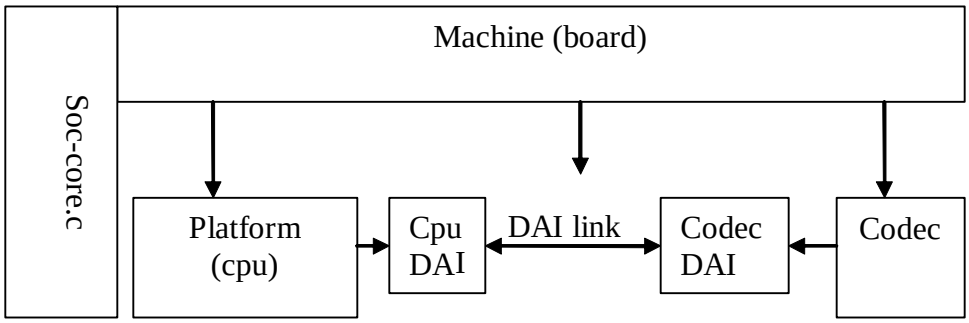

Figure 12-1. ALSA SoC Software Architecture

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

- Machine driver—handles any machine specific controls and audio events, such as turning on an external amp at the beginning of playback.
- Platform driver—contains the audio DMA engine and audio interface drivers (for example, I^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.

12.1 SoC Sound Card

Currently, the stereo codec (`sgtl5000`), 5.1 codec (`wm8580`), 4-channel ADC codec (ak5702), 7.1 codec(cs42888), built-in ADC/DAC codec, and Bluetooth codec drivers are implemented using SoC architecture. The five 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.

12.1.1 Stereo Codec Features

The stereo codec supports the following features:

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

12.1.2 Sound Card Information

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

```
root@freescale /$ aplay -l
**** List of PLAYBACK Hardware Devices ****
      card 0: imx3stack [imx-3stack], device 0: SGTL5000 SGTL5000-PCM-0 []
       Subdevices: 1/1
       Subdevice #0: subdevice #0
      card 0: mxsevk [mxs-evk], device 0: SGTL5000 SGTL5000-0 []
       Subdevices: 1/1
       Subdevice #0: subdevice #0
      card 1: mxsevk_1 [mxs-evk], device 0: MXS SPDIF mxs spdif-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 0: mxsevk [mxs-evk], device 0: SGTL5000 SGTL5000-0 []
       Subdevices: 1/1
       Subdevice #0: subdevice #0
```

12.2 ASoC Driver Source Architecture

As shown in Figure 12-2, `imx-pcm.c` is shared by the stereo ALSA SoC driver, the 5.1 ALSA SoC driver and the Bluetooth codec driver. This file is responsible for preallocating 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 stereo codec is connected to the CPU through the SAIF interface. `mxs-dai.c` registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip SAIF interface. `sgtl5000.c` registers the stereo codec and hifi DAI drivers. The direct hardware operations on the stereo codec are in `sgtl5000.c`. `mxs-devb.c` is the machine layer code which creates the driver device and registers the stereo sound card.

Figure 12-2 shows the ALSA SoC source file relationship.

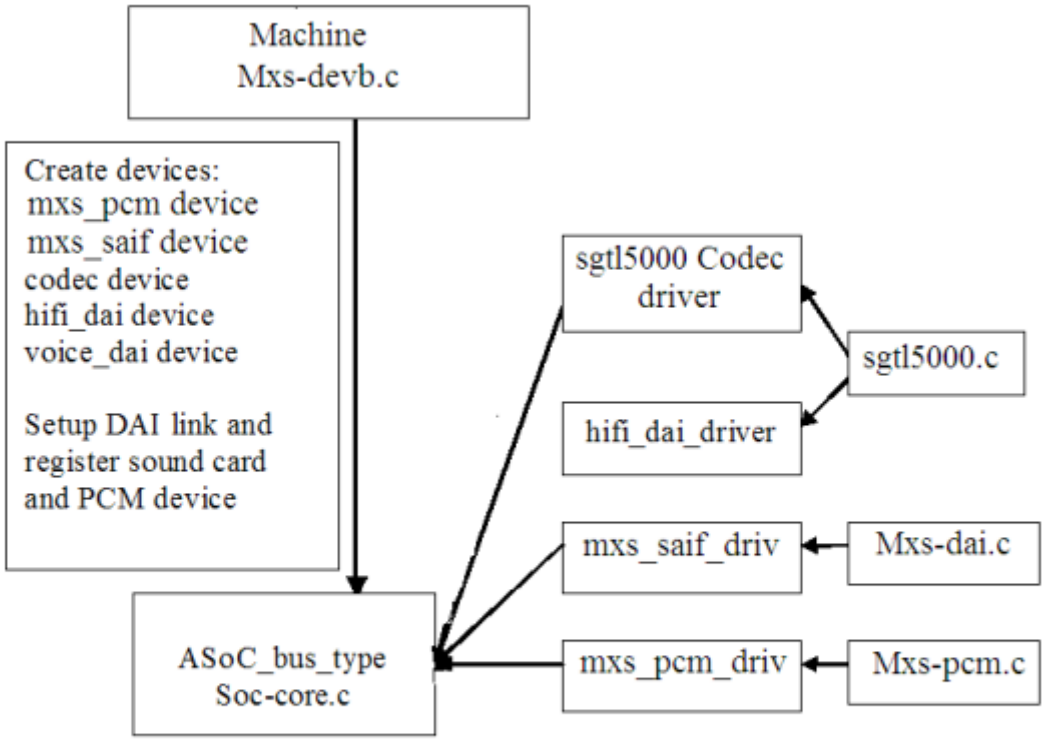

Figure 12-2. ALSA Soc Source FIle Relationship

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

Table 12-1. External Stereo Codec ASoC Driver Source File

File	Description
mxs/mxs-devb.c	Machine layer
mxs/mxs-pcm.c	Platform layer for ALSA SoC codec
mxs/mxs_pcm.h	Header file for PCM driver
mxs/mxs-dai.c	Platform DAI link
mxs/mxs-dai.h	Platform DAI link header
codec/sgtl5000.c	Codec layer
codecs/sgtl5000.h	Header file

12.3 Menu Configuration Options

The following Linux kernel configuration options are provided for this module. To get to these options, use the `./ltib -c` command when located in the `<ltib dir>`. Select Configure the Kernel on the screen displayed 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
- SoC Audio support for i.MX28 EVK Board. In menuconfig, this option is available under

  Device drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio support for MXS-EVK SGTL5000, MXS Digital Audio Interface SAIF

12.4 Hardware Operation

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

12.4.1 Stereo Audio Codec

The stereo audio codec is controlled by the I^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 operates in master mode and provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate.

The stereo audio codec is controlled by the I^2C interface. The audio data is transferred from the user data buffer to/from the SAIF FIFO through the DMA channel. Playback uses SAIF0 and Record uses SAIF1. SAIF0 operates in master mode and provides the MCLK, BCLK and LRCLK, SAIF1 and SGTL5000 codec work in slave mode, using clock of SAIF0. The BCLK and LRCLK are 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.

12.5 Software Operation

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

12.5.1 Sound Card Registration

The codecs have the same registration sequence:

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

12.5.2 Device Open

The ALSA driver:

- Allocates a free substream for the operation to be performed
- Opens the low level hardware device

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

After triggering for the first time, the subsequent DMA read/write operations are configured by the DMA callback.

Chapter 13
Pixel Pipeline (PxP) Driver

The Pixel Pipeline (PxP) is supported as a Linux kernel Video4Linux driver. The PxP driver is designed and implemented to comply with version 2.0 of the Video4Linux API (V4L2). The software implements a platform driver for a V4L2 output and output overlay device. The implementation interacts directly with the PxP hardware block registers. The LCD driver is organized in a flexible and extensible manner. It interacts with the i.MX28 frame buffer driver to accomplish frame buffer overlay of the video stream and output to the display. See Chapter 9, "Frame Buffer Driver," for more details on the frame buffer driver.

13.1 Hardware Operation

The PxP driver uses PxP registers to interact with the hardware. It also uses private APIs to gather frame buffer information and send PxP output to the display using functions in the frame buffer driver.

13.2 Software Operation

A V4L2 driver utilizes the V4L2 driver framework to provide functionality by a standard character driver model. The V4L2 API Specification (Revision 0.24) provides complete details of the API exported to user space applications. The following V4L2 features are supported by the driver:

- RGB555, RGB565, RGB24, YUV420 (planar), and YUV422P (planar) input formats
- Programmable input pixel format and size
- Mmap streaming input buffers
- Direct PxP output to the display
- Overlay of the frame buffer on the PxP input stream
- Color keying and alpha blending of the overlay
- Horizontal and vertical flipping of the PxP output
- 90°, 180°, and 270° rotation of the PxP output
- Programmable scaling of YUV420 and YUV422P input formats
- Programmable default background color
- Selection of YCbCr or YUV color space

These features are supported using custom APIs:

- Output to mmap user buffer

The supported V4L2 IOCTLs include the following:

- VIDIOC_QUERYCAP
- VIDIOC_REQBUFS

- VIDIOC_QUERYBUF
- VIDIOC_QBUF
- VIDIOC_DQBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF
- VIDIOC_ENUMOUTPUT
- VIDIOC_G_OUTPUT
- VIDIOC_S_OUTPUT
- VIDIOC_ENUM_FMT
- VIDIOC_TRY_FMT
- VIDIOC_G_FMT
- VIDIOC_S_FMT
- VIDIOC_G_FBUF
- VIDIOC_S_FBUF
- VIDIOC_G_CROP
- VIDIOC_S_CROP
- VIDIOC_QUERYCTRL
- VIDIOC_G_CTRL
- VIDIOC_S_CTRL

The following V4L2 standard controls are implemented by the driver:

- V4L2_CID_HFLIP
- V4L2_CID_VFLIP

The following V4L2 custom controls have been added to the driver:

- V4L2_CID_PRIVATE_BASE—Rotation (0°, 90°, 180°, or 270°)
- V4L2_CID_PRIVATE_BASE + 1—Background Color
- V4L2_CID_PRIVATE_BASE + 2—Set S0 Chromakey
- V4L2_CID_PRIVATE_BASE + 3—Color space (0 - YCbCr, 1 - YUV)

13.3 Menu Configuration Options

The following Linux kernel configuration option is provided for this module:

- CONFIG_VIDEO_PXP [=M|Y]

  Configuration option for the PxP V4L2 output overlay driver, which is dependent on the VIDEO_DEV, VIDEO_V4L2, and ARCH_MXS options. In menuconfig, this option is available under:

  Multimedia devices > Video capture adapter > MXS PxP

13.4 Source Code Structure

The PxP driver source code is located in `drivers/media/video/mxs_pxp.c` and `drivers/media/video/mxs_pxp.h`.

Chapter 14
NAND Flash Driver

The NAND Flash Memory Technology Devices (MTD) driver is used in the Generic-Purpose Media Interface (GPMI) controller on the i.MX28. Only the hardware specific layer has to be implemented for the NAND MTD driver to operate. 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.

14.1 Hardware Operation

NAND Flash is a nonvolatile storage device used for embedded systems. It does not support random accesses of memory as in the case of RAM or NOR Flash. Reading or writing to NAND Flash must be done through the GPMI. NAND Flash is a sequential access device appropriate for mass storage applications. Code stored on NAND Flash can not be executed from there. Code must be loaded into RAM memory and executed from there. The i.MX28 contains a hardware error-correcting block.

14.2 Software Operation

MTDs in Linux cover all memory devices such as RAM, ROM, and different kinds of NOR/NAND Flashes. The MTD subsystem provides uniform access to all such devices. Above the MTD devices there could be either MTD block device emulation with a Flash file system (JFFS2) or a UBI layer. The UBI layer in turn, can have either UBIFS above the volumes or a Flash Translation Layer (FTL) with a regular file system (FAT, Ext2/3) above it. The hardware specific driver interfaces with the GPMI module on i.MX28. It implements the lowest level operations such as read, write and erase. If enabled, it also provides information about partitions on the NAND device—this information has to be provided by platform code.

The NAND driver is the point where read/write errors can be recovered, if possible. Hardware error correction is performed by ECC8 or BCH blocks and is driven by NAND drivers code.

Detailed information about NAND driver interfaces can be found at http://www.linux-mtd.infradead.org

14.2.1 Basic Operations: Read/Write

The NAND driver exports the following callbacks:

- `gpmi_read_page_syndrome` (with ECC)
- `gpmi_write_page_syndrome` (with ECC)
- `gpmi_read_buf` (without ECC)
- `gpmi_write_buf` (without ECC)
- `gpmi_read_byte` (without ECC)
- `gpmi_read_word` (without ECC)

- gpmi_read_oob (with ECC)
- gpmi_write_oob (with ECC)

These functions read the requested amount of data, with or without error correction. In the case of read, the gpmi_read_dma function is called, which creates the DMA chain, submits it to execute, and waits for completion. The write case is a bit more complex: the data to be written is mapped and flushed out by calling gpmi_flush_dma before processing the command NAND_CMD_PAGEPROG.

14.2.2 Error Correction

When reading or writing data to Flash, some bits can be flipped. This is normal behavior, and NAND drivers utilize various error correcting schemes to correct this. It could be resolved with software or hardware error correction. The GPMI driver uses only a hardware correction scheme with the help of an ECC8 hardware accelerator.

The following functions deal with hardware ECC:

- ecc8_init
- ecc8_exit
- ecc8_read
- ecc8_write
- ecc8_setup
- ecc8_stat
- ecc8_reset

14.2.3 Boot Control Block Management

During startup, the NAND driver scans the first block for the presence of a NAND Control Block (NCB). Its presence is detected by magic signatures. When a signature is found, the boot block candidate is checked for errors using Hamming code. If errors are found, they are fixed, if possible. If the NCB is found, it is parsed to retrieve timings for the NAND chip.

All boot control blocks are created when formatting the medium using the user space kobs application.

14.2.4 Bad Block Handling

When the driver begins, by default, it builds the bad block table. It is possible to determine if a block is bad, dynamically, but to improve performance it is done at boot time. The badness of the erase block is determined by checking a pattern in the beginning of the spare area on each page of the block. However, if the chip uses hardware error correction, the bad marks falls into the ECC bytes area. Therefore, if hardware error correction is used, the bad block mark should be moved. The driver decides if bad block marks should be moved if there is no NAND control block. Then, to prevent another move of bad block marks, the driver writes the default NCB to the Flash.

The following functions that deal with bad block handling are grouped together in the gpmi-bbt.c file:

- gpmi_block_bad
- gpmi_scan_bbt
- gpmi_scan_sigmatel_bbt
- gpmi_write_ncb

14.3 Source Code Structure

The NAND driver is located in the `drivers/mtd/nand/gpmi` directory. The following files are included in the NAND driver:

- `gpmi-base.c`
- `gpmi-bbt.c`
- `gpmi-bch.c`
- `gpmi-ecc8.c`
- `gpmi.h`
- `gpmi-hamming-13-8.c`
- `gpmi-hamming-22-16.c`
- `gpmi-hamming-13-8.h`
- `gpmi-hamming-22-16.h`

14.4 Menu Configuration Options

To enable the NAND driver, the following options must be set:

- CONFIG_MTD_NAND_GPMI = [Y | M]
- CONFIG_MTD_NAND_GPMI_SYSFS_ENTRIES = y
- CONFIG_MTD_NAND_GPMI_TA1 = y
- CONFIG_MTD_NAND_GPMI_TA3 = y

In addition, these MTD options must be enabled:

- CONFIG_MTD_NAND = [y | m]
- CONFIG_MTD = y
- CONFIG_MTD_PARTITIONS = y
- CONFIG_MTD_CHAR = y
- CONFIG_MTD_BLOCK = y

Chapter 15
ENET IEEE-1588 Driver

ENET IEEE-1588 driver performs a set of functions that enabling precise synchronization of clocks in network communication. The driver requires a protocol stack to complete IEEE-1588 full protocol. It complies with the IXXAT stack interfaces.

15.1 Hardware Operation

To allow for IEEE 1588 or similar time synchronization protocol implementations, the ENET MAC is combined with a time-stamping module to support precise time stamping of incoming and outgoing frames. 1588 Support is enabled when the register bit ENA_1588 is set to '1'.

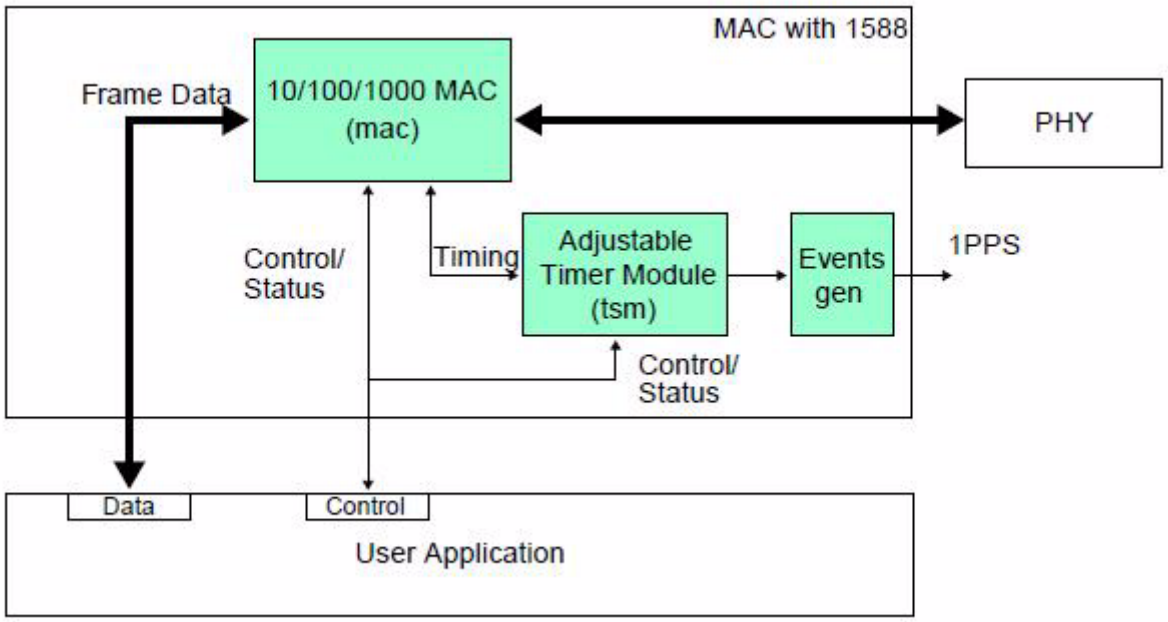

Figure 15-1. IEEE 1588 Functions Overview

15.1.1 Transmit Timestamping

On transmit, only 1588 event frames need to be time-stamped. The Client application (for example, the MAC driver) should detect 1588 event frames and set the signal ff_tx_ts_frm together with the frame.

For every transmitted frame, the MAC returns the captured timestamp on tx_ts (31:0) with the frame sequence number (tx_ts_id(3:0)) and the transmit status. The transmit status bit tx_ts_stat (5) indicates that the application had the ff_tx_ts_frm signal asserted for the frame.

If ff_tx_ts_frm is set to '1', the MAC additionally memorizes the timestamp for the frame in the register TS_TIMESTAMP. The interrupt bit EIR (TS_AVAIL) is set to indicate that a new timestamp is available.

Software would implement a handshaking procedure by setting the ff_tx_ts_frm signal when it transmits the frame it needs a timestamp for and then waits on the EIR (TS_AVAIL) interrupt bit to know when the timestamp is available. It then can read the timestamp from the TS_TIMESTAMP register. This is done for all event frames; other frames do not use the ff_tx_ts_frm indicator and hence do not interfere with the timestamp capture.

15.1.2 Receive Timestamping

When a frame is received, the MAC latches the value of the timer when the frame SFD field is detected and provides the captured timestamp on ff_rx_ts(31:0). This is done for all received frames.

The DMA controller has to ensure that it transfers the timestamp provided for the frame into the corresponding field within the receive descriptor for software access.

15.2 Software Operation

The 1588 Driver has the functions listed below:

- Module initialization—Initializes the module with the device specific structure, and registers a character driver.
- IXXAT stack interface—Respond to protocol stackís command by IOCTL routine, such as GET_TX_TIMESTAMP, SET_RTC_TIME.
- Interrupt servicing routine—Supports events, such as TS_AVAIL, TS_TIMER. The driver shares interrupt servicing routine with FEC driver.
- Miscellaneous routines—Maintain the timestamp circle queue.

15.3 Source Code Structure

Table 15-1 lists the source files available in the `<ltib_dir>/rpm/BUILD/linux/drivers/net` directory.

Table 15-1. ENET 1588 File List

File	Description
`fec_1588.h`	Header file defining registers
`fec_1588.c`	Linux driver for ENET 1588 timer

For more information about the generic Linux driver, see the `<ltib_dir>/rpm/BUILD/linux/drivers/net/fec_1588.c` source file.

15.4 Linux Menu Configuration Options

To get to the ENET 1588 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_FEC_1588 Linux kernel configuration is provided for this module. This option is available under Device Drivers > Network device support > Ethernet (10 or 100 Mbit) > Enable FEC 1588 timestamping.

15.5 Programming Interface

The 1588 driver complies with the IXXAT protocol stack interface. Stack-specific defines are added to the header file (fec_1588.h).

15.5.1 IXXAT Specific Data structure Defines

Protocol-specific defines are added to the header file (`fec_1588.h`).

```
/* PTP standard time representation structure */
struct ptp_time{
u64 sec; /* seconds, unsigned */
u32 nsec;/* nanoseconds, signed */
};

/* interface for PTP driver command GET_TX_TIME */
struct ptp_ts_data {
/* PTP version */
u8 version;
/* PTP source port ID */
u8 spid[10];
/* PTP sequence ID */
u16 seq_ID;
/* PTP message type */
u8 message_type;
/* PTP timestamp */
ptp_time ts;
};

/* interface for PTP driver command SET_RTC_TIME/GET_CURRENT_TIME */
struct ptp_rtc_time {
ptp_time rtc_time;
};

/* interface for PTP driver command SET_COMPENSATION */
struct ptp_set_comp {
u32 drift;
};

/* interface for PTP driver command GET_ORIG_COMP */
struct ptp_get_comp {
/* the initial compensation value */
u32 dw_origComp;
/* the minimum compensation value */
u32 dw_minComp;
/*the max compensation value*/
u32 dw_maxComp;
/*the min drift applying min compensation value in ppm*/
u32 dw_minDrift;
/*the max drift applying max compensation value in ppm*/
u32 dw_maxDrift;
```

```
};

/* PTP default message type */
#define DEFAULT_msg_Sync                                                    0x0
#define DEFAULT_msg_Delay_Req                                               0x1
#define DEFAULT_msg_Peer_Delay_Req0x2
#define DEFAULT_msg_Peer_Delay_Resp0x3

/* PTP message version */
#define PTP_1588_MSG_VER_11
#define PTP_1588_MSG_VER_22
```

15.5.2 IXXAT IOCTL Commands Defines

Command: PTP_GET_TX_TIME

Description: command provides the timestamp of the transmit packet with specific PTP sequence ID and returns the timestamp, the sender port-ID, the PTP version, and the message type through the ptp_ts_data structure.

Command: PTP_GET_RX_TIME

Description: command provides the timestamp of the receive packet with specific PTP sequence ID and returns the timestamp, the sender port-ID, the PTP version, and the message type, through the ptp_ts_data structure.

Command: PTP_SET_RTC_TIME

Description: command sets the RTC time register with provided PTP time through the ptp_rtc_time structure.

Command: PTP_SET_COMPENSATION

Description: command sets the drift compensation with provided compensation value through the ptp_set_comp structure.

Command: PTP_GET_CURRENT_TIME

Description: command provides the current RTC time and returns the timestamp through the ptp_rtc_time structure.

Command: PTP_FLUSH_TIMESTAMP

Description: command flushes the transmit and receive timestamp queues.

Command: PTP_GET_ORIG_COMP

Description: command provides the original frequency compensation, minimum frequency compensation, maximum frequency compensation, minimum drift and maximum drift of RTC through the ptp_get_comp structure.

Chapter 16
Programmable 3-Port Ethernet Switch Driver

The SWITCH driver performs the full set of switch interface functions. The switch requires an external interface adaptor and transceiver function to complete the interface to the Ethernet media. It supports half or full-duplex operation on 10 Mbps or 100 Mbps related Ethernet networks.

The Switch driver has the following features:

- Support Input Port selection.
- Supports IP Snooping and TCP/UDP port number snooping.
- Supports VLAN Input and Output Processing.
- Supports Frame Classification & Priority Resolution.
- Supports L2 MAC address look-up.
- Support Vlan domain Verification in the Frame Forwarding task.
- Support Broadcast/Multicast/VLAN Domain Resolution in the Frame Forwarding task.
- Support Port Mirroring in the Frame Forwarding task.
- Support Bridge Protocol Frame Resolution in the Frame Forwarding task.
- Support Congestion Resolution in the Frame Forwarding tack.

This switch can be accessed through the `ifconfig` command with interface name (`eth0`). The driver shall auto-probe the external adaptor (PHY device).

16.1 Hardware Operation

The Switch Core is designed to be seamlessly connected to the MorethanIP MAC-NET Core and DMA controllers. For control and configuration, the switch implements an APB Register interface and multiple maskable interrupts. See the Figure 16-1.

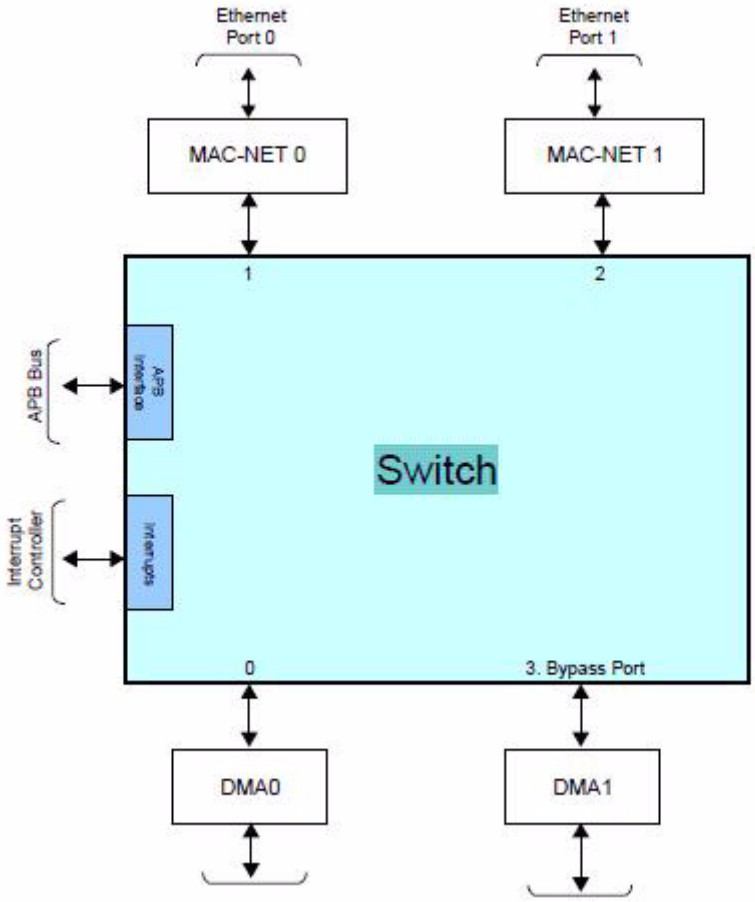

Figure 16-1. Switch Interface

The switch port assignment is listed in Table 16-1 and should be strictly followed when programming and integrating the Switch Core.

Table 16-1. Port Assignment

Switch port	Assignment
0	DMA0
1	MAC-NET 0
2	MAC-NET 1
3(bypass port)	DMA1

The Switch, controlled with the configuration pin sx_ena, can be programmed to operate is two modes:

- Passthrough mode: The switch logic is disabled and bypassed.
- Switch mode: The switch logic is enabled.

16.1.1 Passthrough Mode

When configuration signal sx_ena is set to '0', the switch logic is bypassed and can be totally powered down and disabled with, for example, the Switch clocks clk and pclk stopped and the Switch reset signals reset_clk and reset_pclk set to '1'.

The Switch APB interface and interrupt signals are disabled and should not be used. To control the Frame transfer from DMA0 and DMA1, the MAC-NET 0 and the MAC-NET 1 APB interfaces and interrupt signals should be used.

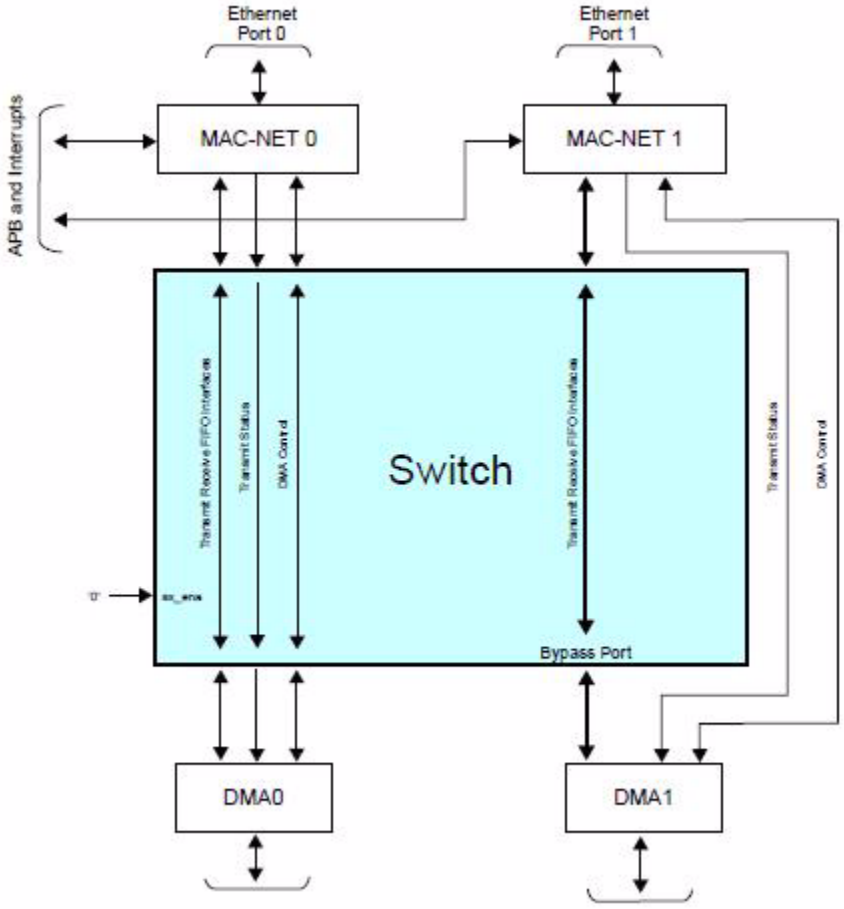

Figure 16-2. Passthrough Mode Configuration Overview

16.1.2 Switch Mode

When the Switch is programmed to operate in Switch Mode (sx_ena set to '1'), the Bypass Mode (Port 1) interface is disabled and should not be used.

Frame transfers to and from the Line are performed on Port 0 only (DMA 0). The Transmit status signals are generated from the Switch Port 0 Receive Buffer and the DMA control signals from the Switch Register Space. The MAC-NET 0 and MAC-NET 1 Transmit status and DMA control signals are not used.

The MAC-NET 0 and MAC-NET 1 APB interfaces and interrupts are enabled and can be used to monitor the line activity and gather the line statistic information.

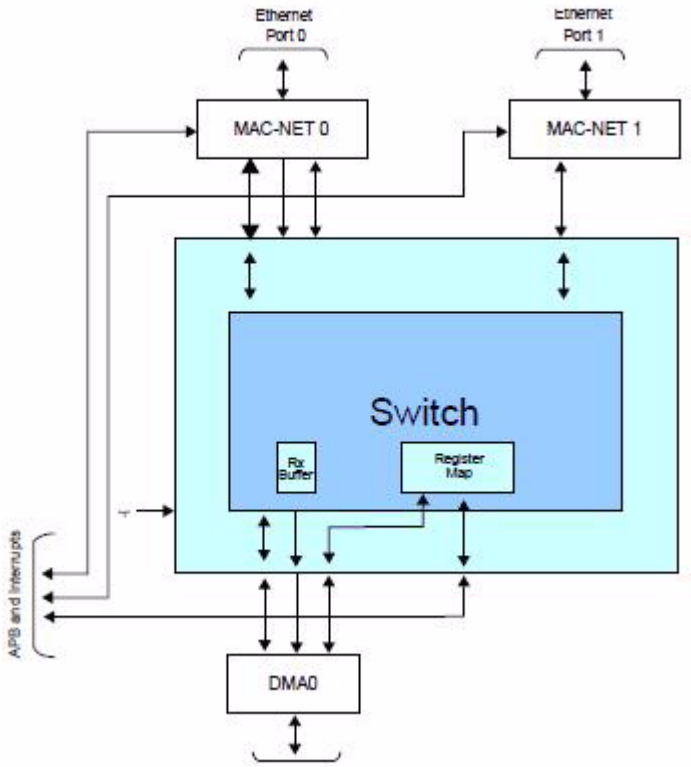

Figure 16-3. Switch Mode Configuration Overview

16.2 Software Operation

The FEC Driver has the functions listed below:

- Module initialization—Initializes the module with the device specific structure.
- Driver entry points—Provides standard entry points for transmission, such as `switch_enet_start_xmit` and for reception of Ethernet packets through the ISR, such as `switch_enet_interrupt`.

- Interrupt servicing routine—Supports events, such as TXF, RXF and MII.
- Miscellaneous routines—Different routines come under this category, such as `switch_timeout` for waking up network stack.

16.3 Source Code Structure

Table 16-2 lists the source files available in the `<ltib_dir>/rpm/BUILD/linux/drivers/net` directory.

Table 16-2. Ethernet File List

File	Description
`fec_switch.h`	Header file defining registers
`fec_switch.c`	Linux driver for 3-Port Ethernet Switch

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

16.4 Linux Menu Configuration Options

To get to the Switch 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_FEC_L2SWITCH Linux kernel configuration is provided for this module. This option is available under Device Drivers > Network device support > Ethernet (10 or 100 Mbit) > L2 Switch Ethernet Controller.

If want to use Switch Ethernet controller, disable the FEC Ethernet controller config in the menuconfig.

16.5 Programming Interface

Table 16-2 lists the source files for the Switch Driver. The following sections show modifications that were required in the original Ethernet driver source for porting it to the i.MX family multimedia application processors.

16.5.1 Device Specific Defines

Device-specific defines are added to the header file (`fec_switch.h`).

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

```
struct switch_t {
        unsigned long ESW_REVISION; /* Revision Register*/
        unsigned long ESW_SCRATCH; /* Scratch Register*/
        unsigned long ESW_PER; /* Port Enable Register*/
        unsigned long reserved0[1];
        unsigned long ESW_VLANV; /* VLAN Verify*/
        unsigned long ESW_DBCR; /*Default Broadcast Resolution*/
        unsigned long ESW_DMCR; /*Default Multicast Resolution*/
        unsigned long ESW_BKLR; /*Blocking and Learning Enable*/
        unsigned long ESW_BMPC; /*Bridge Management Port*/
```

```
        unsigned long ESW_MODE; /*Mode Configuration Register*/
        unsigned long ESW_VIMSEL; /* VLAN Input Manipulation Select*/
        unsigned long ESW_VOMSEL; /*VLAN Output Manipulation Select*/
        unsigned long ESW_VIMEN; /*VLAN input manipulation enable*/
        unsigned long ESW_VID;/* VLAN Tag ID*/
        unsigned long esw_reserved0[2];
        unsigned long ESW_MCR;/* Mirror control register*/
        unsigned long ESW_EGMAP; /* Egress Port Definitions*/
        unsigned long ESW_INGMAP; /* Ingress Port Definitions*/
        unsigned long ESW_INGSAL; /* Ingress and Egress MAC Address Registers*/
        unsigned long ESW_INGSAH; /* Ingress and Egress MAC Address Registers*/
        unsigned long ESW_INGDAL; /* Ingress and Egress MAC Address Registers*/
        unsigned long ESW_INGDAH; /* Ingress and Egress MAC Address Registers*/
        unsigned long ESW_ENGSAL; /* Ingress and Egress MAC Address Registers*/
        unsigned long ESW_ENGSAH; /* Ingress and Egress MAC Address Registers*/
        unsigned long ESW_ENGDAL; /* Ingress and Egress MAC Address Registers*/
        unsigned long ESW_ENGDAH; /* Ingress and Egress MAC Address Registers*/
        unsigned long ESW_MCVAL;/* Mirror Count Value*/
        /*from 0x70--0x7C*/
        unsigned long esw_reserved1[4];
        unsigned long ESW_MMSR;/* Memory Manager Status Register*/
        unsigned long ESW_LMT; /* Low Memory Threshold*/
        unsigned long ESW_LFC; /* Lowest Number of Free Cells*/
        unsigned long ESW_PCSR; /* Port Congestion Status*/
        unsigned long ESW_IOSR; /* Input/Output Interface Status Register*/
        unsigned long ESW_QWT;/* Queue Weights*/
        unsigned long esw_reserved2[1];/*0x98*/
        unsigned long ESW_P0BCT;/* Port 0 Backpressure Congestion Threshold*/
        unsigned long esw_reserved3[7];
        unsigned long ESW_P0FFEN;/* Port 0 Forced Forwarding Enable*/
        unsigned long ESW_PSNP[8]; /* Port Snooping Registers*/
        unsigned long ESW_IPSNP[8]; /* IP snooping registers*/
        unsigned long ESW_PVRES[3]; /* Port 0ñ2 VLAN Priority Resolution Map*/
        unsigned long esw_reserved4[13];
        unsigned long ESW_IPRES;/* IPv4/v6 Priority Resolution Table*/
        /*from 0x144-0x17C*/
        unsigned long esw_reserved5[15];
        unsigned long ESW_PRES[3]; /* Port n Priority Resolution Configuration*/
        /*from 0x18C-0x1FC*/
        unsigned long esw_reserved6[29];
        unsigned long ESW_PID[3]; /* Port n VLAN ID*/
        /*from 0x20C-0x27C*/
        unsigned long esw_reserved7[29];
        unsigned long ESW_VRES[32]; /* VLAN Domain Resolution 0ñ31*/
        unsigned long ESW_DISCN;/* Statistics Registers*/
        unsigned long ESW_DISCB; /* Statistics Registers*/
        unsigned long ESW_NDISCN; /* Statistics Registers*/
        unsigned long ESW_NDISCB;/* Statistics Registers*/
        struct esw_port_statistics_status port_statistics_status[3]; /*Port Statistics
Registers*/
        /*from 0x340-0x400*/
        unsigned long esw_reserved8[48];

        /*0xFC0DC400---0xFC0DC418*/
        /*unsigned long MCF_ESW_ISR;*/
        unsigned long   switch_ievent;            /* Interrupt event reg */
        /*unsigned long MCF_ESW_IMR;*/
```

```
        unsigned long   switch_imask;              /* Interrupt mask reg */
        /*unsigned long MCF_ESW_RDSR;*/
        unsigned long   fec_r_des_start;        /* Receive descriptor ring */
        /*unsigned long MCF_ESW_TDSR;*/
        unsigned long   fec_x_des_start;        /* Transmit descriptor ring */
        /*unsigned long MCF_ESW_MRBR;*/
        unsigned long   fec_r_buff_size;        /* Maximum receive buff size */
        /*unsigned long MCF_ESW_RDAR;*/
        unsigned long   fec_r_des_active;       /* Receive descriptor reg */
        /*unsigned long MCF_ESW_TDAR;*/
        unsigned long   fec_x_des_active;       /* Transmit descriptor reg */
        /*from 0x420-0x4FC*/
        unsigned long esw_reserved9[57];

        /*0xFC0DC500---0xFC0DC508*/
        unsigned long ESW_LREC0; /* aaaa */
        unsigned long ESW_LREC1; /* aaaa */
        unsigned long ESW_LSR; /* aaaa */
};

/*
 *      Define the buffer descriptor structure.
 */
struct cbd_t {
#if defined(CONFIG_ARCH_MXC) || defined(CONFIG_ARCH_MXS)
        unsigned short  cbd_datlen;             /* Data length */
        unsigned short  cbd_sc;                 /* Control and status info */
#else
        unsigned short  cbd_sc;                 /* Control and status info */
        unsigned short  cbd_datlen;             /* Data length */
#endif
        unsigned long   cbd_bufaddr;            /* Buffer address */
#ifdef ENHANCE_BUFFER
        unsigned long   ebd_status;
        unsigned short  length_proto_type;
        unsigned short  payload_checksum;
        unsigned long   bdu;
        unsigned long   timestamp;
        unsigned long   reserverd_word1;
        unsigned long   reserverd_word2;
#endif
};
```

Chapter 17
Low-Level Keypad Driver

The low-level keypad driver interfaces with the Keypad Port Hardware (KPP) in the i.MX device. The MXS keypad driver interfaces with LRADC channel. The keypad driver is implemented as a standard Linux 2.6 keyboard driver, modified for the i.MX device.

The keypad driver supports the following features:

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

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

17.1 Hardware Operation

The keypad sends data using LRADC channel. At the end of each conversion, LRADC triggers an interrupt which allows retrieval of the scan-code of pressed/released buttons.

17.2 Software Operation

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

- When LRADC finishes conversion on selected channel, the keypad interrupt handler is called
- In the keypad interrupt handler, the conversion result is debounced and compared against a table of keycodes
- If found, the scan code is reported to the Linux input subsystem

The keypad driver registers the input device structure using the input register device. The driver sets the input bit fields and conveys all the events that can be generated by the keypad to other parts of the input subsystem. The keypad driver generates the EV KEY, EV REL and EV REP events. All the events are reported by calling the input report key. The reported event is passed up to the event interface. The event interface is implemented in the Evdev driver, if it is compiled in or built as a module. It implements a generic event interface, and passes timestamped input events from the kernel to the userspace. Userspace can read (either blocking or nonblocking) on the `/dev/input/eventX` device node.

17.3 Reassigning Keycodes

The keypad driver takes advantage of the input subsystem's ability to remap key codes. A user space application can use the `EVIOCGKEYCODE` and `EVIOCSKEYCODE` IOCTLs on the device node (for example `/dev/input/event0`) to get and set key codes. Applications such as keyfuzz and input-kbd (from the

input-utils package) use these IOCTLs which are handled by the input subsystem. See the *kernel* `Documentation/input/input-programming.txt` for details on remapping codes.

17.4 Driver Features

The keypad driver supports the following features:

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

17.5 Source Code Structure

Table 17-1 shows the keypad driver source files that are available in the following directories:

```
<ltib_dir>/rpm/BUILD/linux/drivers/input/keyboard
<ltib_dir>/rpm/BUILD/linux/include/linux
```

Table 17-1. Keypad Driver Files

File	Description
mxc_keyb.c	Low-level driver implementation
mxc_keyb.h	Driver structures, control register address definitions
mxs-kbd.c	Keypad ladder driver
input.h	Generic Linux keycode definitions
arch/arm/mach-mx28/device.c	Contains the platform-specific keymapping keycode array

17.6 Menu Configuration Options

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

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

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

  Device Drivers > Input device support > Event interface.

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

- Matrix config—The keypad matrix can be configured for up to eight rows and eight columns. The keypad matrix configuration can be done by changing the `rowmax` and `colmax` members in the `keypad_plat_data` structure in the platform specific file (see Table 17-1).
- Debounce delay—The user can configure the debounce delay by changing the variable `KScanRate` defined in `mxc_keyb.c`.

17.7 Programming Interface

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

17.8 Interrupt Requirements

Table 17-2 lists the keypad interrupt timer requirements.

Table 17-2. Keypad Interrupt Timer Requirements

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

Chapter 18
Touch Screen and ADC Drivers

18.1 Driver Overview

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

The touch screen driver interfaces with the channels 2, 3, 4 and 5 of LRADC. It is implemented within the standard Linux kernel touch screen framework. The touch screen can be accessed through the`/dev/input/eventX` device node.

18.2 Hardware Operation

The touch screen controller includes the following features:

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

The touch screen driver is divided into two parts:

- Touch detection—triggers an interrupt
- Movement capture—implemented using LRADC channel conversions

18.3 Software Operation

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

The touch screen driver implements a finite state machine to facilitate touch detection and movement capture. The initial state is touch detection and once a touch is detected, the state machine goes into X-coordinate movement detection. When enough samples are captured, Y-coordinate movement detection state is entered, and the full picture is formed.

18.4 Source Code Structure

Table 18-1 shows the touch screen driver source files found in the directory

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

Table 18-1. Touch Screen Driver Files

File	Description
`mxs-ts.c`	Touch screen driver implementation file

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:

- TOUCHSCREEN_MXS—Provided for the Touch screen driver. In menuconfig, this option is found under

  Device Driver > Input device support > Touchscreens > MXS LRADC-based touchscreen driver.

18.6 Programming Interface (Exported API)

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

18.7 Interrupt Requirements

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

Chapter 19
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

19.1 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

19.2 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 transfer data. 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.

19.3 Hardware Operation

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

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

19.4 Software Operation

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

19.4.1 I^2C Bus Driver Software Operation

The I^2C bus driver is described by a structure called `i2c_adapter`. The most important field in this structure is `struct i2c_algorithm *algo`. This field is a pointer to the `i2c_algorithm` structure that describes how data is transferred over the I^2C bus. The algorithm structure contains a pointer to a function that is called whenever the I^2C chip driver wants to communicate with an I^2C device.

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

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

19.4.2 I^2C Device Driver Software Operation

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

19.5 Driver Features

The I^2C driver supports the following features:

- I^2C communication protocol
- I^2C master mode of operation

NOTE

The I^2C driver do not support the I^2C slave mode of operation.

19.6 Source Code Structure

The I^2C bus driver source code is available in:

- `drivers/i2c/busses/i2c-mxs.c`
- `drivers/i2c/busses/i2c-mxs.h`
- `arch/arm/mach-mx28/include/mach/regs-i2c.h`

19.7 Menu Configuration Options

- CONFIG_I2C_MXS

19.8 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.`

19.9 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 19-1.

Table 19-1. 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 20
Data Co-Processor (DCP) Driver

The Data Co-Processor (DCP) cryptography driver is used to accelerate cryptographic operations in the kernel space and user-space (Linux Crypto API).

20.1 Hardware Operation

The DCP provides support for general encryption functions typically used for security algorithms. The supported modes are:

- Advanced Encryption Standard (AES)
- Electronic Code Book (ECB)
- Cipher Block Chaining (CBC)
- SHA-1
- SHA-256

The driver uses DMA to process the data in place.

20.2 Software Operation

The software implementation of the DCP driver conforms with the Linux Crypto driver model. It registers a number of ciphers and block ciphers. Refer to the documentation located in the folder `Documentation/crypto/` (kernel source tree) for full information about the Linux Crypto API.

The following ciphers are registered in the DCP driver module:

- AES

The following block ciphers are registered in the DCP driver module:

- AES (ECB)
- AES (CBC)

The following hashing algorithms are registered in the DCP driver module:

- SHA-1
- SHA-256

In addition, the DCP can perform 128-bit AES crypto using the OTP KEY0 which is not accessible by software, and therefore not usable through the Crypto APIs. The driver permits a user application to use AES 128-bit/ECB mode but only supports encrypting/decrypting a single 128-bit block. The ROM uses the OTP key during boot. Therefore, it is possible to verify that a bootstream is valid before committing it to Flash. While the bootstream uses AES/CBC mode, it is far simpler to use the ECB mode. The user space

access is performed by means of a miscellaneous device character file (under /proc/misc: dcpboot), and two IOCTLs: DBS_ENC for encryption and DBS_DEC for decryption.

Typical usage scenarios for encrypting/decrypting using the OTP key is as follows:

```
uint8_t mac[16];
int r, fd;

fd = open(<character-device-file>, O_RDWR);
/* check fd for successful open call */
/* load vector to mac */
r = ioctl(fd, DBS_ENC, mac); /* encrypt */
/* check r for failure (!=0) */
r = ioctl(fd, DBS_DEC, mac); /* decrypt */
/* check r for failure (!=0) */
close(fd);
```

20.3 Source Code Structure

The DCP cryptography module is implemented in the following files:

```
drivers/crypto/dcp.c
drivers/crypto/dcp.h
drivers/crypto/dcp-bootstream-ioctl.h
```

20.4 Menu Configuration Options

The following Linux kernel configuration options are provided for this module:

- CONFIG_CRYPTO_DEV_DCP [=M|Y]

  Configuration option for the DCP cryptography driver, which is dependent on ARCH_MX28 and automatically selects CRYPTO_ALGAPI and CRYPTO_BLKCIPHER.

  In menuconfig, this option is available under:
  Cryptographic API > Hardware crypto devices > Support for the DCP engine

20.5 Programming Interface

This driver is integrated into the Linux Crypto API.

20.6 Unit Test

1. Boot the board & login as root.
2. Run the AES crypto test module:

   ```
   root:~# modprobe tcrypt mode=10
   ```
3. Output should not contain any error messages related to ECB or CBC

NOTE

It is normal to have failures with a code -2 like:

```
testing lrw(aes) decryption
failed to load transform for lrw(aes): -2
```

It is also expected that modprobe fails to load the module, such as:

```
modprobe: failed to load module tcrypt.
```

4. Run the SHA-1 cryto test module:

```
root:~# modprobe tcrypt mode=2
```

5. Output should not contain any error messages related to SHA-1

NOTE

It is normal to have failure related to loading the module:

```
modprobe: failed to load module tcrypt.
```

6. Run the SHA-256 cryto test module:

```
root:~# modprobe tcrypt mode=6
```

7. Output should not contain any error messages related to SHA-256

NOTE

It is normal to have failure related to loading the module:

```
modprobe: failed to load module tcrypt.
```

8. Run the boostream verification test:

```
# kobs-ng imgverify -d imx28_linux.sb
# echo $?
0
```

If the return value is non-zero, then some error occurred during verification of the boostream using the OTP key. Use the “-v” option for a verbose output from kobs-ng.

Chapter 21
SPI Bus Driver

The i.MX28 board provides two synchronous ports for inter-IC communication as well as removable media control and communication. The SPI driver controls the SSP in SPI master mode.

21.1 Hardware Operation

The SPI bus is inherently a full-duplex bidirectional interface. However, the i.MX28 has a single DMA channel for SSP; therefore, the SSP must be programmed for either transmit or receive. In receive mode, the SSP continuously transfers the word written to its data register. In transmit mode, the SSP ignores incoming data.

21.2 Software Operation

During startup, the driver allocates, populates, and registers the structure `spi_master`. Along with other data, the structure contains pointers to the following functions:

- mxs_spi_setup
- mxs_spi_transfer
- mxs_spi_cleanup

When the driver is opened by the SPI device driver, `mxs_spi_setup` is called. This function configures the SSP hardware to the mode requested by the SPI device driver. When the SPI device driver wants to exchange data with the SPI device, the function `mxs_spi_transfer` is called as part of processing the `spi_sync/spi_async` request. The transmitting/receiving process is described in Section 21.2.1, "Transmitting Data," and Section 21.2.2, "Receiving Data." When the SPI device driver is finished with its activity, `mxs_spi_cleanup` is called. It releases the hardware acquired by `mxs_spi_setup`. The driver can work either in PIO mode (slow, not recommended) or DMA mode (the default, faster). The driver cannot send and receive data in the same transfer—this is a hardware limitation.

21.2.1 Transmitting Data

In PIO mode, the driver writes the data byte-by-byte to the HW_SSP_DATA register. Before transferring, the first byte driver manually raises the SS pin. Before transferring the last byte, the driver clears the SS pin.

In DMA mode, the DMA channel is programmed to send the exact number of bytes, and the driver waits for completion of the DMA operation.

21.2.2 Receiving Data

In PIO mode, the driver reads data byte-by-byte, and polls the HW_SSP_STATUS register for the FIFO_EMPTY bit. When the bit is clear, the driver reads the next byte and puts it in the buffer.

In DMA mode, the DMA channel is programmed to receive the exact number of bytes. The driver then waits for completion of the DMA operation.

21.3 Source Code Structure

The driver consists of the following files:

```
drivers/spi/spi_mxs.c
drivers/spi/spi_mxs.h
```

21.4 Menu Configuration Options

To enable the SPI bus driver, the following options must be set:

- CONFIG _SPI = Y
- CONFIG_SPI_MXS = [y | m]

Chapter 22
MMC/SD/SDIO Host Driver

The MultiMediaCard (MMC)/ Secure Digital (SD)/ Secure Digital Input Output (SDIO) Host driver implements a standard Linux driver interface to the SSP SD/MMC module. The host driver is part of the Linux kernel MMC framework.

The MMC driver has the following features:

- 1-bit or 4-bit operation for SD and SDIO cards
- Supports card insertion and removal detections
- Supports the standard MMC commands
- DMA data transfers
- Power management
- Supports 1/4/8-bit operations for MMC cards

22.1 Hardware Operation

The new high speed MMC communication is based on a 711-pin serial bus designed to operate in a low voltage range. The host controller module controls the card by sending commands and running data accesses from/to the card. The two communication protocols defined by the MMC specifications: SD and SPI. Only SD mode is supported.

22.2 Software Operation

The host controller driver is responsible for implementing the `mmc_host_ops` structure, with request, `set_ios`, and `get_ro` functions. These functions are called by the bus protocol driver. The host controller driver talks directly to the hardware.

The `mxs_mmc_request` function handles both read and write requests that come from the protocol driver. It calls the function `mxs_mmc_start_cmd` which configures the proper hardware registers depending on the command type, then runs the DMA operation, and waits for completion.

The `mxs_mmc_set_ios` function sets the bus width, voltage level, and clock rate according to the bus protocol driver requirements.

The `mxs_mmc_get_ro` function returns the status of the write-protection signal. This signal is retrieved using a helper function provided by the platform data callback, otherwise the driver assumes the card is read-write.

22.3 Driver Features

The MMC driver supports the following features:

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

22.4 Source Code Structure

The driver consists only of the file: `drivers/mmc/mxs-mmc.c`

22.5 Menu Configuration Options

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

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

  Device Drivers > MMC/SD/SDIO Card support

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

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

  By default, this option is Y.
- CONFIG_MMC_MXS—i.MX23/i.MX28 driver. In menuconfig, this option is available under Device Drivers > MMC/SD Card Support > Freescale MXC Multimedia Card Interface support.
- CONFIG_MMC_UNSAFE_RESUME—Used for embedded systems which use a MMC/SD/SDIO card for rootfs. In menuconfig, this option is found under

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

22.6 Programming Interface

This driver implements the functions required by the MMC bus protocol to interface with the i.MX SSP SD/MMC mode module. See the *BSP API* document (in the doxygen folder of the documentation package), for additional information.

Chapter 23
Universal Asynchronous Receiver-Transmitter (UART) Driver

Thei.MX28 board contains six serial Universal Asynchronous Receiver-Transmitters (UARTs). One UART has no DMA support and is intended to be used as a debug console (Debug UART). Five UARTs are a high-performance UARTs, which are intended to be used by applications (Application UART, appUART). They offers similar functionality to the industry-standard 16C550 UART device and support baud rates of up to 3.25 Mbits/s. Unlike the debug UART, the application UARTs cannot be used as a serial console.

23.1 Application UART

The following sections describe the hardware and software operation as well as the code structure of the Application UARTs.

23.1.1 Hardware Operation

The CPU or the DMA controller reads and writes data and control/status information through the APBX interface. The transmit and receive paths are buffered with internal FIFO memories, enabling up to 16-bytes to be stored independently in both transmit and receive modes. Two DMA channels are supported, one for transmit and one for receive. If a time-out condition occurs in the middle of a receive DMA block transfer, then the UART ends the DMA transfer and signals the end of the DMA block transfer. A receive DMA can be set up to get the status of the previous receive DMA block transfer. The status indicates the amount of valid data bytes in the previous receive DMA block transfer.

If a framing, parity, or break error occurs during reception, the appropriate error bit is set and stored in the FIFO. If an overrun condition occurs, the overrun register bit is set immediately and FIFO data is prevented from being overwritten. The FIFOs can be programmed to be one-byte deep, providing a conventional double-buffered UART interface. The modem status input signal Clear To Send (CTS) and output modem control line Request To Send (RTS) are supported. A programmable hardware flow control feature uses the nUARTCTS input and the nUARTRTS output to automatically control the serial data flow.

23.1.2 Software Operation

The application UART driver is implemented as a UART driver registered with a UART core in the Linux kernel and thus provides a standard serial driver interface to Linux. The driver can operate in both PIO mode and DMA mode. DMA mode is the default and it allows the use of the FIFO in an optimum manner. For more details, refer to `Documentation/serial/driver`. The driver does not support a console on the application UART port.

23.1.3 Source Code Structure

The application UART driver consists of the following files:

```
drivers/serial/mxs-auart.c
drivers/serial/mxs-auart.h
```

23.2 Debug UART

The following sections describe the hardware and software operation as well as the code structure of the Debug UART.

23.2.1 Hardware Operation

The debug UART performs:

- Serial-to-parallel conversion on data received from a peripheral device
- Parallel-to-serial conversion on data transmitted to the peripheral device

The CPU reads and writes data and control/status information through the APBX interface. The transmit and receive paths are buffered with internal FIFO memories.

23.2.2 Software Operation

The debug UART driver is implemented as a UART driver registered with UART core in the Linux kernel and thus provides a standard serial driver interface to Linux. The driver operates in interrupt mode and uses the FIFO in an optimum manner. Refer to `Documentation/serial/driver` for more details. The driver supports a console on the debug UART port.

23.2.3 Source Code Structure

The debug UART driver consists of the following files:

```
drivers/serial/mxs-duart.c
drivers/serial/mxs-duart.h
```

23.3 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_SERIAL_MXS_AUART = [y|m]
  Configuration option to enable the application UART driver.
- CONFIG_SERIAL_MXS_DUART = [y|m]
  Configuration option to enable the debug UART driver.
- CONFIG_SERIAL_MXS_DBG_CONSOLE
  Configuration option to enable the console on the debug UART.

Chapter 24
ARC USB Driver

The universal serial bus (USB) driver implements a standard Linux driver interface to the ARC USB-HS OTG controller. The USB provides a universal link that can be used across a wide range of PC-to-peripheral interconnects. It supports plug-and-play, port expansion, and any new USB peripheral that uses the same type of port.

The ARC USB controller is enhanced host controller interface (EHCI) compliant. This USB driver has the following features:

- High Speed/Full Speed Host Only core (HOST1)
- Host mode—Supports HID (Human Interface Devices), MSC (Mass Storage Class), and PTP (Still Image) drivers
- Peripheral mode—Supports MSC, and CDC (Communication Devices Class) drivers
- Embedded DMA controller

24.1 Architectural Overview

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

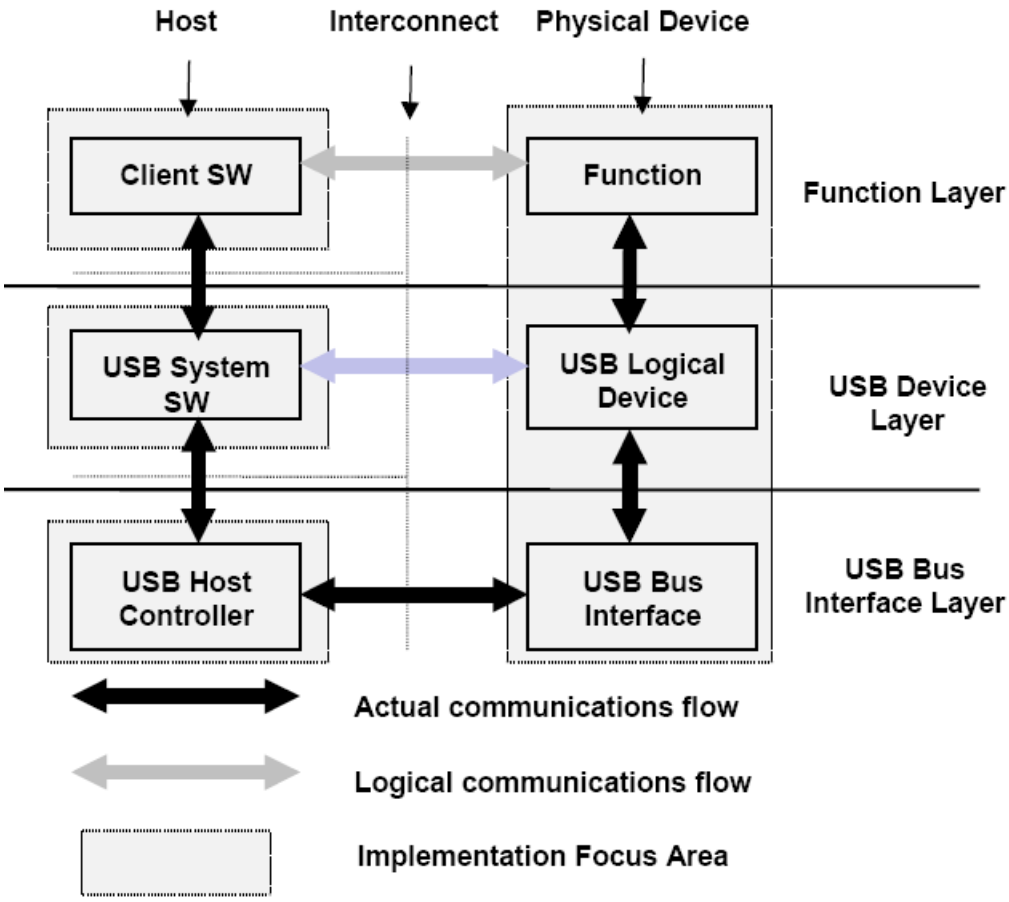

Figure 24-1. USB Block Diagram

24.2 Hardware Operation

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

The i.MX28 EVK has a single Micro-AB receptacle and standard A. Mirco-AB can accept either a type Micro-A (i.MX28 acts as a USB host) or Micro-B (i.MX28 acts as an USB gadget) plug. The A-type receptacle has the 5th pin grounded while this pin on the B-type is floating. The state of this pin can be read from the USBPHY STATUS register. When the pin state is changed, the USB control interrupt is triggered. The standard A port is dedicate USB host only port

24.3 Software Operation

The Linux OS contains a USB driver, which implements the USB protocols. For the USB host, it only implements the hardware specified initialization functions. For the USB peripheral, it implements the gadget framework.

```
static struct usb_ep_ops fsl_ep_ops = {
         .enable = fsl_ep_enable,
         .disable = fsl_ep_disable,
         .alloc_request = fsl_alloc_request,
         .free_request = fsl_free_request,
         .queue = fsl_ep_queue,
         .dequeue = fsl_ep_dequeue,
         .set_halt = fsl_ep_set_halt,
         .fifo_status = arcotg_fifo_status,
         .fifo_flush = fsl_ep_fifo_flush,            /* flush fifo */
         };
static struct usb_gadget_ops fsl_gadget_ops = {
         .get_frame = fsl_get_frame,
         .wakeup = fsl_wakeup,
/*       .set_selfpowered = fsl_set_selfpowered, */   /* Always selfpowered */
         .vbus_session = fsl_vbus_session,
         .vbus_draw = fsl_vbus_draw,
         .pullup = fsl_pullup,
         };
```

- `fsl_ep_enable`—configures an endpoint making it usable
- `fsl_ep_disable`—specifies an endpoint is no longer usable
- `fsl_alloc_request`—allocates a request object to use with this endpoint
- `fsl_free_request`—frees a request object
- `arcotg_ep_queue`—queues (submits) an I/O request to an endpoint
- `arcotg_ep_dequeue`—dequeues (cancels, unlinks) an I/O request from an endpoint
- `arcotg_ep_set_halt`—sets the endpoint halt feature
- `arcotg_fifo_status`—get the total number of bytes to be moved with this transfer descriptor

For OTG, an OTG finish state machine (FSM) is implemented.

24.4 Driver Features

The USB stack supports the following features:

- USB device mode
- Mass storage device profile—subclass 8-1 (RBC set)
- USB host mode
- HID host profile—subclasses 3-1-1 and 3-1-2. (USB mouse and keyboard)
- Mass storage host profile—subclass 8-1
- Ethernet USB profile—subclass 2
- DC PTP transfer

24.5 Source Code Structure

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

Table 24-1. USB Driver Files

File	Description
host/ehci-hcd.c	Host driver source file
host/ehci-arc.c	Host driver source file
host/ehci-mem-iram.c	Host driver source file for IRAM support
host/ehci-hub.c	Hub driver source file
host/ehci-mem.c	Memory management for host driver data structures
host/ehci-q.c	EHCI host queue manipulation
host/ehci-q-iram.c	Host driver source file for IRAM support
gadget/arcotg_udc.c	Peripheral driver source file
gadget/arcotg_udc.h	USB peripheral/endpoint management registers
otg/fsl_otg.c	OTG driver source file
otg/fsl_otg.h	OTG driver header file
otg/otg_fsm.c	OTG FSM implement source file
otg/otg_fsm.h	OTG FSM header file
gadget/fsl_updater.c	FSL manufacture tool usb char driver source file
gadget/fsl_updater.h	FSL manufacture tool usb char driver header file

Table 24-2 shows the platform related source files.

Table 24-2. USB Platform Source Files

File	Description
arch/arm/plat-mxs/include/mach/arc_otg.h	USB register define
include/linux/fsl_devices.h	FSL USB specific structures and enums

Table 24-3 shows the platform-related source files in the directory:`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx28/`

Table 24-3. USB Platform Header Files

File	Description
usb_dr.c	Platform-related initialization
usb_h1.c	Platform-related initialization

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

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

Table 24-4. USB Common Platform Files

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

24.6 Menu Configuration Options

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

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

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

  By default, this option is M.

  CONFIG_USB_EHCI_ARC—Build support for selecting the ARC EHCI host. In menuconfig, this option is available under Device drivers > USB support > Support for Freescale controller.

  By default, this option is Y.
- CONFIG_USB_EHCI_ARC_H1—Build support for selecting the USB Host1. In menuconfig, this option is available underDevice drivers > USB support > Support for Host1 port on Freescale controller. By default, this option is Y.
- CONFIG_USB_EHCI_ARC_OTG—Build support for selecting the ARC EHCI OTG host. In menuconfig, this option is available under

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

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

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

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

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

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

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

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

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

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

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

  By default, this option is M.

24.7 Programming Interface

This driver implements all the functions that are required by the USB bus protocol to interface with the i.MX USB ports. See the *BSP API* document, for more information.

24.8 Default USB Settings

Table 24-5 shows the default USB settings.

Table 24-5. Default USB Settings

Platform	OTG HS	OTG FS	Host1	Host2(HS)	Host2(FS)
i.MX50 EVK	enable	N/A	enable (HS)	N/A	N/A
i.MX28 EVK	enabled	NA	enable	enable	—

- Build USB GADGET driver as M, for example:
  CONFIG_USB_ETH CONFIG_USB_FILE_STORAGE then, if you want to use EVK as mass storage device, insmod g_file_storage.ko file=/dev/mmcblk0p2
  if you want to use the otg as ethernet, insmod g_ether.ko, then you can use ifconfig usb0 to configure the ip

24.9 Remote WakeUp

- OTG device do not support SET/CLEAR_FEATURE Remote-wakeup
- HOST support Remote-wakeup by usb device

24.10 System WakeUp

- Both host and device connect/disconnect event can be system wakeup source

Chapter 25
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)

25.1 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.

25.2 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.

25.3 Source Code Structure

The RTC module is implemented in the `<ltib_dir>/rpm/BUILD/linux/drivers/rtc` directory. Table 25-1 shows the RTC module files. The source file for the RTC specifies the RTC function implementations.

Table 25-1. RTC Driver File List

File	Description
rtc-mxs.c	RTC driver

25.4 Programming Interface

All the Linux RTC functions are based on `rtclib`. The `include/linux/rtc.h` file specifies all the IOCTLs for the RTC. Table 25-1 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.

The following Linux kernel configuration options are provided for this module:

- CONFIG_RTC_DRV_MXS [=M|Y]

  This is the configuration option for the RTC driver, which is dependent on the RTC_CLASS option. In menuconfig, this option is available under: Real Time Clock > Freescale MXS series SoC RTC

Chapter 26
Watchdog (WDOG) Driver

The Watchdog Timer module protects against system failures by providing an escape from unexpected hang, infinite loop situations or programming errors.

26.1 Hardware Operation

Once the Watchdog Timer is activated, it must be serviced by software on a periodic basis. If servicing does not take place in time, the watchdog times out. Upon a time-out, the watchdog resets the chip.

26.2 Software Operation

The Watchdog module software implementation conforms with the Linux watchdog driver model. Besides the initialization function, it provides IOCTL and write functions to set up and maintain the watchdog timer. Refer to `Documentation/watchdog/watchdog-api.txt` for full information on the Linux Watchdog API.

Chapter 27
Battery Charger and Power Source Manager (PSM) Driver

The Battery Charger Device Driver for Linux provides support for controlling the battery interface circuits and power source detection. The battery charger features include:

- Circuits to automatically detect the presence of a USB or AC power source and to recharge the system battery
- Prevents overcharging circuits
- Bad battery detection
- Die temperature monitoring
- Battery voltage measurement

27.1 Hardware Operation

The i.MX28 device includes charging of a Li-Ion battery from a 5 V source. The battery charger is essentially a linear regulator with current and voltage limits. Charge current is software-programmable. Li-Ion batteries can be charged at the lower of 1 C (where C is the charging rate of a battery; in other words, transfer of all of the stored energy in one hour), 785 mA, or the VDD5V current limit. USB charging is typically limited to 500 mA or less to meet compliance requirements.

Typical charge times for a Li-Ion battery are 1.5 to 3 hours with more than 70% of the charge delivered in the first hour. The battery charge voltage limit is determined by the battery type. The Li-Ion charge is typically stopped after a certain time limit or when the charging current drops below 10% of the charge current setting. It includes controls for the maximum charge current and for the stop charge current. The charger avoids exceeding the charge voltage limit on the battery.

The battery voltage can be monitored using the Low Resolution ADC (LRADC). The charger has its own (very robust) voltage limiting that operates independently of the LRADC. The battery charger is capable of generating a large amount of heat within the i.MX28, especially at currents above 400 mA. The dissipated power can be estimated as: (5 V – battery_voltage) × current. At maximum current (785 mA) and a 3 V battery, the charger can dissipate 1.57 W, raising the die temp as much as 80 °C. To ensure that the system operates correctly, the die temperature sensor should be monitored every 100 ms. If the die temperature exceeds 115 °C (the maximum value for the chip temperature sensor), then the battery charge current must be reduced.

27.2 Software Operation

The function `ddi_bc_Init()` configures the battery charger and moves the state machine from uninitialized to disabled. This is the only way to move the state machine out of the uninitialized state. This function is used to start up the battery charger after a system reset or when reloading it into memory.

The structure below shows the configuration information for the `ddi_bc_Init()`:

```
typedef struct _ddi_bc_Cfg_t
{
    uint32_t u32StateMachinePeriod;
    uint16_t u16CurrentRampSlope;
    uint16_t u16ConditioningThresholdVoltage;
    uint16_t u16ConditioningMaxVoltage;
    uint16_t u16ConditioningCurrent;
    uint32_t u32ConditioningTimeout;
    uint16_t u16ChargingVoltage;
    uint16_t u16ChargingCurrent;
    uint16_t u16ChargingThresholdCurrent;
    uint32_t u32ChargingTimeout;
    uint32_t u32TopOffPeriod;
    uint8_t useInternalBias:1;
    uint8_t monitorDieTemp:1;
    uint8_t monitorBatteryTemp:1;
    int8_t u8DieTempHigh;
    int8_t u8DieTempLow;
    uint16_t u16DieTempSafeCurrent;
    uint8_t u8BatteryTempChannel;
    uint16_t u16BatteryTempHigh;
    uint16_t u16BatteryTempLow;
    uint16_t u16BatteryTempSafeCurrent;
} ddi_bc_Cfg_t;
```

Table 27-1 shows the structure field definitions.

Table 27-1. Battery Charger Driver Structure Fields

Field	Units	Description
u32StateMachinePeriod	ms	Expected period between calls to `ddi_bc_StateMachine`. If die temperature monitoring is enabled, then this period should be around 100 ms or less.
u16CurrentRampSlope	mA/s	Configures the slope of the current ramp. When the battery charger increases its current draw, it ramps up the current to this rate.
u16ConditioningThresholdVoltage	mV	Configures the threshold conditioning voltage. If the battery voltage is below this value, it is conditioned until its voltage rises above the maximum conditioning voltage (ConditioningMaxVoltage). After that, the battery is charged normally.
u16ConditioningMaxVoltage	mV	Configures the maximum conditioning voltage. Normal charging begins when the voltage rises above this value. This value should be slightly higher than the threshold conditioning voltage because it is measured while a conditioning current is actually owing to the battery. With a conditioning current of 0.1×C (where C is the battery capacity), reasonable values for the threshold and maximum conditioning voltages are 2.9 V and 3.0 V respectively.
u16ConditioningCurrent	mA	Configures the maximum conditioning current. This is the maximum current that is offered to a battery while it is being conditioned. A typical value is 0.1×C.
u16ConditioningTimeout	ms	Configures the conditioning time-out. This is the maximum amount of time that a battery is conditioned before the battery charger declares it to be broken.
u16ChargingVoltage	mV	Configures the final charging voltage. Only two values are permitted: 4100 or 4200.
u16ChargingCurrent	mA	Configures the maximum current offered to a charging battery.

Table 27-1. Battery Charger Driver Structure Fields (continued)

Field	Units	Description
u16ChargingThresholdCurrent	mA	Configures the current flow below which a charging battery is regarded as fully charged (typical 0.1×C). At this point, the battery is topped off.
u32ChargingTimeout	ms	Configures the charging time-out. This is the maximum amount of time that a battery is charged before the battery charger declares it to be broken.
u32TopOffPeriod	ms	Configures the top-off period. This is the amount of time a battery is held in the Topping Off state before it is declared fully charged.
useInternalBias	—	A value of zero causes the battery charger to use an externally generated bias current, which is expected to be quite precise. Any other value cause the battery charger to generate a lesser-quality bias current internally.
monitorDieTemp	—	If set, this field indicates that the battery charger is to monitor the die temperature. See below for fields that configure the details.
monitorBatteryTemp	—	If set, this field indicates that the battery charger has to monitor the battery temperature. See below for fields that configure the details.
u8DieTempHigh	°C	If the battery charger is monitoring the die temperature, and it rises to a range that includes a temperature greater than or equal to this value, the charging current is clamped to the safe current.
u8DieTempLow	°C	If the charging current is being clamped because of high die temperature, and it falls to a range that does not include a temperatures greater than or equal to this value, the charging current clamp is released.
u16DieTempSafeCurrent	mA	If the battery charger detects a high die temperature, it clamps the charging current at or below this value.
u8BatteryTempChannel	—	If the battery charger is monitoring the battery temperature, this field indicates the LRADC channel to read.
u8BatteryTempHigh	—	If the battery charger is monitoring the battery temperature, and it rises to a measurement greater than or equal to this value, the charging current is clamped to the corresponding safe current.
u8BatteryTempLow	—	If the charging current is being clamped because of a high battery temperature, and it falls below this value, the charging current clamp is released.
u16BatteryTempSafeCurrent	mA	If the battery charger detects a high battery temperature, it clamps the charging current at or below this value.

27.3 Source Code Structure

The battery charger driver code listed in Table 27-2, is located in:

```
drivers/power/stmp37xx/
arch/arm/mach-stmp3xxx/include/mach/
```

Table 27-2. Battery Charger Driver Files

File	Description
ddi_bc_api.c	Battery charger API
ddi_bc_hw.c	Battery charger hardware operations

Table 27-2. Battery Charger Driver Files (continued)

File	Description
ddi_bc_hw.h	Declarations for battery charger hardware operations
ddi_bc_init.c	Battery charger initialization function
ddi_bc_internal.h	Declarations for the battery charger driver
ddi_bc_ramp.c	Battery charger current ramp controller
ddi_bc_ramp.h	Declarations for battery current ramp controller
ddi_bc_sm.c	Battery charger state machine
ddi_bc_sm.h	Declarations for the battery charger state machine
ddi_power_battery.c	Power manipulations for the battery charger driver
ddi_power_battery.h	Declarations for ppower manipulations
linux.c	Linux glue driver to the battery state machine
ddi_bc.h	Header file with externally visible declarations for the battery charger driver

27.4 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_POWER_SUPPLY = [y | m]
  Configuration option to enable power supply class support in the kernel.
- CONFIG_BATTERY_STMP3XXX = [y | m]
  Configuration option to enable the battery charger driver.

Chapter 28
LED Pulse Width Modulator (PWM) Driver

The MXS Pulse Width Modulator (PWM) module provides eight PWM interfaces for system use. The PWM LED driver provides a standard framework to control LEDs attached to PWM interfaces.

28.1 Hardware Operation

The PWM interface is clocked from a 24 MHz reference clock. Each PWM channel can be independently programmed with the following characteristics:

- Reference clock divider
- Pulse active on and off time based on the divided reference clock
- Pulse period based on the divided reference clock

28.2 Software Operation

The PWM LED driver software implementation conforms with the Linux LED driver model. The driver provides the functions necessary to register as a standard Linux LED class driver. Refer to `Documentation/leds-class.txt` for full information on the Linux LED Class API. The driver uses the PWM active time and period registers to implement an 8-bit resolution brightness control.

28.3 Menu Configuration Options

The following Linux kernel configuration options are provided for this module:

- CONFIG_LEDS_MXS [= M|Y]

  This is the configuration option for the PWM LED driver, which is dependent on the LEDS_CLASS option.

  In menuconfig, this option is available under:

  LED Support > Support for PWM LEDs on MXS

28.4 Source Code Structure

The PWM LED driver is implemented in the following files:

```
drivers/leds/leds-mxs-pwm.c
```

Chapter 29
Frequently Asked Questions

29.1 NFS Mounting Root File System

1. Assuming the root file system is under, modify the `/etc/exports` file on the Linux host by adding the following line:
   /tmp/fs *(rw,no_root_squash)/tools/rootfs *(rw,sync,no_root_squash)
2. Make sure the NFS service is started on the Linux host machine. To start it on the host machine, issue:

   Install if not already installed.

29.2 Using 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.