i.MX 6Dual/6Quad Linux Reference Manual

Document Number: IMX6QLXRM
Rev L3.0.35_4.1.0, 09/2013

freescale™

Contents

Section number **Title** **Page**

Chapter 1
About this Book

Chapter 2
Machine Specific Layer (MSL)

Chapter 3
Smart Direct Memory Access (SDMA) API

Chapter 4
AHB-to-APBH Bridge with DMA (APBH-Bridge-DMA)

Chapter 5
Image Processing Unit (IPU) Drivers

Chapter 6
MIPI DSI Driver

Chapter 7
LVDS Display Bridge(LDB) Driver

Chapter 8
Video for Linux Two (V4L2) Driver

Chapter 9
Graphics Processing Unit (GPU)

Chapter 10
Direct FB

Chapter 11
HDMI Driver

Chapter 12
X Windows Acceleration

Chapter 13
Video Processing Unit (VPU) Driver

Chapter 14
OmniVision Camera Driver

Chapter 15
MIPI CSI2 Driver

Chapter 16
Low-level Power Management (PM) Driver

Chapter 17
PF100 Regulator Driver

Chapter 18
CPU Frequency Scaling (CPUFREQ) Driver

Chapter 19
Dynamic Voltage Frequency Scaling (DVFS) Driver

Chapter 20
Thermal Driver

Chapter 21
Anatop Regulator Driver

Chapter 22
SNVS Real Time Clock (SRTC) Driver

Chapter 23
Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver

Chapter 24
Asynchronous Sample Rate Converter (ASRC) Driver

Chapter 25
The Sony/Philips Digital Interface (S/PDIF) Driver

Chapter 26
SPI NOR Flash Memory Technology Device (MTD) Driver

Chapter 27
MMC/SD/SDIO Host Driver

Chapter 28
NAND GPMI Flash Driver

Chapter 29
SATA Driver

Chapter 30
Inter-IC (I2C) Driver

Chapter 31
Enhanced Configurable Serial Peripheral Interface (ECSPI) Driver

Chapter 32
FlexCAN Driver

Chapter 33
Media Local Bus Driver

Chapter 34
ARC USB Driver

Chapter 35
i.MX 6 PCI Express Root Complex Driver

Chapter 36
WEIM NOR Driver

Chapter 37
Fast Ethernet Controller (FEC) Driver

Chapter 38
ENET IEEE-1588 Driver

Chapter 39
Universal Asynchronous Receiver/Transmitter (UART) Driver

Chapter 40
AR6003 WiFi

Chapter 41
Bluetooth Driver

Chapter 42
Pulse-Width Modulator (PWM) Driver

Chapter 43
Watchdog (WDOG) Driver

Chapter 44
OProfile

Chapter 45
CAAM (Cryptographic Acceleration and Assurance Module)

Chapter 1
About this Book

1.1 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 3.0 kernel internals, driver models, and i.MX processors.

1.1.1 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.

1.1.2 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
ARMÂ®	Advanced RISC Machines processor architecture

Table continues on the next page...

Term	Definition
AUDMUX	Digital audio MUX-provides a programmable interconnection for voice, audio, and synchronous data routing between host serial interfaces and peripheral serial interfaces
BCD	Binary Coded Decimal
bus	A path between several devices through data lines
bus load	The percentage of time a bus is busy
CODEC	Coder/decoder or compression/decompression algorithm-used to encode and decode (or compress and decompress) various types of data
CPU	Central Processing Unit-generic term used to describe a processing core
CRC	Cyclic Redundancy Check-Bit error protection method for data communication
CSI	Camera Sensor Interface
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

Table continues on the next page...

Term	Definition
IrDA	Infrared Data Association-a nonprofit organization whose goal is to develop globally adopted specifications for infrared wireless communication
ISR	Interrupt Service Routine
JTAG	JTAG (IEEE Standard 1149.1) A standard specifying how to control and monitor the pins of compliant devices on a printed circuit board
Kill	Abort a memory access
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
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

Table continues on the next page...

Term	Definition
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

Chapter 2
Machine Specific Layer (MSL)

2.1 Introduction

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

- Interrupts including GPIO and EDIO (only on certain platforms)
- Timer
- Memory map
- General Purpose Input/Output (GPIO) including IOMUX on certain platforms
- Shared Peripheral Bus Arbiter (SPBA)
- Smart Direct Memory Access (SDMA)

These modules are normally available in the following directory:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6 for i.MX 6 platform
```

The header files are implemented under the following directory:

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

The MSL layer contains not only the modules common to all the boards using the same processor, such as the interrupts and timer, but it also contains modules specific to each board, such as the memory map. The following sections describe the basic hardware and software operations and the software interfaces for MSL modules. First, the common modules, such as Interrupts and Timer are discussed. Next, the board-specific modules, such as Memory Map and General Purpose Input/Output (GPIO) (including IOMUX on some platforms) are detailed. Because of the complexity of the SDMA module, its design is explained in SDMA relevant chapter.

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

2.2 Interrupts (Operation)

This section explains the hardware and software operation of interrupts on the device.

2.2.1 Interrupt Hardware Operation

The Interrupt Controller controls and prioritizes a maximum of 128 internal and external interrupt sources.

Each source can be enabled or disabled by configuring the Interrupt Enable Register or using the Interrupt Enable/Disable Number Registers. When an interrupt source is enabled and the corresponding interrupt source is asserted, the Interrupt Controller asserts a normal or a fast interrupt request depending on the associated Interrupt Type Register settings.

Interrupt Controller registers can only be accessed in supervisor mode. The Interrupt Controller interrupt requests are prioritized in the following order: fast interrupts and normal interrupts for the highest priority level, then highest source number with the same priority. There are 16 normal interrupt levels for all interrupt sources, with level zero being the lowest priority. The interrupt levels are configurable through eight normal interrupt priority level registers. Those registers, along with the Normal Interrupt Mask Register, support software-controlled priority levels for normal interrupts and priority masking.

2.2.2 Interrupt Software Operation

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

The ARM Linux implementation chooses the high vector address model.

The following file describes 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.

2.2.3 Interrupt Features

The interrupt implementation supports the following features:

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

2.2.4 Interrupt Source Code Structure

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

```
irq.c (If CONFIG_MXC_TZIC is not selected)
tzic.c (If CONFIG_MXC_TZIC is selected)
gic.c (If CONFIG_ARM_GIC is selected)
```

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

```
hardware.h
irqs.h
```

The following table lists the source files for interrupts.

Table 2-1. Interrupt Files

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

2.2.5 Interrupt Programming Interface

The machine-specific interrupt implementation exports a single function.

This function initializes the Interrupt Controller hardware and registers functions for interrupt enable and disable from each interrupt source.

This is done with the global structure irq_desc of type struct irqdesc. After the initialization, the interrupt can be used by the drivers through the request_irq() function to register device-specific interrupt handlers.

In addition to the native interrupt lines supported by the Interrupt Controller, the number of interrupts is also expanded to support GPIO interrupt and (on some platforms) EDIO interrupts. This allows drivers to use the standard interrupt interface supported by ARM Linux, such as the request_irq() and free_irq() functions.

2.3 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 performs the following operations:

- Updates the system uptime.
- Updates the time of day.
- Reschedules a new process if the current process has exhausted its time slice.
- Runs any dynamic timers that have expired.
- Updates resource usage and processor time statistics.

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

2.3.1 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 Timer. Another function provides the time elapsed as the last timer interrupt.

2.3.2 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.

2.3.3 Timer Source Code Structure

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

2.3.4 Timer Programming Interface

The timer module utilizes four hardware timers, to implement clock source and clock event objects.

This is done with the clocksource_mxc structure of struct clocksource type and clockevent_mxc structure of struct clockevent_device 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.

2.4 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.

2.4.1 Memory Map Hardware Operation

The MMU, as a 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.

2.4.2 Memory Map Software Operation

A table mapping the virtual memory to physical memory is implemented for i.MX platforms as defined in the file in <ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/mm.c .

2.4.3 Memory Map Features

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

2.4.4 Memory Map Source Code Structure

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

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

The following table lists the source files for the memory map.

Table 2-2. Memory Map Files

File	Description
mx6.h	Header files for the I/O module physical addresses
mm.c	Memory map definition file

2.4.5 Memory Map Programming Interface

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

2.5 IOMUX

The limited number of pins of highly integrated processors can have multiple purposes.

The IOMUX module controls a pin usage so that the same pin can be configured for different purposes and can be used by different modules.

This is a common way to reduce the pin count while meeting the requirements from various customers. Platforms that do not have the IOMUX hardware module can do pin muxing through the GPIO module.

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

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

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose and cannot be changed by software. Otherwise, the IOMUX module needs to be configured to serve a particular purpose that is dictated by the system (board) design.

- If the pin is connected to an external UART transceiver and therefore to be used as the UART data transmit signal, it should be configured as the primary function.
- If the pin is connected to an external Ethernet controller for interrupting the ARM core, it should be configured as GPIO input pin with interrupt enabled.

The software does not have control over what function a pin should have. The software only configures pin usage according to the system design.

2.5.1 IOMUX Hardware Operation

The following information applies only to those processors that have an IOMUX hardware module.

The IOMUX controller registers are briefly described in this section.

For detailed information, see the pin multiplexing section of the IC reference manual.

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

2.5.2 IOMUX Software Operation

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

2.5.3 IOMUX Features

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

2.5.4 IOMUX Source Code Structure

The following table lists the source files for the IOMUX module. The files are in the directory:

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

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

Table 2-3. IOMUX Files

File	Description
iomux-v3.c	IOMUX function implementation
iomux-mx6q.h	Pin definitions in the iomux_pins enum

2.5.5 IOMUX Programming Interface

All the IOMUX functions required for the Linux port are implemented in the iomux-v3.c file.

2.5.6 IOMUX Control Through GPIO Module

For a multi-purpose pin, the GPIO controller provides the multiplexing control so that each pin may be configured either as a functional pin or a GPIO pin.

The operation of the functional pin, which can be subdivided into either major function or one alternate function, is controlled by a specific hardware module. If it is configured as a GPIO pin, the pin is controlled by the user through software with further configuration

through the GPIO module. In addition, there are some special configurations for a GPIO pin (such as output based A_IN, B_IN, C_IN or DATA register, but input based A_OUT or B_OUT).

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

If the hardware modes are chosen at the system integration level, this pin is dedicated only to that purpose which can not be changed by software. Otherwise, the GPIO module needs to be configured properly to serve a particular purpose that is dictated with the system (board) design.

- If this pin is connected to an external UART transceiver, it should be configured as the primary function.
- If this pin is connected to an external Ethernet controller for interrupting the core, it should be configured as GPIO input pin with interrupt enabled.

The software does not have control over what function a pin should have. The software only configures a pin for that usage according to the system design.

2.5.6.1 GPIO Hardware Operation

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

2.5.6.1.1 Muxing Control

The GPIO In Use Registers control a multiplexer in the GPIO module.

The settings in these registers choose if a pin is utilized for a peripheral function or for its GPIO function. One 32-bit general purpose register is dedicated to each GPIO port. These registers may be used for software control of IOMUX block of the GPIO.

2.5.6.1.2 PULLUP Control

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

2.5.6.2 GPIO Software Operation (general)

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

2.5.6.3 GPIO Implementation

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

2.5.6.4 GPIO Source Code Structure

The GPIO module is implemented in the iomux.cgpio_mux.c file under the relevant MSL directory. The header file to define the pin names is under:

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

The following table lists the source files for the IOMUX.

Table 2-4. IOMUX Through GPIO Files

File	Description
iomux-mx6q.h	Pin name definitions

2.5.6.5 GPIO Programming Interface

All the GPIO muxing functions required for the Linux port are implemented in the iomux-v3.c file.

2.6 General Purpose Input/Output(GPIO)

The GPIO module provides general-purpose pins that can be configured as either inputs or outputs.

When configured as an output, the pin state (high or low) can be controlled by writing to an internal register. When configured as an input, the pin input state can be read from an internal register.

2.6.1 GPIO Software Operation

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

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

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

2.6.1.1 API for GPIO

API for GPIO lists the features supported by the GPIO implementation.

The GPIO implementation supports the following features:

- An API for registering an interrupt service routine to a GPIO interrupt. This is made possible as the number of interrupts defined by NR_IRQS is expanded to accommodate all the possible GPIO pins that are capable of generating interrupts.
- Functions to request and free an IOMUX pin. If a pin is used as GPIO, another set of request/free function calls are provided. The user should check the return value of the request calls to see if the pin has already been reserved before modifying the pin state. The free function calls should be made when the pin is not needed. See the API document for more details.
- Aligned parameter passing for both IOMUX and GPIO function calls. In this implementation the same enumeration for iomux_pins is used for both IOMUX and GPIO calls and the user does not have to figure out in which bit position a pin is located in the GPIO module.
- Minimal changes required for the public drivers such as Ethernet and UART drivers as no special GPIO function call is needed for registering an interrupt.

2.6.2 GPIO Features

This GPIO implementation supports the following features:

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

2.6.3 GPIO Module Source Code Structure

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

Table 2-5. GPIO Files

File	Description
iomux-mx 6q.h	IOMUX common header file
gpio.h	GPIO public header file
gpio.c	Function implementation

2.6.4 GPIO Programming Interface 2

For more information, see the Documentation/gpio.txt under the Linux source code directory for the programming interface.

Chapter 3
Smart Direct Memory Access (SDMA) API

3.1 Overview

The Smart Direct Memory Access (SDMA) API driver controls the SDMA hardware.

It provides an API to other drivers for transferring data between MCU memory space and the peripherals. It supports the following features:

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

3.1.1 Hardware Operation

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

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

3.1.2 Software Operation

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

The table below provides a list of drivers that use SDMA and the number of SDMA physical channels used by each driver. A driver can specify the SDMA channel number that it wishes to use, which is called static channel allocation. It can also have the SDMA driver and provide a free SDMA channel for the driver to use, which is called dynamic channel allocation. For dynamic channel allocation, the list of SDMA channels is scanned from channel 32 to channel 1. Upon finding a free channel, that channel is allocated for the requested DMA transfers.

Table 3-1. SDMA Channel Usage

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

3.1.3 Source Code Structure

The dmaengine.h (header file for SDMA API) is available in the directory /<ltib_dir>/rpm/BUILD/linux/include/linux

The following table shows the source files available in the directory /<ltib_dir>/rpm/BUILD/linux/drivers/dma

Table 3-2. SDMA API Source Files

File	Description
dmaengine.c	SDMA management routine
imx-sdma.c	SDMA implement driver

The following table shows the image files available in the directory /<ltib_dir>/rpm/BUILD/linux/firmware/imx/sdma

Table 3-3. SDMA Script Files

File	Description
sdma-mx6q-to1.bin.ihex	SDMA RAM scripts

3.1.4 Menu Configuration Options

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

- CONFIG_IMX_SDMA_: This is the configuration option for the SDMA API driver. In menuconfig, this option is available under DMA Engine support.
- System type > Freescale MXC implementations > MX6 Options: > Use SDMA API.
- By default, this option is Y.

3.1.5 Programming Interface

The module implements standard DMA API. For more information on the functions implemented in the driver, refer to the API documents, which are included in the Linux documentation package. For additional information, refer to the ESAI driver.

3.1.6 Usage Example

Refer to one of the drivers, such as SPDIF driver, UART driver or SSI driver, that uses the SDMA API driver as a usage example.

Chapter 4
AHB-to-APBH Bridge with DMA (APBH-Bridge-DMA)

4.1 Overview

The AHB-to-APBH bridge provides the processor with an inexpensive peripheral attachment bus running on the AHB's HCLK.

(The H in APBH indicates that the APBH is synchronous to HCLK.)

The AHB-to-APBH bridge includes the AHB-to-APB PIO bridge for a memory-mapped I/O to the APB devices, a central DMA facility for devices on this bus and a vectored interrupt controller for the ARM core. Each one of the APB peripherals, including the vectored interrupt controller, is documented in its own chapter in this document.

There is no separated DMA bus for these devices. An internal arbitration logic solves the conflict that occurs when the DMA uses the APBH bus and the AHB-to-APB bridge functions use the APBH. For conflict between these two units, the DMA is master and the AHB is standby, which will report "not ready" through its HREADY output until the bridge transfer is complete. The arbiter tracks repeated lockouts and inverts the priority, guaranteeing the ARM platform every four rounds of transfer on the APB.

4.1.1 Hardware Operation

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

- Multi-channel DMA supporting up to 32 time-division multiplexed DMA channels
- Powered by a 16-bit Instruction-Set micro-RISC engine
- Each channel executes specific script
- Very fast context-switching with preemptive multi-tasking based on two-level priority

- 4-KB ROM containing startup scripts (that is, boot code) and other common utilities that can be referenced by RAM-located scripts
- 8-KB RAM area divided into a processor context area and a code space area used to store channel scripts that are downloaded from the system memory.

4.1.2 Software Operation

The DMA supports 16 channels of DMA services, as shown in the following table. The shared DMA resource allows each independent channel to follow a simple chained command list. Command chains are built up by using the general structure.

Table 4-1. APBH DMA Channel Assignments

APBH DMA Channel #	Usage
0	GPMI0
1	GPMI1
2	GPMI2
3	GPMI3
4	GPMI4
5	GPMI5
6	GPMI6
7	GPMI7
8	EMPTY
9	EMPTY
10	EMPTY
11	EMPTY
12	EMPTY
13	EMPTY
14	EMPTY
15	EMPTY

4.1.3 Source Code Structure

The following table shows the source files available in the directory drivers/dma/

Table 4-2. APBH DMA Source Files

File	Description
mxs-dma.c	APBH DMA implement driver

4.1.4 Menu Configuration Options

MXS_DMA is the configuration option for the APBH DMA driver. In menu configuration, this option is available under Device Drivers > DMA Engine support > MXS DMA support.

4.1.5 Programming Interface

The module implements standard DMA API. For more information on the functions implemented in the driver such as GPMI NAND driver, refer to the API documents, which are located in the Linux documentation package.

4.1.6 Usage Example

Refer to one of the drivers, such as the GPMI NAND driver, that uses the APBH DMA driver as a usage example.

Chapter 5
Image Processing Unit (IPU) Drivers

5.1 Introduction

The image processing unit (IPU) is designed to support video and graphics processing functions and to connect with video and still image sensors and displays. The IPU driver provides a kernel-level API to manipulate logical channels. A logical channel represents a complete IPU processing flow. For example,

- A complete IPU processing flow (logical channel) might consist of reading a YUV buffer from memory, performing post-processing, and writing an RGB buffer to memory.
- A logical channel maps one to three IDMA channels and maps to either zero or one IC tasks.
- A logical channel can have one input, one output, and one secondary input IDMA channel.

The IPU API consists of a set of common functions for all channels. It aims to initialize channels, set up buffers, enable and disable channels, link channels for auto frame synchronization, and set up interrupts.

Typical logical channels include:

- CSI direct to memory
- CSI to viewfinder pre-processing to memory
- Memory to viewfinder pre-processing to memory
- Memory to viewfinder rotation to memory
- Previous field channel of memory to video deinterlacing and viewfinder pre-processing to memory
- Current field channel of memory to video deinterlacing and viewfinder pre-processing to memory
- Next field channel of memory to video deinterlacing and viewfinder pre-processing to memory
- CSI to encoder pre-processing to memory

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

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

- Synchronous frame buffer functions
- Panel interface initialization
- Set foreground positions
- Set local/global alpha and color key
- Set gamma
- CSI functions
- Sensor interface initialization
- Set sensor clock
- Set capture size

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

5.2 Hardware Operation

The detailed hardware operation of the IPU is described in the *Applications Processor Reference Manual*. The following figure shows the IPU hardware modules.

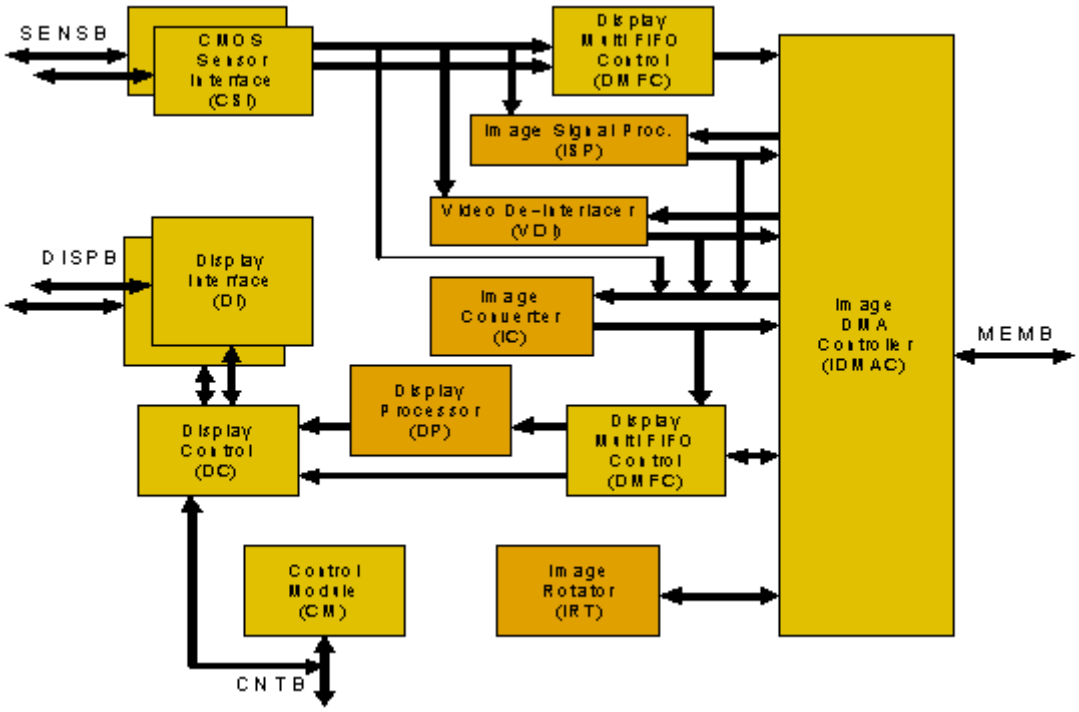


Figure 5-1. IPUv3EX/IPUv3H IPU Module Overview

5.3 Software Operation

The IPU driver is a self-contained driver module in the Linux kernel.

It consists of a custom kernel-level API for the following blocks:

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

The following figure shows the interaction between the different graphics/video drivers and the IPU.

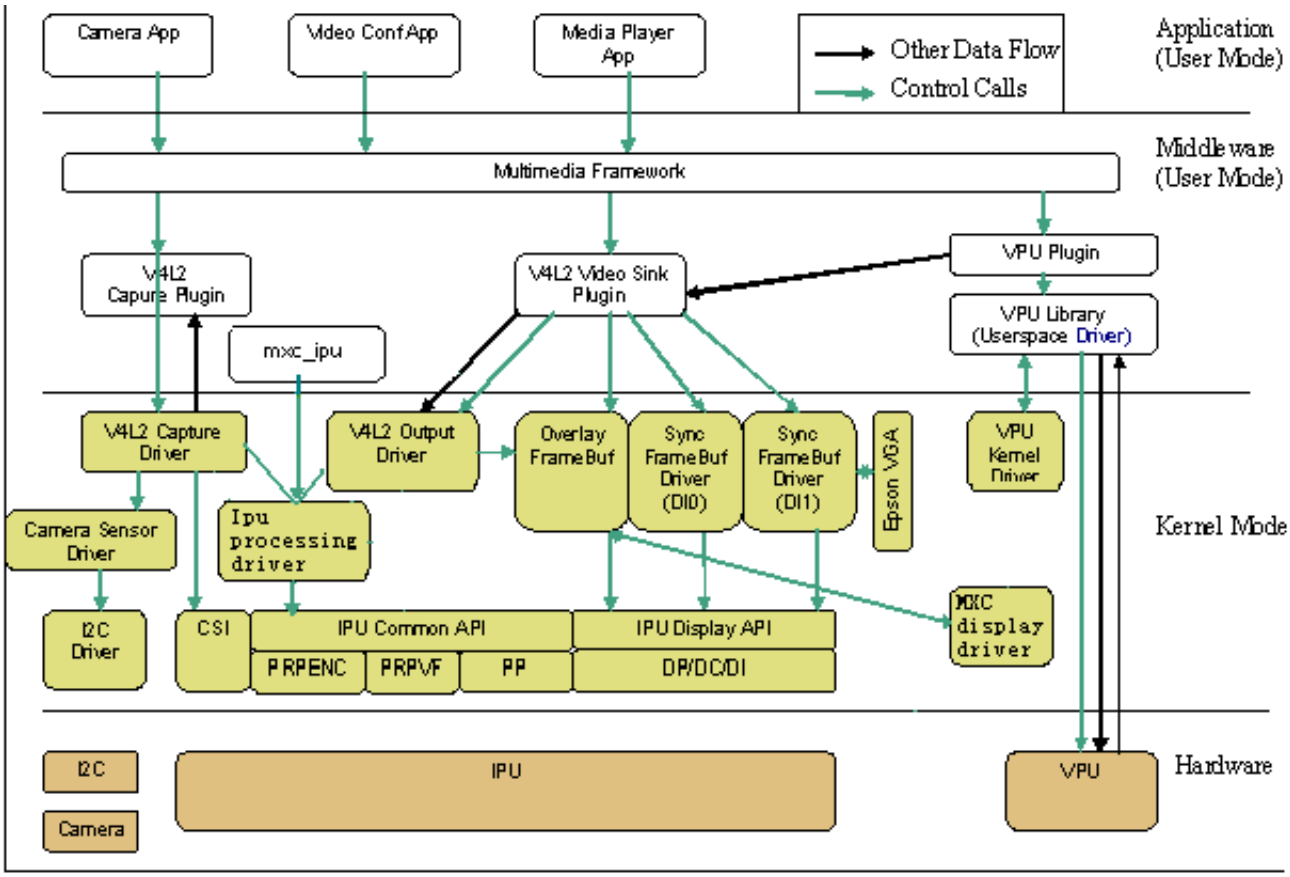


Figure 5-2. Graphics/Video Drivers Software Interaction for IPUv3

The IPU drivers are sub-divided as follows:

- Device drivers: include the frame buffer driver for the synchronous frame buffer, the frame buffer driver for the displays, V4L2 capture drivers for IPU pre-processing, the V4L2 output driver for IPU post-processing, and the IPU processing driver that provides a system interface to the user space or V4L2 drivers. The frame buffer device drivers are available in the <ltib_dir>/rpm/BUILD/linux/drivers/video/mxc directory of the Linux kernel. The V4L2 device drivers are available in the <ltib_dir>/rpm/BUILD/linux/drivers/media/video directory of the Linux kernel.
- The MXC display driver is a simple framework to manage interaction between the IPU and display device drivers (such as LCD, LVDS, HDMI, and MIPI).
- Low-level library routines: connect to the IPU hardware registers. They take input from the high-level device drivers and communicate with the IPU hardware. The low-level libraries are available in the directory of the Linux kernel.

5.3.1 Overview of IPU Frame Buffer Drivers

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

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

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

5.3.1.1 IPU Frame Buffer Hardware Operation

The frame buffer interacts with the IPU hardware driver module.

5.3.1.2 IPU Frame Buffer Software Operation

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

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

- Requesting general information about the hardware, such as name, organization of the screen memory (planes, packed pixels, and so on), and address and length of the screen memory.
- Requesting and changing variable information about the hardware, such as visible and virtual geometry, depth, color map format, timing. The driver suggests values to

meet the hardware capabilities (the hardware returns EINVAL if that is not possible) if this information is changed.
- Getting and setting parts of the color map. Communication is 16 bits-per-pixel (values for red, green, blue, transparency) to support all existing hardware. The driver does all the calculations required to apply the options to the hardware (round to fewer bits, possibly discard transparency value).

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

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

5.3.1.3 Synchronous Frame Buffer Driver

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

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

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

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

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

The following features are supported:

- Configurable screen resolution

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

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

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

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

5.3.2 IPU Backlight Driver

The IPU backlight driver implements IPU PWM backlight control for panels. It exports a system control file under /sys/class/backlight/pwm-backlight.0/brightness to user space. The default backlight intensity value is 128.

5.3.3 IPU Device Driver

IPU (processing) device driver provide image processing features, including resizing, rotation, CSC, combination, and deinterlacing based on IC/IRT modules in IPUv3.

The IPU device driver is task based. Users only need to prepare for task setting, queue task, and then the block waits for the task to finish. The driver now supports the blocking method only, and the non-block method will be added in the future. The task structures are as follows:

```
struct ipu_task {
        struct ipu_input input;
        struct ipu_output output;

        bool overlay_en;
        struct ipu_overlay overlay;

#define IPU_TASK_PRIORITY_NORMAL 0
#define IPU_TASK_PRIORITY_HIGH  1
        u8      priority;

#define IPU_TASK_ID_ANY 0
#define IPU_TASK_ID_VF  1
#define IPU_TASK_ID_PP  2
#define IPU_TASK_ID_MAX 3
        u8      task_id;
```

```
        int     timeout;
};

struct ipu_input {
        u32 width;
        u32 height;
        u32 format;
        struct ipu_crop crop;
        dma_addr_t paddr;

        struct ipu_deinterlace deinterlace;
        dma_addr_t paddr_n; /*valid when deinterlace enable*/
};

struct ipu_overlay {
        u32 width;
        u32 height;
        u32 format;
        struct ipu_crop crop;
        struct ipu_alpha alpha;
        struct ipu_colorkey colorkey;
        dma_addr_t
paddr;

};




struct ipu_output
{


        u32 width;
        u32 height;
        u32 format;
        u8 rotate;
        struct ipu_crop crop;
        dma_addr_t paddr;
};
```

To prepare for the task, users only need to enter the task.input, task.overlay(if need combine) and task.output parameters, and then queue task either by `int ipu_queue_task(struct ipu_task *task);` if from kernel level(v4l2 driver for example), or by IPU_QUEUE_TASK ioctl under /dev/mxc_ipu if from application level.

5.4 Source Code Structure

Table 5-1 lists the source files associated with the IPU, Sensor, V4L2, and Panel drivers. These files are available in the following directories:

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

Table 5-1. IPU Driver Files

File	Description
ipu_common.c	IPU common library functions
ipu_ic.c	IPU IC base driver
ipu_device.c	IPU driver device interface and fops functions
ipu_capture.c	IPU CSI capture base driver
ipu_disp.c	IPU display functions
ipu_calc_stripes_sizes.c	Multi-stripes method functions for ipu_device.c
mxc_ipuv3_fb.c	Driver for synchronous frame buffer
mxc_lcdif.c	Display Driver for CLAA-WVGA and SEIKO-WVGA LCD support
mxc_hdmi.c	Display Driver for HDMI interface
ldb.c	Driver for synchronous frame buffer for on chip LVDS
mxc_dispdrv.c	Display Driver framework for synchronous frame buffer
mxc_dvi.c	Display Driver for DVI interface
mxc_edid.c	Driver for EDID
vdoa.c	VDOA post-processing driver, used by ipu_device.c

Table 5-2 lists the global header files associated with the IPU and Panel drivers. These files are available in the following directories:

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

Table 5-2. IPU Global Header Files

File	Description
ipu_param_mem.h	Helper functions for IPU parameter memory access
ipu_prv.h	Header file for Pre-processing drivers
ipu_regs.h	IPU register definitions
vdoa.h	Header file for VDOA drivers
mxc_dispdrv.h	Header file for display driver
mxcfb.h	Header file for the synchronous framebuffer driver
ipu.h	Header file for ipu basic driver

5.4.1 Menu Configuration Options

The following Linux kernel configuration options are provided for the IPU module.

To get to these options, use the command ./ltib -c when located in the <ltib dir>. On the displayed screen, select Configure the kernel and exit. When the next screen appears, select the options to configure.

- CONFIG_MXC_IPU: includes support for the Image Processing Unit. In menu configuration, this option is available under:

  Device Drivers > MXC support drivers > Image Processing Unit Driver

  By default, this option is Y for all architectures.

  If ARCH_MX37 or ARCH_MX5 is true, CONFIG_MXC_IPU_V3 will be set. Otherwise, CONFIG_MXC_IPU_V1 will be set.

- CONFIG_MXC_CAMERA_OV5640_MIPI: option for both the OV 5640 mipi sensor driver and the use case driver. This option is dependent on the MXC_IPU option. In menu configuration, this option is available under:

  Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > OV 5640 Camera support using mipi

  Only one sensor should be installed at a time.

- CONFIG_MXC_CAMERA_OV5642: option for both the OV5642 sensor driver and the use case driver. This option is dependent on the MXC_IPU option. In menu configuration, this option is available under:

  Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > OmniVision ov5642 camera support

  Only one sensor should be installed at a time.

- CONFIG_MXC_CAMERA_OV5642: option for both the OV5642 sensor driver and the use case driver. This option is dependent on the MXC_IPU option. In menu configuration, this option is available under:

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

  Only one sensor should be installed at a time.

- CONFIG_MXC_IPU_PRP_VF_SDC: option for the IPU (here the > symbols illustrates data flow direction between HW blocks):

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

Use case driver for dumb sensor or

CSI > IC(PRP VF) > MEM

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

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

By default, this option is M for all.

- CONFIG_MXC_IPU_PRP_ENC: option for the IPU:

  Use case driver for dumb sensors

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

  or for smart sensors

  CSI > IC(PRP ENC) > MEM.

  In menu configuration, this option is available under:

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

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

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

  VIDEO_DEV && MXC_IPU && MXC_IPU_PRP_VF_SDC && MXC_IPU_PRP_ENC

  In menu configuration, this option is available under:

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

  By default, this option is M for all.

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

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

  By default, this option is Y for all.

- CONFIG_FB: includes frame buffer support in the Linux kernel. In menu configuration, this option is available under:

  Device Drivers > Graphics support > Support for frame buffer devices

  By default, this option is Y for all architectures.

- CONFIG_FB_MXC: option for the MXC Frame buffer driver. This option is dependent on the CONFIG_FB option. In menu configuration, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support

  By default, this option is Y for all architectures.

- CONFIG_FB_MXC_SYNC_PANEL: chooses the synchronous panel framebuffer. This option is dependent on the CONFIG_FB_MXC option. In menu configuration, this option is available under:

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

  By default this option is Y for all architectures.

- CONFIG_FB_MXC_LDB: selects the LVDS module on iMX53 chip. This option is dependent on CONFIG_FB_MXC_SYNC_PANEL and CONFIG_MXC_IPU_V3 option. In menu configuration, this option is available under:

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

- CONFIG_FB_MXC_SII9022: selects the SII9022 HDMI chip. This option is dependent on CONFIG_FB_MXC_SYNC_PANEL option. In menu configuration, this option is available under:

  Device Drivers > Graphics support > MXC Framebuffer support > Synchronous Panel Framebuffer > Si Image SII9022 DVI/HDMI Interface Chip

5.5 Unit Test

NOTE

In order to execute the tests properly, make sure that you select the util-linux package and load the following modules:

```
insmod ipu_prp_enc.ko
insmod ipu_bg_overlay_sdc.ko
insmod ipu_fg_overlay_sdc.ko
```

```
insmod ipu_csi_enc.ko
insmod ov5642_camera.ko
insmod mxc_v4l2_capture.ko
```

5.5.1 Framebuffer Tests

There is a test application named mxc_fb_test.c under the <ltib_dir>/rpm/BUILD/imx-test-"version"/test/mxc_fb_test directory.

Execute the fb test as follows:

./mxc_fb_test.out

The result should be Exiting PASS. The test includes fb0(background) and fb1(foreground) devices open, framebuffer parameters configure, global alpha blending, fb pan display test and gamma test.

Redirect an image directly to the framebuffer device as follows:

cat image.bin > /dev/fb0

5.5.2 Video4Linux API test

There are test applications named mxc_v4l2_test.c and mxc_v4l2_output.c under the <ltib_dir>/rpm/BUILD/imx-test-"version"/test/mxc_v4l2_test directory.

Before running the v4l2 capture test application, make sure that the /dev/v4l/video0 is created.

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

```
        # mxc_v4l2_capture.out -iw 640 -ih 480 -m 0 -r 0 -c 50 -fr 30 test.yuv

        Capture the camera and store the 50 frames of YUV420 (VGA size)to a file called
test.yuv and set the frame rate to 30 fps. Look at mxc_v4l2_capture.out -help to see
usage.
```

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

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

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

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

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

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

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

```
        # mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 120 -oh 120 -ot 40 -ol 40 -r
0 -t 50 -d 1 -fg -fr 30

        Direct preview the camera to foreground, and set frame rate to 30 fps.
```

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

```
        # mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 120 -oh 120 -ot 40 -ol 40 -r
4 -t 50 -d 1 -fg -fr 30

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

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

```
        # mxc_v4l2_output.out -iw 640 -ih 480 -ow 1024 -oh 768 -r 0 -fr 60 test.yuv

        Read the YUV420 stream file on test.yuv created by the mxc_v4l2_capture test as run
in test FSL-UT-V4L2-capture-0010. Apply color space conversion and resize, then
display on the framebuffer.
```

NOTE

The PRP channels require the stride line to be a multiple of 8. For example, with no rotation, the width needs to be 8 bit aligned; with 90 degree rotation, the height needs to be 8 bit aligned. Downsizing cannot exceed 8:1. For example, for a VGA sensor, the smallest downsize will be 80x60.

5.5.3 IPU Device Unit test

There is a test application named mxc_ipudev_test.c under the <ltib_dir>/rpm/BUILD/imx-test-"version"/test/mxc_ipudev_test directory.

Before running the ipu device test application, make sure that the /dev/mxc_ipu is created.

Run the test as follows:

```
./mxc_ipudev_test.out -C config_file raw_data_file

./mxc_ipudev_test.out -command_line_options raw_data_file
```

For file configuration instructions, refer to <ltib_dir>/rpm/BUILD/imx-test-"version"/test/ipudev_config_file.

Below is a simple test source code of IPU device overlay which useS alpha (global/local) blending to combine two layers:

```
static unsigned int fmt_to_bpp(unsigned int pixelformat)
{
        unsigned int bpp;

        switch (pixelformat) {
                case IPU_PIX_FMT_RGB565:
                /*interleaved 422*/
                case IPU_PIX_FMT_YUYV:
                case IPU_PIX_FMT_UYVY:
                /*non-interleaved 422*/
                case IPU_PIX_FMT_YUV422P:
                case IPU_PIX_FMT_YVU422P:
                        bpp = 16;
                        break;
                case IPU_PIX_FMT_BGR24:
                case IPU_PIX_FMT_RGB24:
                case IPU_PIX_FMT_YUV444:
                        bpp = 24;
                        break;
                case IPU_PIX_FMT_BGR32:
                case IPU_PIX_FMT_BGRA32:
                case IPU_PIX_FMT_RGB32:
                case IPU_PIX_FMT_RGBA32:
                case IPU_PIX_FMT_ABGR32:
                        bpp = 32;
                        break;
                /*non-interleaved 420*/
                case IPU_PIX_FMT_YUV420P:
                case IPU_PIX_FMT_YVU420P:
                case IPU_PIX_FMT_YUV420P2:
                case IPU_PIX_FMT_NV12:
                        bpp = 12;
                        break;
                default:
                        bpp = 8;
                        break;
        }
        return bpp;

}

static void dump_ipu_task(struct ipu_task *t)
{
        printf("====== ipu task ======\n");
        printf("input:\n");
        printf("\twidth: %d\n", t->input.width);
        printf("\theight: %d\n", t->input.height);
        printf("\tcrop.w = %d\n", t->input.crop.w);
        printf("\tcrop.h = %d\n", t->input.crop.h);
        printf("\tcrop.pos.x = %d\n", t->input.crop.pos.x);
        printf("\tcrop.pos.y = %d\n", t->input.crop.pos.y);
        printf("output:\n");
        printf("\twidth: %d\n", t->output.width);
```

```
        printf("\theight: %d\n", t->output.height);
        printf("\tcrop.w = %d\n", t->output.crop.w);
        printf("\tcrop.h = %d\n", t->output.crop.h);
        printf("\tcrop.pos.x = %d\n", t->output.crop.pos.x);
        printf("\tcrop.pos.y = %d\n", t->output.crop.pos.y);

       if (t->overlay_en) {
                printf("overlay:\n");
                printf("\twidth: %d\n", t->overlay.width);
                printf("\theight: %d\n", t->overlay.height);
                printf("\tcrop.w = %d\n", t->overlay.crop.w);
                printf("\tcrop.h = %d\n", t->overlay.crop.h);
                printf("\tcrop.pos.x = %d\n", t->overlay.crop.pos.x);
                printf("\tcrop.pos.y = %d\n", t->overlay.crop.pos.y);
        }

}

int main(int argc, char *argv[])
{
        int fd, fd_fb, isize, ovsize, alpsize, cnt = 50;
        int blank, ret;
        FILE * file_in = NULL;
        struct ipu_task task;
        struct fb_var_screeninfo fb_var;
        struct fb_fix_screeninfo fb_fix;
        void *inbuf, *ovbuf, *alpbuf, *vdibuf;

        fd = open("/dev/mxc_ipu", O_RDWR, 0);
        fd_fb = open("/dev/fb1", O_RDWR, 0);
        file_in = fopen(argv[argc-1], "rb");

        memset(&task, 0, sizeof(task));

        /* input setting */
        task.input.width = 320;
        task.input.height = 240;
        task.input.crop.pos.x = 0;
        task.input.crop.pos.y = 0;
        task.input.crop.w = 0;
        task.input.crop.h = 0;
        task.input.format = IPU_PIX_FMT_YUV420P;

        isize = task.input.paddr =
                task.input.width * task.input.height
                * fmt_to_bpp(task.input.format)/8;
        ioctl(fd, IPU_ALLOC, &task.input.paddr);
        inbuf = mmap(0, isize, PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, task.input.paddr);

       /*overlay setting */
        task.overlay_en = 1;
        task.overlay.width = 1024;
        task.overlay.height = 768;
        task.overlay.crop.pos.x = 0;
        task.overlay.crop.pos.y = 0;
        task.overlay.crop.w = 0;
        task.overlay.crop.h = 0;
        task.overlay.format = IPU_PIX_FMT_RGB24;
#ifdef GLOBAL_ALP
        task.overlay.alpha.mode = IPU_ALPHA_MODE_GLOBAL;
        task.overlay.alpha.gvalue = 255;
        task.overlay.colorkey.enable = 1;
        task.overlay.colorkey.value = 0x555555;
#else
        task.overlay.alpha.mode = IPU_ALPHA_MODE_LOCAL;
        alpsize = task.overlay.alpha.loc_alp_paddr =
                task.overlay.width * task.overlay.height;
        ioctl(fd, IPU_ALLOC, &task.overlay.alpha.loc_alp_paddr);
        alpbuf = mmap(0, alpsize, PROT_READ | PROT_WRITE,
```

```
                MAP_SHARED, fd, task.overlay.alpha.loc_alp_paddr);
        memset(alpbuf, 0x00, alpsize/4);
        memset(alpbuf+alpsize/4, 0x55, alpsize/4);
        memset(alpbuf+alpsize/2, 0x80, alpsize/4);
        memset(alpbuf+alpsize*3/4, 0xff, alpsize/4);
#endif

        ovsize = task.overlay.paddr =
                task.overlay.width * task.overlay.height
                * fmt_to_bpp(task.overlay.format)/8;
        ioctl(fd, IPU_ALLOC, &task.overlay.paddr);
        ovbuf = mmap(0, ovsize, PROT_READ | PROT_WRITE,
                MAP_SHARED, fd, task.overlay.paddr);
#ifdef GLOBAL_ALP
        memset(ovbuf, 0x55, ovsize/4);
        memset(ovbuf+ovsize/4, 0xff, ovsize/4);
        memset(ovbuf+ovsize/2, 0x55, ovsize/4);
        memset(ovbuf+ovsize*3/4, 0x00, ovsize/4);
#else
        memset(ovbuf, 0x55, ovsize);
#endif
#endif

        /* output setting*/
        task.output.width = 1024;
        task.output.height = 768;
        task.output.crop.pos.x = 0;
        task.output.crop.pos.y = 0;
        task.output.crop.w = 0;
        task.output.crop.h = 0;
        task.output.format = IPU_PIX_FMT_RGB565;
        task.output.rotate = IPU_ROTATE_NONE;

        ioctl(fd_fb, FBIOGET_VSCREENINFO, &fb_var);
        fb_var.xres = task.output.width;
        fb_var.xres_virtual = fb_var.xres;
        fb_var.yres = task.output.height;
        fb_var.yres_virtual = fb_var.yres * 3;
        fb_var.activate |= FB_ACTIVATE_FORCE;
        fb_var.nonstd = task.output.format;
        fb_var.bits_per_pixel = fmt_to_bpp(task.output.format);
        ioctl(fd_fb, FBIOPUT_VSCREENINFO, &fb_var);
        ioctl(fd_fb, FBIOGET_VSCREENINFO, &fb_var);
        ioctl(fd_fb, FBIOGET_FSCREENINFO, &fb_fix);
        task.output.paddr = fb_fix.smem_start;
        blank = FB_BLANK_UNBLANK;
        ioctl(fd_fb, FBIOBLANK, blank);

        task.priority = IPU_TASK_PRIORITY_NORMAL;
        task.task_id = IPU_TASK_ID_ANY;
        task.timeout = 1000;

again:
        ret = ioctl(fd, IPU_CHECK_TASK, &task);

        if (ret != IPU_CHECK_OK)  {
                if (ret > IPU_CHECK_ERR_MIN) {
                        if (ret == IPU_CHECK_ERR_SPLIT_INPUTW_OVER)  {
                                task.input.crop.w -= 8;
                                goto again;
                        }
                        if (ret == IPU_CHECK_ERR_SPLIT_INPUTH_OVER) {
                                task.input.crop.h -= 8;
                                goto again;
                        }
                        if (ret == IPU_CHECK_ERR_SPLIT_OUTPUTW_OVER) {
                                task.output.crop.w -= 8;
                                goto again;
                        }
                        if (ret == IPU_CHECK_ERR_SPLIT_OUTPUTH_OVER) {
```

```
                                task.output.crop.h -= 8;
                                goto again;
                        }
                        ret = -1;
                        return ret;
                }
        }


        dump_ipu_task(&task);

        while (--cnt > 0) {
                fread(inbuf, 1, isize, file_in);
                ioctl(fd, IPU_QUEUE_TASK, &task);
        }

        munmap(ovbuf, ovsize);

        ioctl(fd, IPU_FREE, task.input.paddr);
        ioctl(fd, IPU_FREE, task.overlay.paddr);

        close(fd);
        close(fd_fb);
        fclose(file_in);
}
```

NOTE

The overlay width and height must be the same as those of the output. For example, if the input is 240x320, and the output is 1024x768 which uses rotation of 90 degree, the overlay must be the same as the output, that is, 1024x768.

Chapter 6
MIPI DSI Driver

6.1 Introduction

The MIPI DSI driver for Linux is based on the IPU framebuffer driver.

This driver has two parts:

- MIPI DSI IP driver: low-level interface used to communicate with MIPI device controller on the display panel.
- MIPI DSI display panel driver: provides an interface to configure the display panel through MIPI DSI.

6.1.1 Overview of MIPI DSI IP Driver

The MIPI DSI IP driver is registered through the IPU framebuffer driver interface and it is not exposed to the user space.

The driver enables the platform-related regulators and clocks. It requests OS related system resources and registers framebuffer event notifier for blank/unblank operation. Additionally, the driver initializes MIPI D-PHY and configures the MIPI DSI IP according to the MIPI DSI display panel. The MIPI DSI driver supports the following features:

- Compatibility with MIPI Alliance Specification for DSI, Version1.01.00.
- Compatibility with MIPI Alliance Specification for D-PHY, Version 1.00.00.
- Supports up to two D-PHY data lanes.
- Bidirectional Communication and Escape Mode Support through Data Lane 0.
- Programmable display resolutions, from 160x120 (QQVGA) to 1024x768 (XVGA).
- Video Mode Pixel Formats, 16bpp (565RGB),18bpp (666RGB) packed, 18bpp (666RGB) loosely, 24bpp (888RGB).
- Supports the transmission of all generic commands.
- Supports ECC and checksum capabilities.

- Supports End-of-Transmission Packet(EoTp).
- Supports ultra low power mode.

6.1.2 Overview of MIPI DSI Display Panel Driver

The MIPI DSI display panel driver is used to configure the MIPI DSI display panel.

It uses the APIs provided by the MIPI DSI IP driver to read/write the display module registers. Usually, there is a MIPI DSI slave controller integrated on the display panel. After being powered on and reset, the MIPI DSI display panel needs to be configured through standard MIPI DCS command or MIPI DSI Generic command according to manufacturer's specification.

6.1.3 Hardware Operation

The MIPI DSI module provides a high-speed serial interface between a host processor and a display module.

It has higher performance, lower power, less EMI and fewer pins compared with legacy parallel bus. It is designed to be compatible with the standard MIPI DSI protocol. MIPI DSI is built on exisiting MIPI DPI-2, MIPI DBI-2 and MIPI DCS standards. It sends pixels or commands to the peripheral and reads back status or pixel information from the peripheral. MIPI DSI serializes all pixels data, commands and events, and contains two basic modes: command mode and video mode. It uses command mode to read/write register and memory to the display controller while reading display module status information. On the other hand, it uses video mode to transmit a real-time pixel streams from host to peripheral in high speed mode. It also generates an interrupt when an error occurs.

6.2 Software Operation

The MIPI DSI driver for Linux has two parts: MIPI DSI IP driver and MIPI DSI display panel driver.

6.2.1 MIPI DSI IP Driver Software Operation

The MIPI DSI IP driver has a private structure called mipi_dsi_info. The IPU instance to which the MIPI DSI IP is attached is described in the field int ipu_id while the DI instance inside IPU is described in the field int disp_id

During startup, the MIPI DSI IP driver is registered with the IPU framebuffer driver through the field struct mxc_dispdrv_entry when the driver is loaded. It also registers a framebuffer event notifier with framebuffer core to perform the display panel blank/unblank operation. The field struct fb_videomode *mode and struct mipi_lcd_config *lcd_config are received from the display panel callback. The MIPI DSI IP needs this infomation to configure the MIPI DSI hardware registers.

After initializing the MIPI DSI IP controller and the display module, the MIPI DSI IP gets the pixel streams from IPU through DPI-2 interface and serializes pixel data and video event through high speed data links for display. When there is an framebuffer blank/unblank event, the registered notifier will be called to enter or leave low power mode.

The MIPI DSI IP driver provides three APIs for MIPI DSI display panel driver to configure the display module.

6.2.2 MIPI DSI Display Panel Driver Software Operation

The MIPI DSI Display Panel driver enables a particular display panel through the MIPI DSI interface. The driver should provide struct fb_videomode configuration and struct mipi_lcd_config data: some MIPI DSI parameters for the display panel such as maximum D-PHY clock, numbers of data lanes and DPI-2 pixel format. Finally, the display driver needs to set up display panel initialize routine by calling the APIs provided by MIPI DSI IP drivers.

6.3 Driver Features

The MIPI DSI driver supports the following features:

- MIPI DSI communication protocol
- MIPI DSI command mode and video mode
- MIPI DCS command operation

NOTE

The MIPI DSI driver does not support the DBI-2 mode, because the DBI-2 and DPI-2 cannot be enabled at the same time on this controller.

6.3.1 Source Code Structure

Table below shows the MIPI DSI driver source files available in the directory:

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

Table 6-1. MIPI DSI Driver Files

File	Description
mipi_dsi.c	MIPI DSI IP driver source file
mipi_dsi.h	MIPI DSI IP driver header file
mxcfb_hx8369_wvga.c	MIPI DSI Display Panel driver source file

6.3.2 Menu Configuration Options

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

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

6.3.3 Programming Interface

The MIPI DSI Display Panel driver can use the API interface to read and write the registers of the display panel device connected to MIPI DSI link.

For more information, see <ltib_dir>/rpm/BUILD/linux/driver/video/mxc/mipi_dsi.h.

Chapter 7
LVDS Display Bridge(LDB) Driver

7.1 Introduction

This section describes the LVDS Display Bridge(LDB) driver which controls LDB module to connect with external display devices through the LVDS interface.

7.1.1 Hardware Operation

The purpose of the LDB is to support the flow of synchronous RGB data from IPU to external display devices through the LVDS interface.

This support covers all aspects of these activities:

1. Connectivity to relevant devices; displays with LVDS receivers.
2. Arranging data as required by the external display receiver and by LVDS display standards.
3. Synchronization and control capabilities.

For detailed information about LDB, see the LDB chapter of the *Multimedia Applications Processor Reference Manual*.

7.1.2 Software Operation

LDB driver is functional if the driver is built-in and if the user adds **ldb** option to boot-up command line.

When more options with **ldb=** prefixed are added, LDB can be configured when the device is probed, including the LVDS channel mapping mode and bit mapping mode.

When the LDB device is probed properly, the driver will configure LDB reference resistor mode and LDB regulator by using platform data information. The LDB driver probe function will also try to match video modes for external display devices to LVDS interface. The display signal polarities control bits of LDB are set according to the matched video modes. LVDS channel mapping mode and bit mapping mode of LDB are set according to the bootup LDB option chosen by the user, if available. Otherwise, an appropriate LDB setting is chosen by the driver if the video mode can be found in local video mode database. LDB is fully enabled in probe function if the driver identifies a display device with LVDS interface as the primary display device.

The driver takes the following steps to enable a LVDS channel:

1. Set the ldb_di_clk's parent clk and the parent clk's rate.
2. Set the ldb_di_clk's rate.
3. Enable both ldb_di_clk and its parent clk.
4. Set the LDB in a proper mode including display signals' polarities, LVDS channel mapping mode, bit mapping mode, and reference resistor mode.
5. Enable related LVDS channels.

LDB driver registers FB event handler to control LDB and related clocks when the FB is blanked or unblanked.

See `<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc/ldb.c` for more information.

7.1.3 Source Code Structure

The source code is available in the following location:

```
<ltib_dir>/rpm/BUILD/linux/drivers/video/mxc/ldb.c
```

7.1.4 Menu Configuration Options

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

To get to these options, use the `./ltib -c` command when located in the `<ltib dir>`. On the displayed screen, select **Configure the Kernel** and exit. When the next screen appears, select the following options as build-in status to enable this module:

1. `Device Drivers -> Graphics support -> MXC Framebufer support -> Synchronous Panel Framebuffer -> MXC LDB`

Chapter 8
Video for Linux Two (V4L2) Driver

8.1 Introduction

The Video for Linux Two (V4L2) drivers are plug-ins to the V4L2 framework that enable support for camera and preprocessing functions, as well as video and post-processing functions.

The V4L2 camera driver implements support for all camera related functions. The V4l2 capture device takes incoming video images, either from a camera or a stream, and manipulates them. The output device takes video and manipulates it, and then sends it to a display or similar device. The V4L2 Linux standard API specification is available at http://v4l2spec.bytesex.org/spec

The features supported by the V4L2 driver are as follows:

- Direct preview and output to SDC foreground overlay plane (with synchronized to LCD refresh)
- Direct preview to graphics frame buffer (without synchronized to LCD refresh)
- Color keying or alpha blending of frame buffer and overlay planes
- Streaming (queued) capture from IPU encoding channel
- Direct (raw Bayer) still capture (sensor dependent)
- Programmable pixel format, size, frame rate for preview and capture
- Programmable rotation and flipping using custom API
- RGB 16-bit, 24-bit, and 32-bit preview formats
- Raw Bayer (still only, sensor dependent), RGB 16, 24, and 32-bit, YUV 4:2:0 and 4:2:2 planar, YUV 4:2:2 interleaved, and JPEG formats for capture
- Control of sensor properties including exposure, white-balance, brightness, and contrast
- Plug-in of different sensor drivers
- Link post-processing resize and CSC, rotation, and display IPU channels
- Streaming (queued) input buffer
- Double buffering of overlay and intermediate (rotation) buffers

- Configurable 3+ buffering of input buffers
- Programmable input and output pixel format and size
- Programmable scaling and frame rate
- RGB 16, 24, and 32-bit, YUV 4:2:0 and 4:2:2 planar, and YUV 4:2:2 interleaved input formats
- TV output

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

8.2 V4L2 Capture Device

The V4L2 capture device includes two interfaces:

- Capture interface-uses IPU pre-processing ENC channels to record the YCrCb video stream
- Overlay interface-uses the IPU device driver to display the preview video to the SDC foreground and background panel.

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

The V4L2 capture device driver is in the mxc_v4l2_capture.c file. The low level overlay driver is in the ipu_fg_overlay_sdc.c, ipu_bg_overlay_sdc.c

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

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

8.2.1 V4L2 Capture IOCTLs

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

- VIDIOC_QUERYCAP
- VIDIOC_G_FMT

- VIDIOC_S_FMT
- VIDIOC_REQBUFS
- VIDIOC_QUERYBUF
- VIDIOC_QBUF
- VIDIOC_DQBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF
- VIDIOC_OVERLAY
- VIDIOC_G_FBUF
- VIDIOC_S_FBUF
- VIDIOC_G_CTRL
- VIDIOC_S_CTRL
- VIDIOC_CROPCAP
- VIDIOC_G_CROP
- VIDIOC_S_CROP
- VIDIOC_S_PARM
- VIDIOC_G_PARM
- VIDIOC_ENUMSTD
- VIDIOC_G_STD
- VIDIOC_S_STD
- VIDIOC_ENUMOUTPUT
- VIDIOC_G_OUTPUT
- VIDIOC_S_OUTPUT

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

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

The following figure shows a block diagram of V4L2 Capture API interaction.

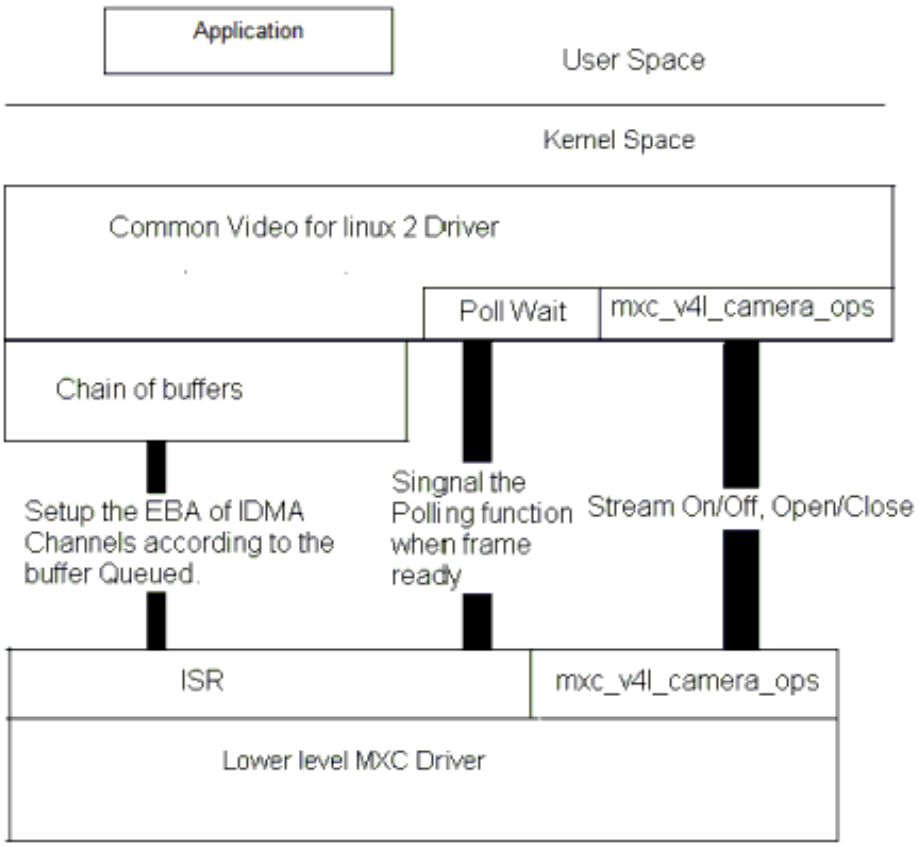


Figure 8-1. Video4Linux2 Capture API Interaction

8.2.2 Using the V4L2 Capture APIs

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

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

8. Stores the buffer to a YCrCb file.
9. Replaces the buffer in the queue of the V4L2 driver by executing VIDIOC_QBUF again.

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

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

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

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

8.3 V4L2 Output Device

The V4L2 output driver uses the IPU post-processing functions for video output.

The driver implements the standard V4L2 API for output devices. V4L2 output device support can be selected during kernel configuration. The driver is available at <ltib_dir>/rpm/BUILD/linux/drivers/media/video/mxc/output/mxc_vout.c.

8.3.1 V4L2 Output IOCTLs

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

- VIDIOC_QUERYCAP
- VIDIOC_REQBUFS
- VIDIOC_G_FMT
- VIDIOC_S_FMT
- VIDIOC_QUERYBUF
- VIDIOC_QBUF
- VIDIOC_DQBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF
- VIDIOC_G_CTRL
- VIDIOC_S_CTRL
- VIDIOC_CROPCAP

- VIDIOC_G_CROP
- VIDIOC_S_CROP
- VIDIOC_ENUM_FMT

The V4L2 control code has been extended to provide support for de-interlace motion. For this purpose, the ID is V4L2_CID_MXC_MOTION. Supported values include the following:

- 0-Medium motion
- 1-Low motion
- 2-High motion

8.3.2 Using the V4L2 Output APIs

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

1. Sets the input pixel format and size by using IOCTL VIDIOC_S_FMT.
2. Sets the control information by using IOCTL VIDIOC_S_CTRL, for rotation and de-interlace motion (if need).
3. Sets the output information by using IOCTL VIDIOC_S_CROP.
4. Requests a buffer by using IOCTL VIDIOC_REQBUFS. The common V4L2 driver creates a chain of buffers (not allocated yet).
5. Memory maps the buffer to its user space.
6. Executes IOCTL VIDIOC_QUERYBUF to query buffers.
7. Passes the data that requires post-processing to the buffer.
8. Queues the buffer by using the IOCTL command VIDIOC_QBUF.
9. Executes the IOCTL VIDIOC_DQBUF to dequeue buffers.
10. Starts the stream by executing IOCTL VIDIOC_STREAMON.
11. Stops the stream by excuting IOCTL VIDIOC_STREAMOFF.

8.4 Source Code Structure

The following table lists the source and header files associated with the V4L2 drivers.

These files are available in the following directory:

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

Table 8-1. V2L2 Driver Files

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

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

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

8.4.1 Menu Configuration Options

The Linux kernel configuration options are provided in the chapter on the IPU module.

See Menu Configuration Options.

8.4.2 V4L2 Programming Interface

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

The API Specification is available at LINUX MEDIA INFRASTRUCTURE API.

Chapter 9
Graphics Processing Unit (GPU)

9.1 Introduction

The Graphics Processing Unit (GPU) is a graphics accelerator targeting embedded 2D/3D graphics applications.

The 3D graphics processing unit (GPU3D) is based on the Vivante GC2000 core, which is an embedded engine that accelerates user level graphics Application Programming Interface (APIs) such as OpenGL ES 1.1, OpenGL ES 2.0 and OpenCL 1.1EP. The 2D graphics processing unit (GPU2D) is based on the Vivante GC320 core, which is an embedded 2D graphics accelerator targeting graphical user interfaces (GUI) rendering boost. The VG graphics processing unit (GPUVG) is based on the Vivante GC355 core, which is an embedded vector graphic accelerator for supporting the OpenVG 1.1 graphics API and feature set. The GPU driver kernel module source is in kernel source tree, but the libs are delivered as binary only.

9.1.1 Driver Features

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

- EGL (EGL is an interface between Khronos rendering APIs such as OpenGL ES or OpenVG and the underlying native platform window system) 1.4 API defined by Khronos Group.
- OpenGL ES (OpenGL® ES is a royalty-free, cross-platform API for full-function 2D and 3D graphics on embedded systems) 1.1 API defined by Khronos Group.
- OpenGL ES 2.0 API defined by Khronos Group.
- OpenVG (OpenVG is a royalty-free, cross-platform API that provides a low-level hardware acceleration interface for vector graphics libraries such as Flash and SVG) 1.1 API defined by Khronos Group.

- OpenCL (OpenCL is the first open, royalty-free standard for cross-platform, parallel programming of modern processors.) 1.1 EP API defined by Khronos Group.
- OpenGL 2.1 API defined by Khronos Group.
- Automatic 3D core slowing down, when hot notification from thermal driver is active, 3D core will run at 1/64 clock.

9.1.1.1 Hardware Operation

For detailed hardware operation and programming information, see the GPU chapter in the *i.MX 6Dual/6QuadApplications Processor Reference Manual*.

9.1.1.2 Software Operation

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

- OpenGL ES 1.1 and 2.0 API
- EGL 1.4 API
- OpenVG 1.1 API
- OpenCL 1.1 EP API

9.1.1.3 Source Code Structure

Table below lists GPU driver kernel module source structure:

<ltib_dir>/rpm/BUILD/linux/drivers/mxc/gpu-viv

Table 9-1. GPU Driver Files

File	Description
Kconfig Kbuild config	kernel configure file and makefile
arch/XAQ2/hal/kernel	hardware specific driver code for GC2000 and GC320
arch/GC350/hal/kernel	hardware specific driver code for GC350
hal/kernel	Kernel mode HAL driver
hal/os	os layer HAL driver

9.1.1.4 Library Structure

Table below lists GPU driver user mode library structure:

<ROOTFS>/usr/lib

Table 9-2. GPU Library Files

File	Description
libCLC.so	OpenCL frontend compiler library
libEGL.so*	EGL1.4 library
libGAL.so*	GAL user mode driver
libGLES_CL.so	OpenGL ES 1.1 common lite library (without EGL API, no float point support API)
libGL.so.1.2	OpenGL 2.1 common library
libGLES_CM.so	OpenGL ES 1.1 common library (without EGL API, include float point support API)
libGLESv1_CL.so	OpenGL ES 1.1 common lite library (with EGL API, no float point support API)
libGLESv1_CM.so	OpenGL ES 1.1 common library (with EGL API, include float point support API)
libGLESv2.so	OpenGL ES 2.0 library
libGLSLC.so	OpenGL ES shader language compiler library
libOpenCL.so	OpenCL 1.1 EP library
libOpenVG.so*	OpenVG 1.1 library
libVDK.so	VDK wrapper library.
libVIVANTE.so*	Vivante user mode driver.
directfb-1.4-0/gfxdrivers/libdirectfb_gal.so	DirectFB 2D acceleration library.
dri/vivante_dri.so	DRI library for OpenGL2.1.

* These libraries are actually symbolic links to the actual library file in the folder.

By default, these symbolic links are installed to point to the frame buffer version of the libraries as such:

```
libGAL.so -> libGAL-fb.so
libEGL.so -> libEGL-fb.so
libVIVANTE.so -> libVIVANTE-fb.so
libOpenVG.so -> libOpenVG_3D.so
```

On X11 systems, the symbolic links to these libraries need to be redirected. This can be done using the following sequence of commands:

```
> cd <ROOTFS>/usr/lib
> sudo ln -s libGAL-x11.so libGAL.so
> sudo ln -s libEGL-x11.so libEGL.so
> sudo ln -s libEGL-x11.so libEGL.so.1
> sudo ln -s libVIVANTE-x11.so libVIVANTE.so
```

On directFB backend, the symbolic links to these libraries need to be redirected. This can be done using the following sequence of commands:

```
> cd <ROOTFS>/usr/lib
> sudo ln -s libGAL-dfb.so libGAL.so
> sudo ln -s libEGL-dfb.so libEGL.so
> sudo ln -s libEGL-dfb.so libEGL.so.1
> sudo ln -s libVIVANTE-dfb.so libVIVANTE.so
```

For libOpenVG.so, there are two libraries for OpenVG feature. libOpenVG_3D.so is gc2000 based OpenVG library. libOpenVG_355.so is gc355 based OpenVG library. If gc355 based OpenVG library want to be used, this can be done by using the following sequence of commands:

```
> cd <ROOTFS>/usr/lib
> sudo ln -s libOpenVG_355.so libOpenVG.so
```

9.1.1.5 API References

Refer to the following web sites for detailed specifications:

- OpenGL ES 1.1 and 2.0 API: http://www.khronos.org/opengles/
- OpenCL 1.1 EP http://www.khronos.org/opencl/
- EGL 1.4 API: http://www.khronos.org/egl/
- OpenVG 1.1 API: http://www.khronos.org/openvg/

9.1.1.6 Menu Configuration Options

The following Linux kernel configurations are provided for GPU driver:

CONFIG_MXC_GPU_VIV is a configuration option for GPU driver. In menucon figuration this option is available under Device Drivers > MXC support drivers > MXC Vivante GPU support > MXC Vivante GPU support.

To get to the GPU library package in LTIB, use the command ./ltib -c when located in the <ltib dir>. On the displayed screen, select **Configure the kernel**, select Device Drivers > MXC support drivers > MXC Vivante GPU support > MXC Vivante GPU support, and then exit. When the next screen appears, select the following options to enable the GPU driver:

- Package list > gpu-viv-bin-mx6q
- This package provides proprietary binary libraries, and test code built from the GPU for framebuffer

Chapter 10
Direct FB

10.1 Introduction

DirectFB is a thin library that provides hardware graphics acceleration, input device handling and abstraction, integrated windowing system with support for translucent windows and multiple display layers, not only on top of the Linux Framebuffer Device. It is a complete hardware abstraction layer with software fallbacks for every graphics operation that is not supported by the underlying hardware. DirectFB adds graphical power to embedded systems and sets a new standard for graphics under Linux.

10.1.1 Hardware Operation

DirectFB acceleration uses the Vivante GPU.

The process is described in the Driver Features. Acceleration is also dependent on the frame buffer memory.

10.2 Software Operation

The DirectFB version which is currently supported is DirectFB-1.4.0.

Subsequent versions have not been tested and are not officially supported.

Since DirectFB is a thin Graphics library, it is lightweight and has a small footprint optimized for embedded devices. It is not a client/server model like X11.

It provides a hardware abstraction layer for hardware graphics acceleration: Anything that is not supported by hardware and still supported by software, but uses hardware where possible.

10.2.1 DirectFB Acceleration Architecture

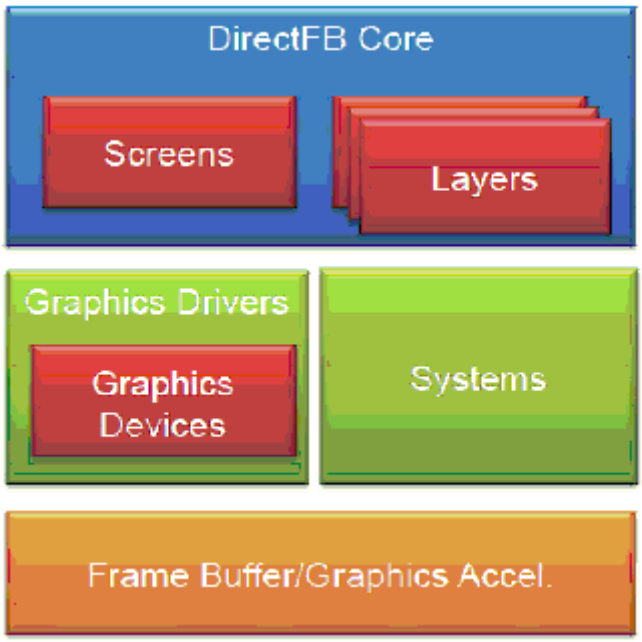


Figure 10-1. DirectFB Acceleration Architecture

	Header	Module Declaration/Registration	Required Functions
Systems	src/core/system.h src/core/core_system.h	DFB_CORE_SYSTEM(<name>)	See CoreSystemFuncs in core_system.h and system.h
Graphics Drivers	src/core/graphics_driver.h	DFB_GRAPHICS_DRIVER(<name>)	See GraphicsDriverFuncs in graphics_driver.h
Graphics Devices	src/core/gfxcard.h	via driver_init_driver() in GraphicsDriverFuncs	See GraphicsDeviceFuncs in gfxcard.h
Screens	src/core/screens.h	dfb_screens_register()	See ScreenFuncs in screens.h
Layers	src/core/layers.h	dfb_layers_register()	See DisplayLayerFuncs in layers.h

Figure 10-2. DirectFB Acceleration Architecture Details

Systems provides frame buffer and hardware management to access to the resources.

10.2.2 i.MX DirectFB Driver Details

The following are the types of operations that are accelerated for DirectFB:

- Rectangle filling/drawing
- Triangle filling/drawing
- Line drawing
- Flat shaded triangles
- Simple blitting
- Stretched blitting
- Textured triangles (perspective correct)
- Blending with an alphachannel (per pixel alpha)
- Blending with an alpha factor (alpha modulation)
- Nine source and destination blend functions
- Porter/Duff rules are supported
- Premultiplied alpha supported
- Colorized blitting (color modulation)
- Source color keying
- Destination color keying

Management

DirectFB has its own resource management for video memory. Resources like display layers or input devices can be locked for exclusive access, for example, for fullscreen games. DirectFB provides abstraction for the different graphics targets such as display layers, windows and any general purpose surfaces. The programming effort for switching from windowed to fullscreen and back is minimized to set the desired cooperative level.

DirectFB Modules

The API and structure of DirectFB is designed to provide an easy way of implementing the following parts:

- Graphics acceleration
- Input devices (currently keyboard, serial and PS/2 mice, joysticks)
- Image Provider (currently PNG, GIF and JPEG)
- Video Provider (currently Video4Linux, AVI (using avifile), MPEG1/2 (using libmpeg3))
- Font Provider (currently DirectFB bitmap font, TrueType via FreeType 2)

10.2.3 The gal_config File for i.MX DirectFB Driver

This is the configuration file for Vivante GFX plug-in driver. You can use this file to control which primitive is accelerated with specific features.

For example, if you want to accelerate blit with alpha blending and rotate180 features, add the following line to the file.

blit=alphachannel,coloralpha,rotate180

Then blit with other features (including xor and src_colorkey) are not accelerated by HW. Even blit without any features is not accelerated.

"none" in the feature list means the rendering primitive without any features.

Following is the full matrix of the primitives and features:

drawline=none,xor,blend

drawrectangle=none,xor,blend

fillrectangle=none,xor,blend

filltriangle=none,xor,blend

blit=none,xor,alphachannel,coloralpha,src_colorkey,rotate180

stretchblit=none,xor,alphachannel,coloralpha,src_colorkey,rotate180

To use the configuration file, set environment variable GAL_CONFIG_FILE pointing to this file. For example, for a bash user,

export GAL_CONFIG_FILE=/home/gfx/gal_config

If you don't set the environment variable, a default configuration matrix will be used. The default configuration matrix is as follows:

fillrectangle=none,xor,blend

filltriangle=none,xor,blend

filltriangle=none,xor,blend

blit=none,xor,alphachannel,coloralpha,src_colorkey,rotate180

stretchblit=none,xor,alphachannel,coloralpha,src_colorkey,rotate180

Configuration file has higher priority.

10.3 DirectFB EGL

EGL in DirectFB can be used for OpenGL and OpenVG applications.

To enable DirectFB EGL, you need to increase the size of FBDEV, which can be achieved by increasing the size in memory of /dev/fb0. This can be done by using fbset.

The typical size to of the fb is the same as the one used for triple buffer:

fbset -fb /dev/fb0 -g Xsize Ysize Xsize 3*Ysize BPP

For the hanstar lvds panel:

fbset -fb /dev/fb0 -g 1024 768 2034 32

For more information on fbset, check the fbset man pages.

10.4 Setting Up DirectFB Acceleration

Perform the following actions to set up DirectFB Acceleration:

1. Install ltib and select "min profile" or "gnome Mobile profile" and build with the default settings.
2. Cross compile DirectFB and their examples. You can do it by using the toolchain and follow the instructions in the DirectFB-1.4.0 package or have LTIB to do it all (recommended). The next steps use LTIB.
3. Because ltib checks the md5sum of the tar.gz, you need to to generate .md5 file for DirectFB-examples-1.2.0 in the pkgs directory:
   - `md5sum DirectFB-examples.1.2.0.tgz > DirectFB-examples-1.2.0.tgz.md5`

   NOTE

   You may also need to verify that the .spec file has the correct version:/ltib/dist/lfs-5.1/DirectFB/DirectFB-examples.spec

4. Run ltib with command
   - `./ltib -m config`
5. Make sure that “Configure the kernel” is selected
6. When configuring ltib, go to Package List and select
   - `gpu-viv-bin-mx6q`
     `DirectFB`
     `DirectFB-examples`
     `zlib`
7. Exit and save your configuration.
8. Run ltib
   - `./ltib`

9. Now you should be in the kernel configuration. Go to Device Drivers-> MXC Support Drivers->MXC Vivante GPU Support and include
   - `./MXC Vivante GPU Support`
10. Exit, save your configuration and wait for the build.

NOTE

Make sure that you do “insmod /lib/modules/kernel-version/kernel/drivers/mxc/gpu-viv/galcore.ko” before trying to run DirectFB applications.

To run the DFB examples, run “/usr/bin/df_dok”. It will perform a series of benchmarks and show the results, but they require settings such as jpgs, pngs, and fonts, which are in the DirectFB-examples tar file. Refer to the DirectFB-examples-1.2.0.tar.gz README for more details.

Chapter 11
HDMI Driver

11.1 Introduction

The High Definition Multimedia Interface (HDMI) driver supports the on-chip DesignWare HDMI hardware module, which provides the capability to transfer uncompressed video, audio, and data by using a single cable.

The HDMI driver is divided into four sub-components: A video display device driver that integrates with the Linux Frame Buffer API, an audio driver that integrates with the ALSA/SoC sub-system, a CEC driver, and a multi-function device (MFD) driver which manages the shared software and hardware resources of the HDMI driver.

The HDMI driver supports the following features:

- Integration with the MXC Display Device framework, used for managing display device connections with the IPU(s)
- HDMI video output up to 1080p60 resolution
- Support for reading EDID information from an HDMI sink device
- Hotplug detection
- Support for CEC
- Automated clock management to minimize power consumption
- Support for system suspend/resume
- HDMI audio playback (2, 4, 6, or 8 channels, 16 bit, for sample rates 32 KHz to 192 KHz)
- IEC audio header information exposed through ALSA by using ‘iecset’ utility

11.1.1 Hardware Operation

The HDMI module receives video data from the Image Processing Unit (IPU), audio data from the external memory interface, and control data from the CPU, as shown in the following figure.

Output data is transmitted through three Transition-Minimized Differential Signaling (TMDS) channels to an HDMI sink device external to the SoC. Additionally, the HDMI carries a VESA Data Display Channel (DDC). The DDC is an I2C interface which allows the HDMI source to query the HDMI sink for Extended Display Identification Data (EDID). A CEC channel provides optional high-level control functions between the source and sink device.

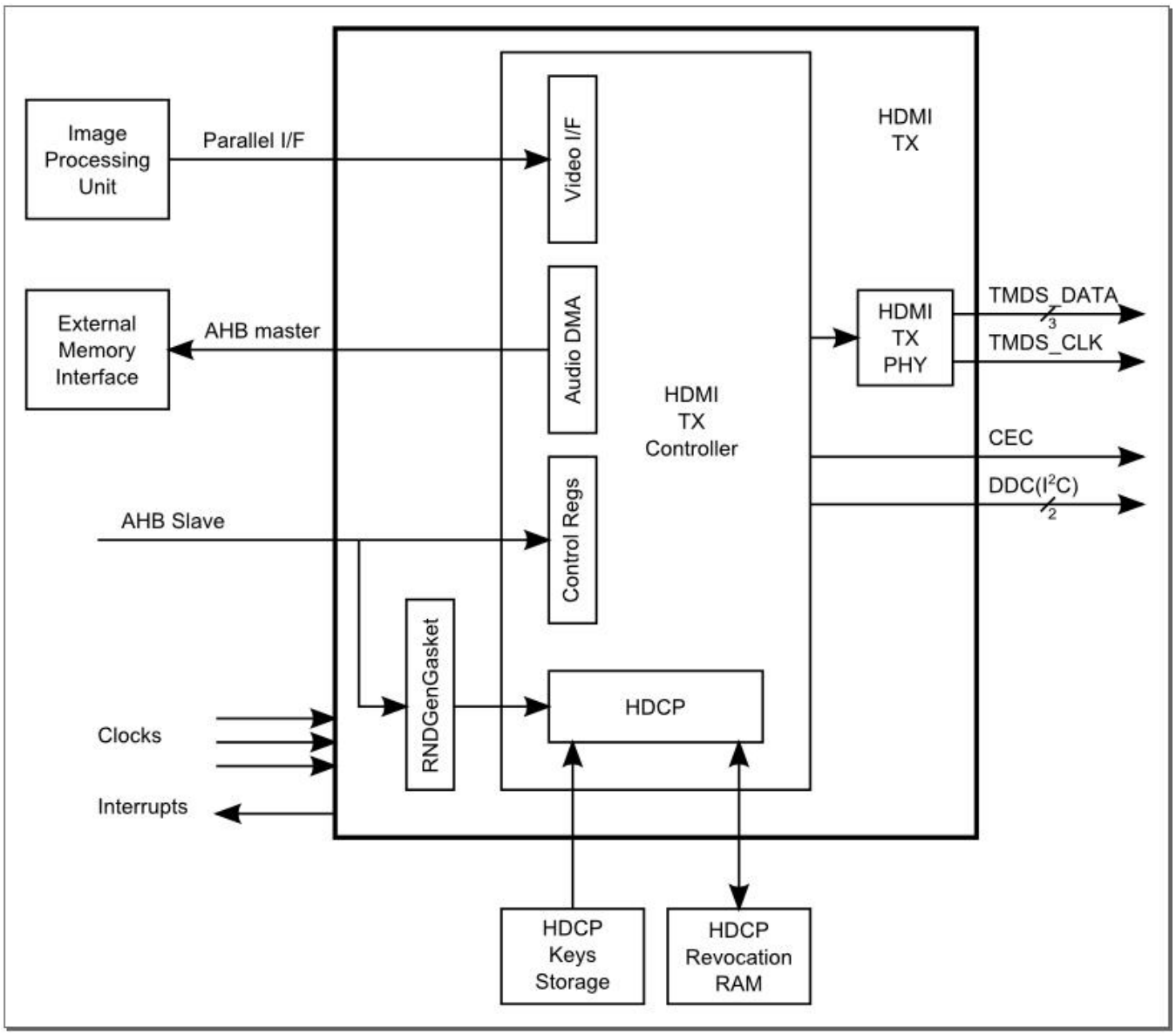


Figure 11-1. HDMI HW Integration

For additional details of the hardware operation of the HDMI module, see the HDMI section of the i.MX 6Dual/6Quad reference manual.

The video input to the HDMI is configurable and may come from either of the two IPU modules in the i.MX 6Dual/6Quad, and from either of the two Display Interface (DI) ports of the IPU, DI0 or DI1. This configuration is controlled through the IOMUX module by using the HDMI_MUX_CTRL register field. The following figure shows an illustration of this interconnection.

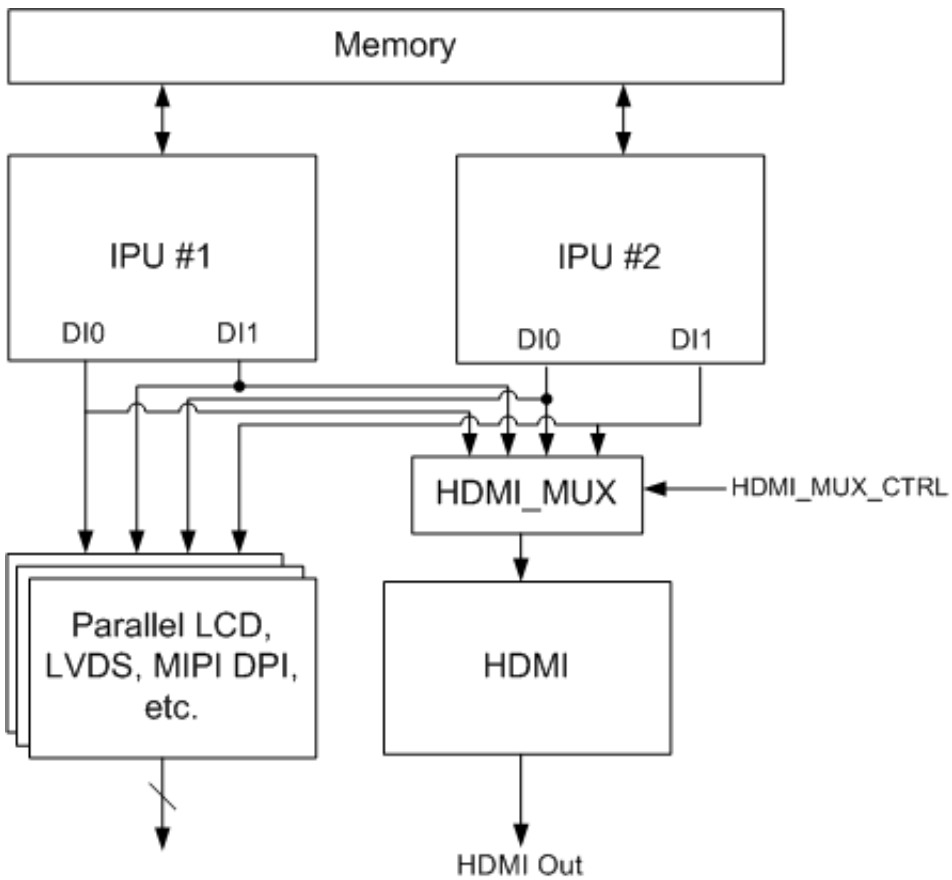


Figure 11-2. IPU-HDMI Hardware Interconnection

11.2 Software Operation

The HDMI driver is divided into sub-components based on its two primary purposes: providing video and audio to an HDMI sink device.

The video display driver component and audio driver component require an additional core driver component to manage common HDMI resources, including the HDMI registers, clocks, and IRQ.

11.2.1 Core

The HDMI core driver manages resources that must be shared between the HDMI audio and video drivers. The HDMI audio and video drivers depend on the HDMI core driver, and the HDMI core driver should always be loaded and initialized before audio and video. The core driver serves the following functions:

- Mapping the HDMI register region and providing APIs for reading and writing to HDMI registers.
- Performing one-time initialization of key HDMI registers.
- Initializing the HDMI IRQ and providing shared APIs for enabling and disabling the IRQ.
- Providing a means for sharing information between the audio and video drivers (such as the HDMI pixel clock).
- Providing a means for synchronization between HDMI video and HDMI audio while blank/unbalnk, plug in/plug out events occur. HDMI audio cannot start work while the HDMI cable is in the state of plug out or HDMI is in state of blank. Every time HDMI audio starts a playback, HDMI audio driver should register its PCM into core driver and unregister PCM when the playback is finished. Once HDMI video blank or cable is plugged out, the core driver would pause HDMI audio DMA controller if its PCM is registered. When HDMI is unblanked or the cable is plugged in, the core driver would firstly check if the cable is in the state of plug in, the video state is unblank and the PCM is registered. If items listed above are all yes, the core driver would restart HDMI audio DMA.

11.2.2 Video

The following diagram illustrates both the interconnection between the various HDMI sub-drivers and the interconnection between the HDMI video driver and the Linux Frame Buffer subsystem.

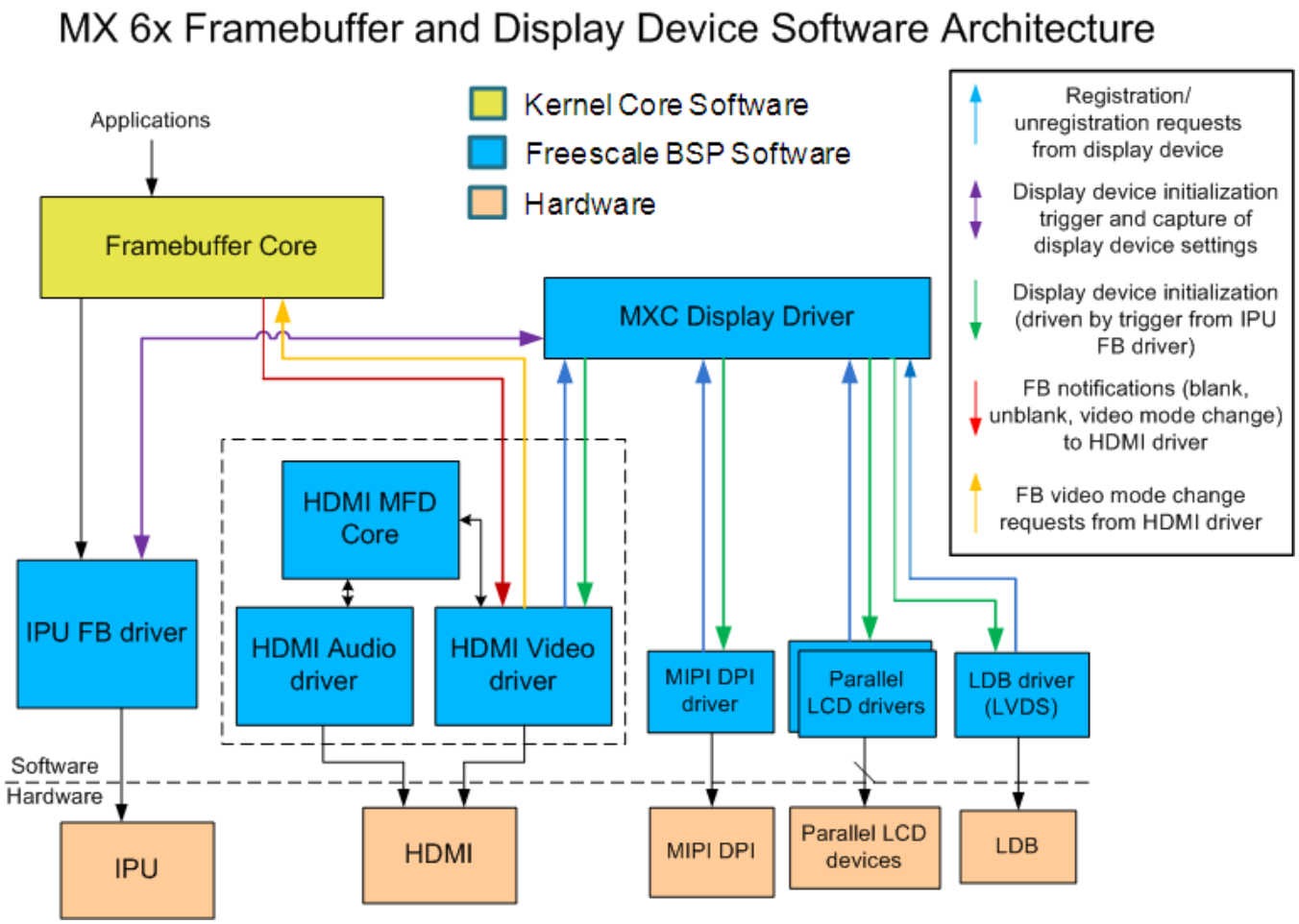


Figure 11-3. HDMI Video SW Architecture

The i.MX 6Dual/6Quad supports many different types of display output devices (such as LVDS, LCD, HDMI, and MIPI displays) connected to and driven by the IPU modules. The MXC Display Driver API provides a system for registering display devices and configuring how they should be connected to each of the IPU DIs. The HDMI driver registers itself as a display device by using this API to receive the correct video input from the IPU.

11.2.3 Display Device Registration and Initialization

The following sequence of software activities occurs in the OS boot flow to connect the HDMI display device to the IPU FB driver through the MXC Display Driver system:

1. During the HDMI video driver initialization, `mxc_dispdrv_register()` is called to register the HDMI module as a display device and to set the `mxc_hdmi_disp_init()` function as the display device init callback.

2. When the IPU FB driver is initialized, `mxc_dispdrv_init()` is called. This results in an initialization call to all registered display devices.
3. The `mxc_hdmi_disp_init()` callback is executed. The HDMI driver receives a structure from the IPU FB driver containing frame buffer information (fbi). The HDMI driver also provides return information about which IPU and DI to select and the preferred output format for video data from the IPU. The HDMI driver registers itself to receive notifications of FB driver events. Finally, the HDMI driver can complete its initialization by configuring the HDMI to receive a hotplug interrupt.

NOTE

All display device drivers must be initialized before the IPU FB driver for all display devices to be registered as MXC Display Driver devices before the IPU FB driver can initialize them.

11.2.4 Hotplug Handling and Video Mode Changes

Once the connection between the IPU and the HDMI has been established through the MXC Display Driver interface, the HDMI video driver waits for a hotplug interrupt, indicating that a valid HDMI sink device is connected and ready to receive HDMI video data. Subsequent communications between the HDMI and IPU FB are conducted through the Linux Frame Buffer APIs. The following list demonstrates the software flow to recognize an HDMI sink device and configure the IPU FB driver to drive video output to it:

1. The HDMI video driver receives a hotplug interrupt and reads the EDID from the HDMI sink device, constructing a list of video modes from the retrieved EDID information. Using either the video mode string from the Linux kernel command line (for the initial connection) or the most recent video mode (for a later HDMI cable connection), the HDMI driver selects a video mode from the mode list that is the closest match.
2. The HDMI video driver calls `fb_set_var()` to change the video mode in the IPU FB driver. The IPU FB driver completes its reconfiguration for the new mode.
3. As a result of calling fb_set_var(), an FB notification is sent back to the HDMI driver indicating that an FB_EVENT_MODE_CHANGE has occurred. The HDMI driver configures the HDMI hardware for the new video mode..
4. The HDMI module is enabled to generate output to the HDMI sink device.

11.2.5 Audio

The HDMI Tx audio driver uses the ALSA SoC framework, so it is broken into several files, as is listed in Table 11-4. Most of the code is in the platform DMA driver (sound/soc/imx/imx-hdmi-dma.c) and the codec driver (sound/soc/codecs/mxc_hdmi.c). The machine driver (sound/soc/imx/imx-hdmi.c) exists to allocate the SoC audio device and link all the SoC components together. The DAI driver (sound/soc/imx/imx-hdmi-dai.c) mostly exists because SoC wants there to be a DAI driver; it gets the platform data, but doesn't do anything else.

The HDMI codec driver does most of the initialization of the HDMI audio sampler. The HDMI Tx block only implements the AHB DMA audio and not the other audio interfaces (SSI, S/PDIF, etc). The other main function of the HDMI codec driver is to set up a struct of the IEC header information which needs to go into the audio stream. This struct is hooked into the ALSA layer, so the IEC settings will be accessible in userspace using the 'iecset' utility.

The platform DMA driver handles the HDMI Tx block's DMA engine. Note that HDMI audio uses the HDMI block's DMA as well as SDMA. SDMA is used to help implement the multi-buffer mechanism. The HDMI Tx block does not automatically merge the IEC audio header information into the audio stream, so the platform DMA driver does this in its hdmi_dma_copy() (for no memory map use) or hdmi_dma_mmap_copy() (for memory map mode use) function before the DMA sends the buffers out. Due to IEC audio header adding operation,the user space application may not be able to get enough CPU periods to feed data into HDMI audio driver in time, especially when system loading is high. In this case, some spark noise would be heard. In different audio framework (ALSA LIB, or PULSE AUDIO), different log about this noise may be printed. For example, in ALSA LIB, logs like "underrung!!! at least * ms is lost" are printed.

HDMI audio playback depends on HDMI pixel clock. So while in the state of HDMI blank and cable plug out, HDMI audio would be stopped or can't be played. See detailed infomation in software_operation_core.

Because HDMI audio driver needs to add an IEC header, the driver needs to know the quantity of data that the application writes into HDMI audio driver. If the application is not able to tell the data quantity (for example, DMIX plugin in ALSA LIB), the HDMI audio driver is not able to work properly. There would be no sound heard.

The HDMI audio supports the following features:

- Playback sample rate
  - 32k, 44.1k, 48k, 88.2k, 96k, 176.4k, 192k
  - capability of HDMI sink
- Playback Channels:

- 2, 4, 6, 8
- capability of HDMI sink

- Playback audio formats:
  - SNDRV_PCM_FMTBIT_S16_LE

11.2.6 CEC

HDMI CEC is a protocol that provides high-level control functions between all of the various audiovisual products. The HDMI CEC driver implements software part of HDMI CEC low-Level protocol. It includes getting Logical address, CEC message sending and receiving, error handle, and message re-transmitting.

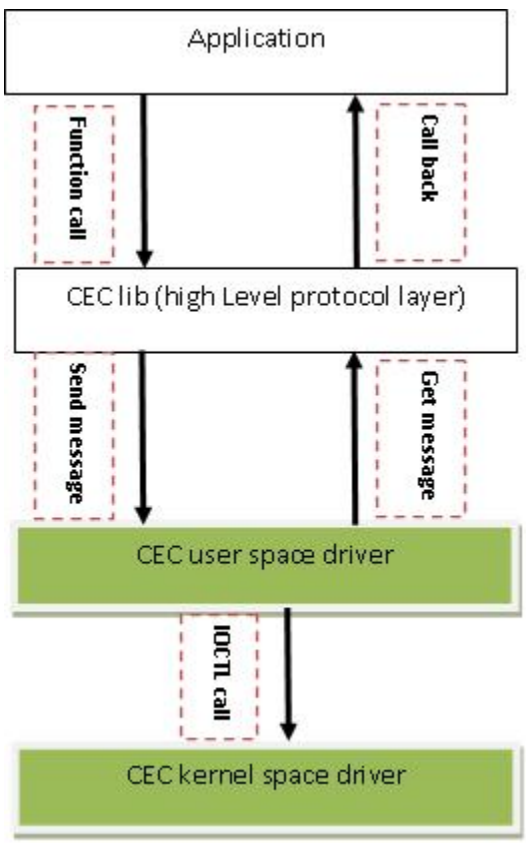


Figure 11-4. HDMI CEC SW Architecture

11.3 Source Code Structure

The bulk of the source code for the HDMI driver is divided amongst the three software components that comprise the driver: the HDMI core driver, the HDMI display driver, and the HDMI audio driver.

Additional platform-specific source code files provide the code for declaring and registering these HDMI drivers.

The source code for the HDMI core driver is available in the <ltib_dir>/rpm/BUIL D/ linux/drivers/mfd/ directory.

Table 11-1. HDMI Core Driver File List

File	Description
mxc-hdmi-core.c	HDMI core driver implemention

A public header for the HDMI core driver is available in the <ltib_dir>/rpm/BUIL D/ linux/include/linux/mfd/ directory.

Table 11-2. HDMI Core Display Driver Public Header File List

File	Description
mxc-hdmi-core.h	HDMI core driver header file

The source code for the HDMI display driver is available in the <ltib_dir>/rpm/BUIL D/ linux/drivers/video/ directory.

Table 11-3. HDMI Display Driver File List

File	Description
mxc_hdmi.c	HDMI display driver implemention

The source code for the HDMI audio driver is available in the <ltib_dir>/rpm/BUIL D/ linux/drivers and sound/soc/ directory. Although the HDMI is one hardware block, the audio driver is divided into four c files corresponding to the ALSA SoC layers:

Table 11-4. HDMI Audio Driver File List

File	Description
codecs/mxc_hdmi.c	HDMI Audio SoC codec driver implemention
imx/imx-hdmi-dai.c	HDMI Audio SoC DAI driver implemention
imx/imx-hdmi-dma.c	HDMI Audio SoC platform DMA driver implemention
imx/imx-hdmi.c	HDMI Audio SoC machine driver implemention

The source code for the HDMI CEC driver is available in the <ltib_dir>/rpm/BUIL D/ linux/drivers/mxc/ directory.

Table 11-5. HDMI CEC Driver File List

File	Description
drivers/mxc/hdmi-cec.c	HDMI CEC driver implemention

The source code for the HDMI lib is available in the <ltib_dir>/rpm/BUIL D/imx-lib/ hdmi-cec/ directory.

Table 11-6. HDMI CEC lib File List

File	Description
hdmi-cec/mxc_hdmi-cec.c	HDMI CEC lib implemention
hdmi-cec/hdmi-cec.h	HDMI CEC lib header file
hdmi-cec/android.mk	HDMI CEC lib make file

The following platform-level source code files provide structures and functions for registering the HDMI drivers. These files can be found in the <ltib_dir>/rpm/BUILD/ linux/arch/arm/plat-mxc/ directory.

Table 11-7. HDMI Platform File List

File	Description
devices/platform-mxc-hdmi-core.c	HDMI core driver platform device code
devices/platform-mxc_hdmi.c	HDMI display driver platform device code
devices/platform-imx-hdmi-soc.c	HDMI audio driver platform device code
devices/platform-imx-hdmi-soc-dai.c	HDMI audio driver platform device code
include/mach/mxc_hdmi.h	HDMI register defines

11.3.1 Linux Menu Configuration Options

There are three main Linux kernel configuration options used to select and include HDMI driver functionality in the Linux OS image.

The CONFIG_FB_MXC_HDMI option provides support for the HDMI video driver, and can be selected in menu configuration at the following menu location:

Device Drivers > Graphics support > MXC HDMI driver support

HDMI video support is dependent on support for the Synchronous Panel Framebuffer and also on the inclusion of IPUv3 support.

The CONFIG_SND_SOC_IMX_HDMI option provides support for HDMI audio through the ALSA/SoC subsystem, and can be found in menu configuration at the following location:

Device Drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio support for IMX - HDMI

When either of the previous two configuration options is selected, the MXC HDMI Core configuration option, CONFIG_MFD_MXC_HDMI will be selected. This option can also be found in the menu configuration here:

Device Drivers > Multifunction device drivers > MXC HDMI Core

The CONFIG_MXC_HDMI_CEC option provides support for the HDMI CEC driver, and can be selected in menu configuration at the following menu location:

Device Drivers > MXC support drivers > MXC HDMI CEC (Consumer Electronic Control) support

11.4 Unit Test

The HDMI video and audio drivers each have their own set of tests.

The HDMI video driver does not lend itself well to automated testing, so a number of manual tests are required to verify the correct functionality. For audio driver testing, the `aplay` audio file player and `iecset` utility provide confirmation of the the driver's proper integration into the ALSA framework. The following two section describe unit testing for both the HDMI audio and video drivers.

11.4.1 Video

The following set of manual tests can be used to verify the proper operation of the HDMI video driver:

1. Linux kernel command line-based tests: The initial mode used to display HDMI video can be specified through the Linux kernel command line boot parameters. Try several different valid display resolutions through the kernel parameters, re-booting the system each time and verifying that the desired resolution is displayed on the connected HDMI display.
2. Hotplug testing: Connect and disconnect the HDMI cable several times, from either the end attached to the i.MX board, or the end attached to the HDMI sink device. Each time the cable is reconnected, the driver should re-determine the appropriate

video mode, based on the modes read via EDID from the HDMI sink, and display that mode on the sink device.

3. HDMI output device testing: Test by dynamically switching the HDMI sink device. The HDMI driver should be able to detect the valid video modes for each different HDMI sink device and provide video to that display that is closest to the most recent video mode configured in the HDMI driver.

11.4.2 Audio

The following sequence of tests can verify the correct operation of the HDMI audio driver:

1. Ensure that an HDMI cable is connected between the i.MX board and the HDMI sink device, and that the HDMI video image is being properly displayed on the device.
2. Use 'aplay -l' (that's a single dash and a lower-case L) to list the audio playback cards and determine the card number. This is different on our various boards.
3. For example, if the hdmi ends up being card 2, use this command line to play out a pcm audio file "file.wav":

```
$ aplay -Dplughw:2,0 file.wav
```

4. Use 'iecset' to list out the IEC information about the device. You will need to specify card number like:

```
$ iecset -c2
```

NOTE

HDMI audio is dependent on a reasonable pixel clock rate being established. If this is not the case, error messages indicating “pixel clock not supported” will appear. This is because there is no clock regenerator cts value that could be calculated for the current pixel clock.

11.4.3 CEC

The following test can be used to simple verify HDMI CEC function:

```
$ /unit_test/mxc_cec_test
```

Bootup device and connect HDMI sink to board, and then run the above command. The HDMI CEC will send Poweroff command to HDMI sink.

Chapter 12
X Windows Acceleration

12.1 Introduction

X-Windows System (aka X11 or X) is a portable, client-server based, graphics display system.

X-Windows System can run with a default frame buffer driver which handles all drawing operations to the main display. Because there is a 2D GPU (graphics processing unit) available, some drawing operations can be accelerated. High level X operations may get decomposed into many low-level drawing operations where these low level operations are accelerated for X-Windows System.

12.2 Hardware Operation

X-Windows System acceleration on i.MX 6 uses the Vivante GC320 2D GPU.

Acceleration is also dependent on the frame buffer memory.

12.3 Software Operation

X-Windows acceleration is supported by X.org X Server version 1.7.6 and later versions, as well as the EXA interface version 2.5.

The types of operations that are accelerated for X11 are as follows. All operations involve frame buffer memory which may be on screen or off screen:

- Solid fill of a rectangle.
- Upload image from the system memory to the video memory.

- Copy of a rectangle with same pixel format with possible source-target rectangle overlap.
- Copy of a rectangle supporting most XRender compositing operations with these options:
  - Pixel format conversion.
  - Repeating pattern source.
  - Porter-Duff blending of source with target.
  - Source alpha masking.

The following are additional features supported by the X-Windows acceleration:

- X pixmaps can be allocated directly in frame buffer memory.
- EGL swap buffers where the EGL window surface is an X-window.
- X-window can be composited into an X pixmap which can be used directly as any EGL surface.

12.3.1 X Windows Acceleration Architecture

The following block diagram shows the components that are involved in the acceleration of X-Windows System:

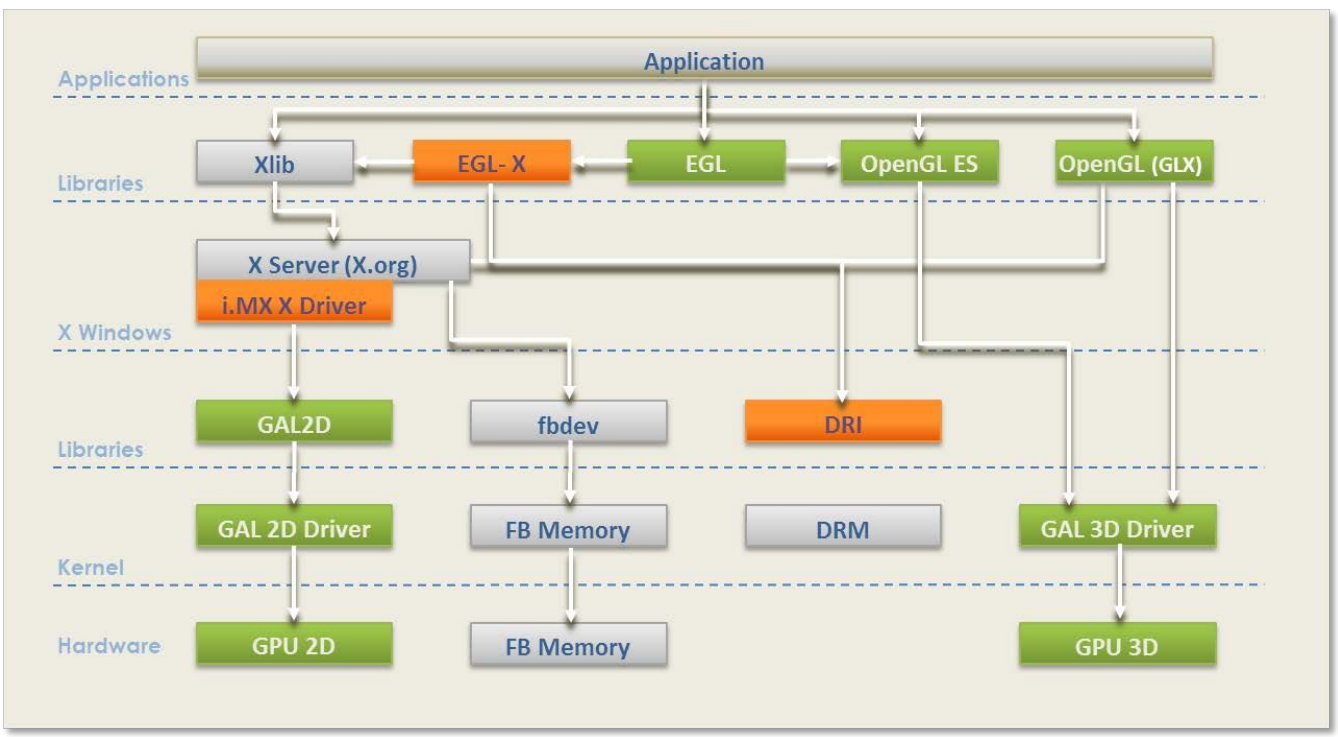


Figure 12-1. X Driver Architecture

The components shown in green are those provided as part of the Vivante 2D/3D GPU driver support which includes OpenGL/ES and EGL, though some of the families in the i.MX 6 series, such as the i.MX 6SoloLite, do not contain 3D HW module. The components shown in light gray are the standard components in the X-Windows System without acceleration. The components shown in orange are those added to support X-Windows System acceleration and are briefly described here.

The **i.MX X Driver** library module (`vivante-drv.so`) is loaded by the X server and contains the high-level implementation of the X-Windows acceleration interface for i.MX platforms containing the GC320 2D GPU core. The entire linearly contiguous frame buffer memory in `/dev/fb0` is used for allocating pixmaps for X both on screen and off screen. The driver supports a custom X extension which allows X clients to query the GPU address of any X pixmap stored in frame buffer memory.

The **libGAL.so** library module (`libGAL.so`) contains the register-level programming interface to the GC320 GPU module. This includes the storing of register programming commands into packets which can be streamed to the device. The functions in the libGAL.so library are called by the i.MX X Driver code.

The **EGL-X** library module (`libEGL.so`) contains the X-Windows implementation of the low level EGL platform-specific support functions. This allows X-window and X pixmap objects to be used as EGL window and pixmap surfaces. The EGL-X library uses Xlib function calls in its implementation along with the i.MX X Driver module's X extension for querying the GPU address of X pixmaps stored in frame buffer memory.

12.3.2 i.MX 6 Driver for X-Windows System

The i.MX X Driver, referred to as vivante-drv.so, implements the EXA interface of the X server in providing acceleration.

The implementation details are as follows:

- The implementation builds upon the source from the fbdev frame buffer driver for X, so that it can be the fallback when the acceleration is disabled.
- The implementation is based on X server EXA version 2.5.0.
- The EXA solid fill operation is accelerated, except for source/target drawables containing less than 1024x1024 pixels, in which case software failure may occur.
- The EXA copy operation is accelerated, except for source/target drawables containing less than 1024x1024 pixels, in which case software failure may occur.
- EXA putimage (upload into video memory) is accelerated, except for source drawables containing less than 400x400 pixels, in which case software failure may occur.

- For EXA solid fill, only solid plane masks and only GXcopy raster-op operations are accelerated.
- For EXA copy operation, the raster-op operations (GXandInverted, GXnor, GXorReverse, GXorInverted, GXnand) are not accelerated.
- EXA composite allows for many options and combinations of source/mask/target for rendering. Commonly used EXA composite operations are accelerated.

The following types of EXA composite operations are accelerated:

- Composite operations for source/target drawables containing at least 640 pixels. If less than 640 pixels, the composite path falls to software.
- Simple source composite operations are used when source/target drawables contain more than 1024x1024 pixels (operations with mask not supported).
- Constant source (with or without alpha mask) composite with target.
- Repeating pattern source (with or without alpha mask) composite with target.
- Only these blending functions: SOURCE, OVER, IN, IN-REVERSE, OUT-REVERSE, and ADD (some of these need to support the accelerated component-alpha blending).
- In general, the following types of less commonly used EXA composite operations are not accelerated:
  - Transformed (meaning scaled, rotated) sources and masks.
  - Gradient sources.
  - Alpha masks with repeating patterns.

The implementation handles all pixmap allocation for X through the EXA callback interface. A first attempt is made to allocate the memory where it can be accessed by a physical GPU address. This attempt may fail if there is insufficient GPU accessible memory remaining, but it can also fail when the bits per pixel, which are being requested for the pixmap, are less than 8. If the attempt to allocate from the GPU accessible memory fails, the memory is allocated from the system. If the pixmap memory is allocated from the system, this pixmap cannot be involved in GPU accelerated option. The number of pitch bytes used to access the pixmap memory may be different depending on whether it was allocated from GPU accessible memory or from the system.

Once the memory for X pixmap has been allocated, no matter it is from GPU accessible memory or from the system, the pixmap is locked and can never migrate to other type of memory. Pixmap migration from GPU accessible memory to system memory is not necessary since a system virtual address is always available for GPU accessible memory. Pixmap migration from system memory to GPU accessible memory is not currently implemented, but would only help in situations where there was insufficient GPU accessible memory at initial allocation. More memory, however, becomes available (through de-allocation) at a later time.

The GPU accessible memory pitch (horizontal) alignment for the GC320 is 8 pixels. Because the memory can be allocated from GPU accessible memory, these pixels could be used in EGL for OpenGL/ES drawing operations.

All of the memory allocated for `/dev/fb0` is made available to an internal linear offscreen memory manager based on the one used in EXA. The portion of this memory beyond the screen memory is available for allocation of X pixmap, where this memory area is GPU accessible. The amount of memory allocated to `/dev/fb0` needs to be several MB more than the amount needed for the screen. The actual amount needed depends on the number of X-Windows and pixmaps used, the possible usage of X pixmaps as textures, and whether X-Windows are using the XComposite extension.

X extension is provided, so that X clients can query the physical GPU address associated with an X pixmap if that X pixmap was allocated in the GPU accessible memory.

12.3.3 i.MX 6 Direct Rendering Infrastructure (DRI) for X-Windows System

The Direct Rendering Infrastructure, also known as the DRI, is a framework that allows direct access to graphics hardware under the X Window System in a safe and efficient manner. It includes changes to the X server, to several client libraries, and to the kernel (DRM, Direct Rendering Manager). The most important function for the DRI is to create fast OpenGL and OpenGL ES implementations that render to framebuffer memory directly. Without DRI, the OpenGL driver has to depend on X server for final rendering (indirect rendering), which degrades the overall performance significantly.

The components of Vivante's DRI OpenGL implementation include:

- The Direct Rendering Manager (DRM) is a kernel module that provides APIs to userland to synchronize access to hardware and to manage different classes of video memory buffers. Vivante's DRI implementation uses selected DRM APIs for opening/closing DRI device, and locking/unlocking FB. Most other buffer management and DMA management functions are handled by Vivante's specific kernel module: galcore.ko.
- The EXA driver is a DRI-enabled DDX 2D driver which initializes the DRM when X server starts. As all X Window pixmap buffers are allocated by the EXA driver from GPU memory, the GPU can render directly into these buffers if the buffer information is passed from the X server process to the X client processes (GL or GLES applications) properly.
- The libdri.so implements Vivante's customized DRI protocol that passes the buffer information between X server and X clients (GL or GLES applications).

The integration of GL/GLES application windows with Ubuntu Unity2D desktop is achieved by following steps:

1. GL/GLES applications render a frame into the pixmap buffers that are allocated in the EXA driver.
2. In the SwapBuffers implementation, the driver notifies X server that the pixmap buffer region is damaged through Xdamage and Xfixes APIs.
3. Then the X server will present the latest pixmap buffer to the Unity2D desktop while maintaining the proper window overlap characteristics relative to the other windows on the desktop.

12.3.4 EGL- X Library

The EGL-X library implements the low-level EGL interface used in X-Windows System.

The implementation details are as follows:

- The `eglDisplay` native display type is "`Display*`" in X-Windows.
- The `eglWindowSurface` native window surface type is "`Window`" in X-Windows.
- The `eglPixmapSurface` native pixmap surface type is "`Pixmap`" in X-Windows.

When an `eglWindowSurface` is created, the back buffers, used for double-buffering, can have different representations from the window surface (based on the selected `eglConfig`). An attempt is made to create each back buffer by using the representation, which provides the most efficient blit of the back buffer contents to the window surface when `eglSwapBuffers` is called.

The back buffer is allocated by creating an X pixmap of the necessary size. Use the X extension for the i.MX X Driver module to query the physical frame buffer address for this X pixmap if it was allocated in the offscreen frame buffer memory.

12.3.5 xorg.conf for i.MX 6

The /etc/X11/xorg.conf file must be properly configured to use the i.MX 6 X Driver.

This configuration appears in a `Device` section of the file which contains both mandatory and optional entries. The following example shows a preferred configuration for using the i.MX 6 X Driver:

```
Section "Device"
        Identifier      "i.MX Accelerated Framebuffer Device"
        Driver          "vivante"
        Option          "fbdev"         "/dev/fb0"
        Option          "vivante_fbdev" "/dev/fb0"
```

```
        Option          "HWcursor"      "false"
EndSection



Section "Monitor"
        Identifier      "Configured Monitor"
EndSection



Section "Screen"
        Identifier      "Default Screen"
        Monitor         "Configured Monitor"
        Device          "i.MX Accelerated Framebuffer Device"
EndSection



Section "ServerLayout"
        Identifier      "Default Layout"
        Screen          "Default Screen"
EndSection
```

Some important entries recognized by the i.MX X Driver are described as follows:

- Device Identifier and Screen Device String

  The mandatory Identifierentry in the Device section specifies the unique name to associate with this graphics device.

  ```
  Section "Device"
          Identifier      "i.MX Accelerated Framebuffer Device"
  ```

  The following entry ties a specific graphics device to a screen. The Device Identifier string must match the `Device` string in a `Screen` section of the `xorg.conf` file. For example,

  ```
  Section "Screen"
          ...
          Device          "i.MX Accelerated Framebuffer Device"
          ...
  EndSection
  ```

- Device Driver String

  The mandatory Driver entry specifies the name of the loadable i.MX X driver.

  ```
  Section "Device"
          ...
          Driver          "vivante"
          ...
  EndSection
  ```

- Device fbdevPath Strings

The mandatory entries fbdev and vivante_devspecifies the path for the frame buffer device to use.

```
Section "Device"
        ...
        Option          "fbdev"         "/dev/fb0"
        Option          "vivante_fbdev" "/dev/fb0"
        ...
EndSection
```

12.3.6 Setup X-Windows System Acceleration

Setup of X-Windows system acceleration consists of package installation and verification, file verification, and verifying acceleration. The debian packages are only available for ubuntu root fs. There's no gpu driver for X11 on gnome mobile root fs or LTIB

- Package installation and verification:
  - Verify that the following packages are available and installed:

    `gpu-viv-bin-mx6q_<bsp-version>_armel.deb`
  - This package contains gpu driver develop headers, and is installed in the `/usr/include` folder
  - This package contains gpu driver hal library`libGAL.so`
  - This package contains 3D client libraries, include `libEGL.so`, `libGLESv1_.so`, `libGLESv2.so`, libGL.so, vivante_dri.so ... library
  - All above libraries are installed in the `/usr/lib` folder except vivante_dri.so, which is installed at /usr/lib/dri
  - `xserver-xorg-video-imx-viv_<bsp-version>_armel.deb`
  - This package contains the vivante-drv.so and libdri.so driver module for X acceleration and is installed in the folder with all the other X.org driver modules
- File verification:
  - Verify that the device file `/dev/galcore` is present.
  - Verify that the file `/etc/X11/xorg.conf` contains the correct entries as described in the previous section.
- Verify acceleration:
  - Assuming the above steps have been performed, do the following to verify that X Window System acceleration is indeed operating.
  - Examine the log file `/var/log/Xorg.0.log` and confirm that the following lines present:

```
[    33.767] (II) LoadModule: "vivante"
[    33.782] (II) Loading /usr/lib/xorg/modules/drivers/vivante_drv.so
...
```

```
[    33.881] (II) VIVANTE(0): using default device
[    33.881] (WW) VGA arbiter: cannot open kernel arbiter, no multi-card support
[    33.881] (II) VIVANTE(0): Creating default Display subsection in Screen section
    "Default Screen" for depth/fbbpp 16/16
[    33.881] (==) VIVANTE(0): Depth 16, (==) framebuffer bpp 16
[    33.881] (==) VIVANTE(0): RGB weight 565
[    33.881] (==) VIVANTE(0): Default visual is TrueColor
[    33.881] (==) VIVANTE(0): Using gamma correction (1.0, 1.0, 1.0)
[    33.881] (II) VIVANTE(0): hardware: mxc_elcdif_fb (video memory: 2250kB)
[    33.882] (II) VIVANTE(0): checking modes against framebuffer device...
[    33.882] (II) VIVANTE(0): checking modes against monitor...
[    33.882] (--) VIVANTE(0): Virtual size is 800x480 (pitch 800)
[    33.882] (**) VIVANTE(0):  Built-in mode "current": 33.5 MHz, 31.2 kHz, 58.6 Hz
[    33.882] (II) VIVANTE(0): Modeline "current"x0.0   33.50  800 964 974 1073  480
490 500 533 -hsync -vsync -csync (31.2 kHz)
[    33.882] (==) VIVANTE(0): DPI set to (96, 96)
...
[    34.228] (II) VIVANTE(0): FB Start = 0x333a8000  FB Base = 0x333a8000  FB
Offset = (nil)
[    34.228] (II) VIVANTE(0): test Initializing EXA
[    34.228] (II) EXA(0): Driver allocated offscreen pixmaps
[    34.229] (II) EXA(0): Driver registered support for the following operations:
[    34.229] (II)         Solid
[    34.229] (II)         Copy
[    34.229] (II)         Composite (RENDER acceleration)
[    34.229] (II)         UploadToScreen
[    34.244] (==) VIVANTE(0): Backing store disabled
[    34.244] (==) VIVANTE(0): DPMS enabled
```

- Note:
  - Although there's some error info for AIGLX, it could be ignored. AIGLX is for glx non-DRI implementation, while our glx library will never call AIGLX module at X server, since our glx is a library for DRI.

```
[     7.205] (EE) AIGLX error: vivante exports no extensions (/usr/lib/arm-linux-
gnueabihf/dri/vivante_dri.so: undefined symbol: __driDriverExtensions)
[     7.214] (EE) AIGLX: reverting to software rendering
[     7.278] (II) AIGLX: Loaded and initialized swrast
[     7.278] (II) GLX: Initialized DRISWRAST GL provider for screen 0
```

Chapter 13
Video Processing Unit (VPU) Driver

13.1 Hardware Operation

The VPU hardware performs all of the codec computation and most of the bitstream parsing/packeting.

Therefore, the software takes advantage of less control and effort to implement a complex and efficient multimedia codec system.

The VPU hardware data flow is shown in the MPEG4 decoder example in the following figure.

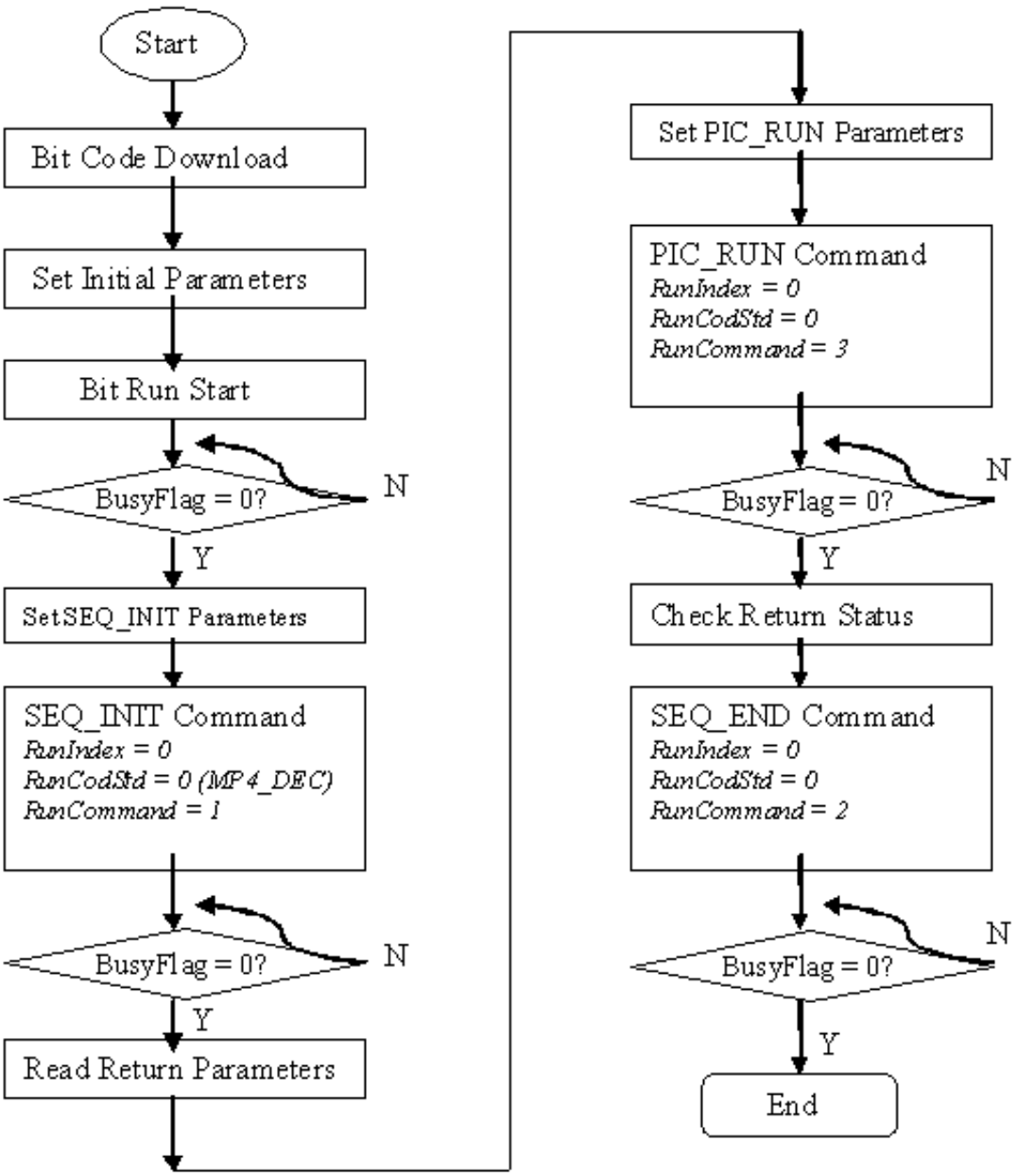


Figure 13-1. VPU Hardware Data Flow

13.1.1 Software Operation

The VPU software can be divided into two parts: the kernel driver and the user-space library as well as the application in user space. The kernel driver takes responsibility for system control and reserving resources (memory/IRQ). It provides an IOCTL interface

for the application layer in user-space as a path to access system resources. The application in user-space calls related IOCTLs and codec library functions to implement a complex codec system.

The VPU kernel driver includes the following functions:

- Module initialization which initializes the module with the device specific structure
- Device initialization which initializes the VPU clock and hardware and request the IRQ
- Interrupt servicing routine which supports events that one frame has been finished
- File operation routine which provides the following interfaces to user space:
    - File open
    - File release
    - File synchronization
    - File IOCTL to provide interface for memory allocating and releasing
- Memory map for register and memory accessing in user space
- Device Shutdown: shuts down the VPU clock and hardware, and releases the IRQ

The VPU user space driver has the following functions:

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

13.1.2 Source Code Structure

The following table lists the kernel space source files available in the following directories:

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

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

Table 13-1. VPU Driver Files

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

The following table lists the user-space library source files available in the <ltib_dir>/rpm/BUILD/imx-lib-11.11.00/vpu directory:

Table 13-2. VPU Library Files

File	Description
vpu_io.c	Interfaces with the kernel driver for opening the VPU device and allocating memory
vpu_io.h	Header file for IOCTLs
vpu_lib.c	Core codec implementation in user space
vpu_lib.h	Header file of the codec
vpu_reg.h	Register definition of VPU
vpu_util.c	File implementing common utilities used by the codec
vpu_util.h	Header file

The following table lists the firmware files available in the following directories:

`<ltib_dir>/rpm/BUILD/firmware-imx-11.11.00/lib/firmware/vpu/ directory`

Table 13-3. VPU firmware Files

File	Description
vpu_fw_xxx.bin	VPU firmware

NOTE

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

13.1.3 Menu Configuration Options

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

- CONFIG_MXC_VPU, provided for the VPU driver. In menu configuration, this option is available under
- Device Drivers > MXC support drivers > MXC VPU (Video Processing Unit) support

13.1.4 Programming Interface

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

The codec library APIs are listed below:

```
RetCode vpu_Init(void *);
void vpu_UnInit(void);
RetCode vpu_GetVersionInfo(vpu_versioninfo * verinfo);

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

System I/O operations are listed below:

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

13.1.5 Defining an Application

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

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

For how to use API in the application, refer to i.MX 6Dual/6Quad VPU Application Programming Interface Linux Reference Manual.

Chapter 14
OmniVision Camera Driver

14.1 OV5640 Using MIPI CSI-2 interface

This is an introduction to the OV5640 camera driver that uses the MIPI CSI-2 interface.

14.1.1 Hardware Operation

The OV5640 is a small camera sensor and lens module with low-power consumption. The camera driver is located under the Linux V4L2 architecture and it implements the V4L2 capture interfaces. Applications cannot use the camera driver directly. Instead, the applications use the V4L2 capture driver to open and close the camera for preview and image capture, controlling the camera, getting images from camera, and starting the camera preview.

The OV5640 uses the serial camera control bus (SCCB) interface to control the sensor operation. It works as an I^2C client. V4L2 driver uses I^2C bus to control camera operation.

OV5640 supports two transfer modes: parallel interface and MIPI interface.

When using MIPI mode, OV5640 connects to i.MX AP chip through the MIPI CSI-2 interface. MIPI receives the sensor data and transfers them to IPU CSI.

Refer to OV5640 datasheet to get more information on the sensor.

Refer to the *i.MX 6 Multimedia Applications Processor Reference Manual* for more information on MIPI CSI-2 and IPU CSI.

14.1.2 Software Operation

The camera driver implements the V4L2 capture interface and applications and uses the V4L2 capture interface to operate the camera.

The supported operations of V4L2 capture are:

- Capture stream mode

The supported picture formats are:

- YUV422P
- UYVY
- YUV420

The supported picture sizes are:

- QVGA
- VGA
- 720P
- 1080P

14.1.3 Source Code Structure

Table below shows the camera driver source files available in the directory.

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

Table 14-1. Camera Driver Files

File	Description
ov5640_mipi.c	Camera driver implementation for ov5640 using MIPI CSI-2 interface

14.1.4 Linux Menu Configuration Options

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

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

- Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > OmniVision ov5640 camera support using mipi.

14.2 OV5640 Using parallel interface

This is an introduction to the OV5640 camera driver that uses the parallel interface.

14.2.1 Hardware Operation

The OV5640 is a small camera sensor and lens module with low-power consumption. The camera driver is located under the Linux V4L2 architecture. and it implements the V4L2 capture interfaces. Applications cannot use the camera driver directly. Instead, the applications use the V4L2 capture driver to open and close the camera for preview and image capture, controlling the camera, getting images from camera, and starting the camera preview.

The OV5640 uses the serial camera control bus (SCCB) interface to control the sensor operation. It works as an I^2C client. V4L2 driver uses I^2C bus to control camera operation.

OV5640 supports only parallel interface.

Refer to OV5640 datasheet to get more information on the sensor.

Refer to the *i.MX 6 Multimedia Applications Processor Reference Manual* for more information on CSI.

14.2.2 Software Operation

The camera driver implements the V4L2 capture interface and applications and uses the V4L2 capture interface to operate the camera.

The supported operations of V4L2 capture are:

- Capture stream mode
- Capture still mode

The supported picture formats are:

- UYVY
- YUYV

The supported picture sizes are:

- QVGA

- VGA
- 720P
- 1080P
- QSXGA

14.2.3 Source Code Structure

Table below shows the camera driver source files available in the directory.

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

Table 14-2. Camera Driver Files

File	Description
ov5640.c	Camera driver implementation for ov5640 using parallel interface

14.2.4 Linux Menu Configuration Options

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

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

- Device Drivers > Multimedia devices > Video capture adapters > MXC Video For Linux Camera > MXC Camera/V4L2 PRP Features support > OmniVision ov5640 camera support.

Chapter 15
MIPI CSI2 Driver

15.1 Introduction

MIPI CSI-2 for i.MX 6 is MIPI-Camera Serial Interface Host Controller. It is a high performance serial interconnect bus for mobile application which connects camera sensors to the host system. The CSI-2 Host Controller is a digital core that implements all protocol functions defined in the MIPI CSI-2 Specification. In doing so, it provides an interface between the system and the MIPI D-PHY and allows communication with MIPI CSI-2 compliant Camera Sensor.

The MIPI CSI2 driver is used to manage the MIPI D-PHY and allows it to co-work with MIPI sensor and IPU CSI. MIPI CSI2 driver implements functions as follows:

- MIPI CSI-2 low-level interface for managing the mipi D-PHY register and clock
- MIPI CSI-2 common API for communication between MIPI sensor and MIPI D-PHY

By calling MIPI common APIs, MIPI sensor can set certain information about sensor (such as datatype, lanes number, etc.) to MIPI CSI2 driver to configure D-PHY. In order for the IPU CSI module driver to have the correct configuration, receive appropriate data, and process it correctly, it is necessary for it to receive information about sensor (such as datatype, virtual channel, IPU id, CSI id, etc.) from the MIPI CSI2 driver.

15.1.1 MIPI CSI2 Driver Overview

MIPI CSI2 driver is invoked only by the mipi sensor driver and IPU CSI module. It is not exposed to the user space.

MIPI CSI2 driver supports the following features:

- Supporting 1-4 lanes
- Supporting IPU(0,1) and CSI(0,1).

- Supporting 0-3 virtual channels.
- Supporting the following date types:
  - RGB formats: RGB888, RGB666, RGB565, RGB555, RGB444
  - YUV formats: YUV422 8bit, YUV422 10bit, YUV420 8bit, YUV420 10bit
  - RAW data: RAW6, RAW7, RAW8, RAW10, RAW12, RAW14

15.1.2 Hardware Operation

There are four blocks in the MIPI CSI-2 D-PHY: PHY adaptation layer, packet analyzer, image date interface, and register bank.

Functions and operations are listed as follows:

- PHY Adaptation Layer is responsible for managing the D-PHY interface, including PHY error handling.
- Packet Analyzer is responsible for data lane merging if required, together with header decoding, error detection and correction, frame size error detection and CRC error detection.
- Image Date Interface separates CSI-2 packet header information and reorders data according to memory storage format. It also generates timing accurate video synchronization signals. Several error detections are also performed at frame level and line level.
- Register Bank is accessible through a standard AMBA-APB slave interface and provides access to the CSI-2 Host Controller register for configuration and control. There is also a fully programmable interrupt generator to inform the system upon certain events.

15.2 Software Operation

MIPI CSI2 driver for Linux has two parts:

- MIPI CSI2 driver: initializes the mipi_csi2_info structure
- MIPI CSI2 common APIs: exports APIs for the CSI module driver and mipi sensor driver

15.2.1 MIPI CSI2 Driver Initialize Operation

The steps for MIPI CSI2 driver initialization are as follows:

1. MIPI CSI2 driver initializes the mipi_csi2_info structure, some key information about mipi sensor, such as connected IPU ID, CSI ID, virtual channel and date type.
2. The driver initilizes the D-PHY clock and pixel clock. The pixel clock is used for MIPI D-PHY to transfer data to IPU CSI.
3. The driver waits for sensor connection.

15.2.2 MIPI CSI2 Common API Operation

MIPI CSI2 driver exports a large number of APIs to manage MIPI D-PHY.

The following is the introduction to all APIs:

- mipi_csi2_get_info: get the mipi_csi_info
- mipi_csi2_enable: enable mipi csi2 interface
- mipi_csi2_disable: disable mipi csi2 interface
- mipi_csi2_get_status: get mipi csi2 interface disable/enable status
- mipi_csi2_get_bind_ipu: get the ipu id which mipi csi2 will connect
- mipi_csi2_get_bind_csi: get the csi id which mipi csi2 will connect
- mipi_csi2_get_virtual_channel: get the virtual channel number by which mipi sensor will tansfer data to MIPI D-PHY
- mipi_csi2_set_lanes: set the lanes number by which mipi sensor will tansfer data to MIPI D-PHY
- mipi_csi2_set datatype: set the mipi sensor data type
- mipi_csi2_get_datatype: get the mipi sensor data type; This function will be called by csi module to set csi register
- mipi_csi2_dphy_status: get the MIPI D-PHY status
- mipi_csi2_get_error1: get the mipi error1 register information
- mipi_csi2_get_error2: get the mipi error2 register informaiton
- mipi_csi2_pixelclk_enable: enable the pixel clock
- mipi_csi2_pixelclk_disable: disable the pixel clock
- mipi_csi2_reset: reset the MIPI D-PHY for data receiving and transferring

15.3 Driver Features

MIPI CSI2 driver supports the following features:

- Supporting 1-4 lanes
- Supporting IPU(0,1) and CSI(0,1)
- Supporting 0-3 virtual channels
- Supporting the following date types:

- RGB formats: RGB888, RGB666, RGB565, RGB555, RGB444
- YUV formats: YUV422 8bit, YUV422 10bit, YUV420 8bit, YUV420 10bit
- RAW data: RAW6, RAW7, RAW8, RAW10, RAW12, RAW14

15.3.1 Source Code Structure

Table below shows the MIPI CSI2 driver source files available in the directory.

<ltib_dir>/rpm/BUILD/linux/drivers/mxc/mipi.

Table 15-1. MIPI CSI2 Driver Files

File	Description
mxc_mipi_csi2.c	mipi csi2 driver source file

15.3.2 Menu Configuration Options

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

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

Device Drivers > MXC support drivers > MXC MIPI Support > MIPI CSI2 support.

15.3.3 Programming Interface

MIPI CSI2 Common APIs can only be called by the MIPI sensor driver and IPU CSI module driver.

Before calling the API, in system initialization stage, use the mipi_csi2_platform_data structure and imx6q_add_mipi_csi2 funciton to add a MIPI CSI2 driver.

For the MIPI sensor driver, the initialization steps are as follows:

- Get MIPI information by calling mipi_csi2_get_info().
- Enable the MIPI CSI2 interface by calling mipi_csi2_enable().
- Set the lanes by calling mipi_csi2_set_lanes().
- Reset the MIPI D-PHY by calling mipi_csi2_reset().
- Configure the MIPI sensor.

- Wait for MIPI D-PHY to receive the sensor clock and data until clock and data are stable by calling mipi_csi2_dphy_status() and mipi_csi2_get_error1().
- When uninstalling the sensor driver, disable the MIPI CSI2 interface by calling mipi_csi2_disable().

For sample code that explains how the MIPI sensor uses MIPI APIs, refer to the OV5640_mipi driver source code.

For the IPU CSI module driver, the call steps are as follows:

- Get the MIPI information by calling mipi_csi2_get_info().
- Get the IPU ID and CSI ID to assure configuration of the correct CSI module by calling mipi_csi2_get_bind_ipu() and mipi_csi2_get_bind_csi().
- Get the data type and virtual channel from MIPI CSI2 driver and configure the CSI module by calling mipi_csi2_get_datatype() and mipi_csi2_get_virtual_channel().
- Perform other configuration operations for the CSI module and enable CSI.
- Enable the pixel clock to transfer data from MIPI D-PHY to IPU CSI by calling mipi_csi2_pixelclk_enable().
- When all tasks are done, disable the CSI module first,and then disable the MIPI pixel clock by calling mipi_csi2_pixelclk_disable().

For sample code that explains how CSI module driver uses MIPI APIs, refer to the IPU CSI module driver source code.

15.3.4 Interrupt Requirements

No interrupt is needed for the MIPI CSI2 driver.

Chapter 16
Low-level Power Management (PM) Driver

16.1 Hardware Operation

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

The i.MX 6 supports four low-power modes: RUN, WAIT, STOP, and DORMANT.

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

Table 16-1. Low Power Modes

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

For the detailed information about lower power modes, see the *MCIMX 6Dual/6Quad Multimedia Applications Processor Reference Manual (MCIMX6DQRM)*.

16.1.1 Software Operation

The i.MX 6 PM driver maps the low-power modes to the kernel power management states as follows:

- Standby: maps to STOP mode that offers significant power saving, as all blocks in the system are put into a low-power state, except for ARM core that is still powered on, and memory is placed in self-refresh mode to retain its contents.

- Mem (suspend to RAM): maps to DORMANT mode that offers most significant power saving as all blocks in the system are put into a low-power state, except for memory that is placed in self-refresh mode to retain its contents.
- System idle: maps to WAIT mode.

The i.MX 6 PM driver performs the following steps to enter and exit low-power mode:

1. Allow the Coretex-A9 platform to issue a deep-sleep mode request.
2. If it is in STOP or DORMANT mode:
   - Program CCM CLPCR register to set low-power control register.
   - If it is in DORMANT mode, request switching off CPU power when pdn_req is asserted.
   - Request switching off embedded memory peripheral power when pdn_req is asserted.
   - Program GPC mask register to unmask wakeup interrupts.
3. Call cpu_do_idle to execute WFI pending instructions for wait mode.
4. Execute mx6_do_suspend in IRAM.
5. If it is in DORMANT mode, save the ARM context, change the drive strength of MMDC PADs to "low" to minimize the power leakage in DDR PADs. Execute WFI pending instructions for stop mode.
6. Generate a wakeup interrupt and exit low-power mode. If it is in DORMANT mode, restore the ARM core and DDR drive strength.

In DORMANT and STOP mode, the i.MX 6 can assert the VSTBY signal to the PMIC and request a voltage change. The Machine Specific Layer (MSL) usually sets the standby voltage in STOP mode according to i.MX 6 data sheet.

16.1.2 Source Code Structure

Table below shows the PM driver source files. These files are available in <ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/.

Table 16-2. PM Driver Files

File	Description
pm.c	Supports suspend operation
system.c	Supports low-power modes
mx6_suspend.S	Assembly file for CPU suspend

16.1.3 Menu Configuration Options

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

- CONFIG_PM builds support for power management. In menu configuration, this option is available under:
  - Power management options > Power Management support
  - By default, this option is Y.
- CONFIG_SUSPEND builds support for suspend. In menu configuration, this option is available under:
  - Power management options > Suspend to RAM and standby

16.1.4 Programming Interface

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

16.1.5 Unit Test

To enter different system-level low-power modes:

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

To wake up system from low-power modes:

```
enable wake up source first, USB device, debug uart or RTC etc.
can be used as wakeup source, below is the example of uart wakeup and rtc wakeup:
echo enabled > /sys/devices/platform/imx-uart.'x'/tty/ttymxc'x'/power/wakeup; Here 'x' is
your debug uart's index;
echo +x > /sys/class/rtc/rtc0/wakealarm; RTC will wake up system after x seconds.
```

To test this mode automatically, refer to the script in /unit_tests/suspend_auto.sh

Chapter 17
PF100 Regulator Driver

17.1 Introduction

PF100 is a PMIC chip that is implemented on the i.MX 6 series development platforms.

The PF100 regulator driver provides the low-level control of the power supply regulators, selection of voltage levels, and enabling/disabling of regulators. This device driver makes use of the PF100 core driver to access the PF100 hardware control registers. The PF100 core driver is based on the MFD structure and it is attached to the kernel I2C bus.

17.2 Hardware Operation

PF100 provides reference and supply voltages for the application processor and peripheral devices.

Four buck (step down) converters (up to 6 independent output) and one boost (step up) converter are included. The buck converters provide the power supply to processor cores and to other low voltage circuits such as memory. Dynamic voltage scaling is provided to allow controlled supply rail adjustments for the processor cores and other circuitry.

Linear regulators are directly supplied from the battery or from the switchers, including supplies for I/O and peripherals, audio, camera, BT, and WLAN. Naming conventions are suggestive of typical or possible use case applications, but the switchers and regulators may be used for other system power requirements within the guidelines of specified capabilities.

The only power on event of PF100 is that PWRON is high, and the only power off event of PF100 is that PWRON is low. PMIC_ON_REQ pin of i.MX 6, which is controlled by SNVS block of i.MX 6, will connect with PWRON pin of PF100 to control PF100 on/off, so that system can power off.

17.2.1 Driver Features

The PF100 regulator driver is based on PF100 core driver and regulator core driver. It provides the following services for regulator control of the PMIC component:

- Switch ON/OFF all voltage regulators.
- Set the value for all voltage regulators.
- Get the current value for all voltage regulators.
- Write/Read PF100 registers by sysfs interface.

17.3 Software Operation

PF100 regulator client driver performs operations by reconfiguring the PMIC hardware control registers.

This is done by calling PF100 core driver APIs with the required register settings.

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

17.3.1 Regulator APIs

The regulator power architecture is designed to provide a generic interface to voltage and current regulators within the Linux kernel.

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

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

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

- `regulator_get` is an unified API call to lookup and obtain a reference to a regulator:

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

- `regulator_put` is an unified API call to free the regulator source:

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

- `regulator_enable` is an unified API call to enable regulator output:

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

- `regulator_disable` is an unified API call to disable regulator output:

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

- `regulator_is_enabled` is the regulator output enabled:

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

- `regulator_set_voltage` is an unified API call to set regulator output voltage:

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

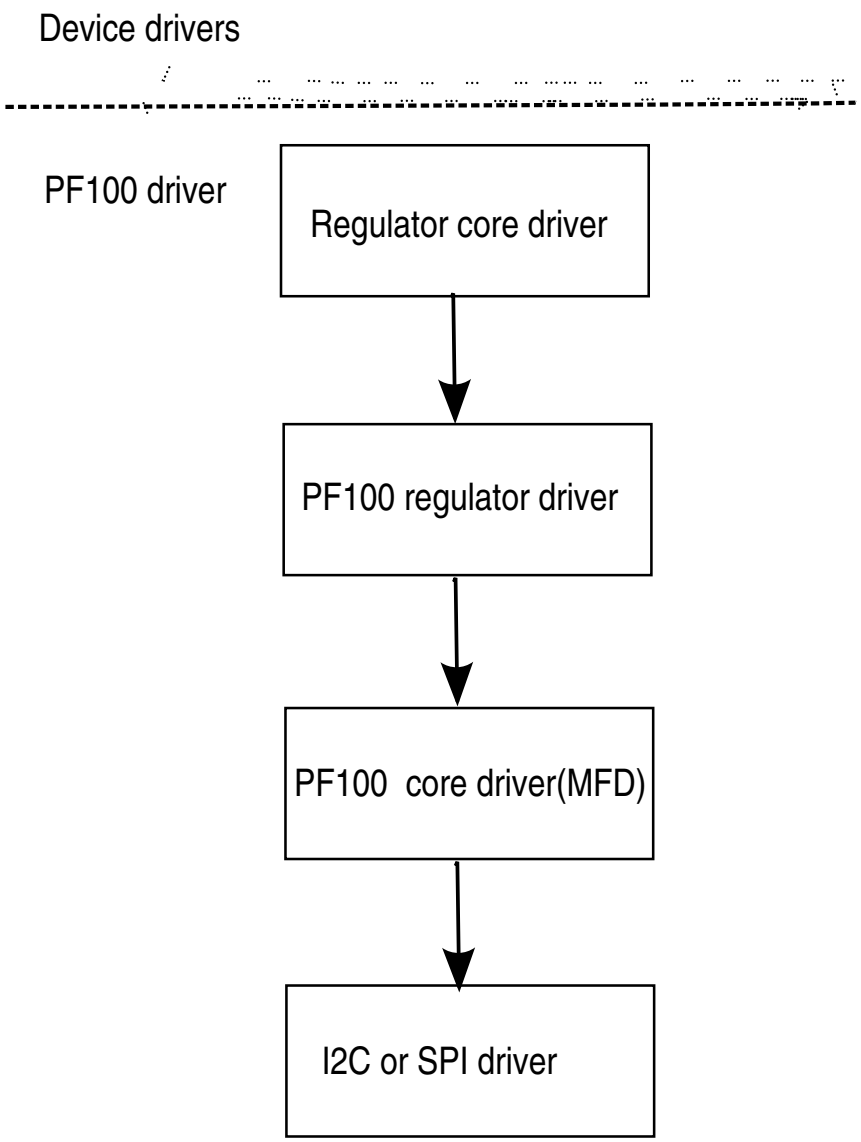
- `regulator_get_voltage` is an unified API call to get regulator output voltage:

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

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

17.4 Driver Architecture

Figure below shows the basic architecture of the PF100 regulator driver.



17.4.1 Driver Interface Details

Access to PF100 regulator is provided through the API of the regulator core driver.

The PF100 regulator driver provides the following regulator controls:

- Four buck switch regulators on normal mode (up to 6 independent rails): SW1AB, SW1C, SW2, SW3A, SW3B, and SW4.
- Buck switch can be programmed to a state of standby with specific register (PF100_SWxSTANDBY) in advance.
- Six Linear Regulators: VGEN1, VGEN2, VGEN3, VGEN4, VGEN5, and VGEN6.
- One LDO/Switch supply for VSNVS support on i.MX processors.
- One Low current, high accuracy, voltage reference for DDR Memory reference voltage.
- One Boost regulator with USB OTG support.
- Most power rails from PF100 have been programmed properly according to the hardware design. Therefore, you cannot find the kernel by using PF100 regulators. The PF100 regulator driver has implemented these regulators so that customers can use it freely if default PF100 value can't meet their hardware design.

17.4.2 Source Code Structure

The PF100 regulator driver is located in the regulator device driver directory:

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

Table 17-1. PF100 core Driver Files

File	Description
`drivers/mfd/pf100-core.c`	Linux kernel interface for regulators.
`drivers/regulator/pf100-regulator.c`	Implementation of the PF100 regulator client driver.

The PF100 regulators for MACH_MX6Q_SABRESD board are registered under <ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/mx6q_sabresd_pmic_pf100.c.

The PF100 regulators for MACH_MX6Q_SABREAUTO board are registered under <ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/mx6q_sabreauto_pmic_pf100.c.

17.4.3 Menu Configuration Options

The following are menu configuration options:

1. When located in the `<ltib dir>`, to get to the PMIC power configuration, use the command:

   ```
   ./ltib -c
   ```

2. On the configuration screen select Configure Kernel, and then exit. When the next screen appears, choose the following:

   Device Drivers > Voltage and Current regulator support > Support regulators on Freescale PF100 PMIC.

Chapter 18
CPU Frequency Scaling (CPUFREQ) Driver

18.1 Introduction

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 voltage VDDCORE, VDDSOC and VDDPU are changed to the voltage value defined in cpu_op-mx6.c . This method can reduce power consumption (thus saving battery power), because the CPU uses less power as the clock speed is reduced.

18.1.1 Software Operation

The CPUFREQ device driver is designed to change the CPU frequency and voltage on the fly.

If the frequency is not defined in cpu_op-mx6.c, the CPUFREQ driver changes the CPU frequency to the nearest higher frequency in the array. The frequencies are manipulated using the clock framework API, while the voltage is set using the regulators API. The CPU frequencies in the array are based on the boot CPU frequency which can be changed by using the clock command in U-Boot. Interactive CPU frequency governor is used and it cannot be changed manually. To change CPU frequency manually, you can use the userspace CPU frequency governor.

Refer to the API document for more information on the functions implemented in the driver.

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 792 MHz), use this command:

```
echo 792000 > /sys/devices/system/cpu/cpu0/cpufreq/scaling_setspeed
```

The frequency 792000 is in KHz, which is 792 MHz.

The maximum frequency can be checked by 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 view available governors:

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

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

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

18.1.2 Source Code Structure

Table below shows the source files and headers available in the following directory.

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

Table 18-1. CPUFREQ Driver Files

File	Description
cpufreq.c	CPUFREQ functions

For CPU frequency working point settings, see arch/arm/mach-mx6/cpu_op-mx6.c.

18.2 Menu Configuration Options

The following Linux kernel configuration is provided for this module:

CONFIG_CPU_FREQ: In menu configuration, 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 (interactive)
- Performance governor
- Powersave governor
- Userspace governor for userspace frequency scaling
- Interactive CPU frequency policy governor
- Conservative CPU frequency governor
- CPU frequency driver for i.MX CPUs

18.2.1 Board Configuration Options

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

Chapter 19
Dynamic Voltage Frequency Scaling (DVFS) Driver

19.1 Introduction

In order to improve power consumption, the Bus Frequency driver dynamically manages the various system frequencies.

The frequency changes are transparent to the higher layers and require no intervention from the drivers or middleware. Depending on activity of the peripheral devices and CPU loading, the bus frequency driver varies the DDR frequency between 24MHz and its maximum frequency. Similarly the AHB frequency is varied between 24MHz and 132MHz.

19.1.1 Operation

The Bus Frequency driver is part of the power management module in the Linux BSP. The main purpose of this driver is to scale the various operating frequency of the system clocks (like AHB, DDR, AXI etc) based on peripheral activity and CPU loading.

19.1.2 Software Operation

The bus frequency depends on the usecount of the various clocks in the system for its operation. Drivers enable/disable their clocks based on peripheral activity. Every peripheral is associated with a frequency setpoint. The bus frequency will set the system frequency to highest frequency setpoint based on the peripherals that are currently active.

The following setpoints are defined for all i.MX 6 platforms:

1. High Frequency Setpoint: AHB is at 132MHz, AXI is at 264Mhz and DDR is at the maximum frequency. This mode is used when most periphrals that need higher frequency for good performance are active. For ex, video playback, graphics processing etc.

2. Audio Playback setpoints : AHB is at 25MHz, AXI is at 50MHz and DDR is at 50MHz. This mode is used in audio playback mode.

3. Low Frequency setpoint: AHB is at 24MHz, AXI is at 24MHz and DDR is at 24MHz. This mode is used when the system is idle waiting for user input (display is off).

To Enable the bus frequency driver, use the following command:

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

To Disable the bus frequency driver, use the following command:

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

19.1.3 Source Code Structure

Table below lists the source files and headers available in the following directory:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6
```

Table 19-1. BusFrequency Driver Files

File	Description
bus_freq.c	Bus Frequency functions
mx6_mmdc.c, mx6_ddr_freq.S	DDR frequency change functions

19.2 Menu Configuration Options

There are no menu configuration options for this driver. The Bus Frequency drivers is included and enabled by default.

19.2.1 Board Configuration Options

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

Chapter 20
Thermal Driver

20.1 Introduction

Thermal driver is a necessary driver for monitoring and protecting the SoC. The thermal driver will monitor the SoC temperature in a certain frequency.

It defines three trip points: critical, hot, and active. Cooling device will take actions to protect the SoC according to different trip points that SoC has reached:

- When reaching critical point, cooling device will reset the system.
- When reaching hot point, cooling device will lower CPU frequency and notify GPU to run at a lower frequency.
- When the temperature drops to below active point, cooling device will release all the cooling actions.

Thermal driver has two parts:

- Thermal zone defines trip points and monitors the SoC's temperature.
- Cooling device takes the actions according to different trip points.

20.1.1 Thermal Driver Overview

The thermal driver implements the SoC temperature monitoring function and protection. It creates a system file interface of /sys/class/thermal/thermal_zone0/ for user. Internally, the thermal driver will monitor the SoC temperature and do necessary protection according to different trip points that SoC's temperature reaches.

20.2 Hardware Operation

The thermal driver uses an internal thermal sensor to monitor the SoC temperature. The cooling device uses the CPU frequency to protect the SoC.

All the related modules are in SoC.

20.2.1 Thermal Driver Software Operation

The thermal driver registers a thermal zone and a cooling device. The structure `thermal_zone_device_ops` describes the necessary interface that the thermal framework needs. The framework will call the related thermal zone interface to monitor the SoC temperature and do the cooling protection.

20.3 Driver Features

The thermal driver supports the following features:

- Thermal zone monitors the SoC temperature.
- Cooling device protects the SoC when the temperature reaches hot or critical points.

20.3.1 Source Code Structure

Table below shows the driver source files available in the directory:

<ltib_dir>/rpm/BUILD/linux/drivers/mxc/thermal

Table 20-1. Thermal Driver Files

File	Description
thermal.c	thermal zone driver source file
cooling.c	cooling device source file

20.3.2 Menu Configuration Options

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

Device Drivers > MXC support drivers > ANATOP_THERMAL > Thermal Zone

20.3.3 Programming Interface

The thermal driver can be accessed through /sys/class/thermal/thermal_zone/.

20.3.4 Interrupt Requirements

The thermal driver uses irq #81. Set the alarm value to critical trip point. When the temperature exceeds the critical trip point, the interrupt handler will reset the system to protect SoC.

20.4 Unit Test

Modify the trip point's temperature through /sys/class/thermal/thermal_zone0/ trip_point_x_temp. Here, 'x' can be 0, 1 and 2, indicating critical, hot and active trip point, and the value of trip points should be critical > hot > active. Then run some program to make SoC in heavy loading. When the SoC temperature reaches the trip points, the thermal driver will take action to do some protections according to each trip point's mechanism. Restore the trip point's temperature. When SoC temperature drops to below active trip point, thermal driver will remove all the protections.

Chapter 21
Anatop Regulator Driver

21.1 Introduction

The Anatop regulator driver provides the low-level control of the power supply regulators, and selection of voltage levels.

This device driver makes use of the regulator core driver to access the Anatop hardware control registers.

21.1.1 Hardware Operation

The Power Management Unit on the die is built to simplify the external power interface and allow the die to be configured in a power appropriate manner. The power system consists of the input power sources and their characteristics, the integrated power transforming and controlling elements, and the final load interconnection and requirements.

Using seven LDO regulators, the number of external supplies is greatly reduced. If the backup coin and USB inputs are neglected, the number of external supplies is reduced to two. Missing from this external supply total are the necessary external supplies to power the desired memory interface. This will change depending on the type of external memory selected. Other supplies might also be necessary to supply the voltage to the different I/O power segments if their I/O voltage needs to be different than what is provided above.

Some internal regulators can be bypassed, so that external PMIC can supply these power directly to decrease power numer, such as VDD_SOC and VDD_ARM.

21.2 Driver Features

The Anatop regulator driver is based on regulator core driver. The following services are provided for regulator control:

- Switching ON/OFF all voltage regulators.
- Setting the value for all voltage regulators.
- Getting the current value for all voltage regulators.

21.2.1 Software Operation

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

21.2.2 Regulator APIs

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

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

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

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

    - `int regulator_set_voltage(struct regulator *regulator, int uV);`
- `regulator_get_voltage` used to get regulator output voltage:
    - `int regulator_get_voltage(struct regulator *regulator);`

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

21.2.3 Driver Interface Details

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

- Seven LDO regulators.
- All of the regulator functions are handled by setting the appropriate Anatop hardware register values. This is done by calling the regulator core APIs to access the Anatop hardware registers.

21.2.4 Source Code Structure

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

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

Table 21-1. Anatop Power Management Driver Files

File	Description
`core.c`	Linux kernel interface for regulators.
`anatop-regulator.c`	Implementation of the Anatop regulator client driver

The Anatop regulators for i.MX 6Quad ARM2 or i.MX 6Quad sabrelite board are registered under

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/mx6_anatop_regulator.c.`

21.2.5 Menu Configuration Options

To get to the Anatop regulator configuration, use the command ./ltib -c when located in the <ltib dir>. On the configuration screen, select Configure Kernel, and then exit. The following Linux kernel configurations are provided for the Anatop Regulator driver:

- Device Drivers > Voltage and Current regulator support > Anatop Regulator Support.
- System Type > Freescale MXC Implementations > Internal LDO in i.MX 6Quad and i.MX 6DualLite bypass.

Chapter 22
SNVS Real Time Clock (SRTC) Driver

22.1 Introduction

The SNVS Real Time Clock (SRTC) module is used to keep the time and date. It provides a certifiable time to the user and can raise an alarm if tampering with counters is detected. The SRTC is composed of two sub-modules: Low power domain (LP) and High power domain (HP). The SRTC driver only supports the LP domain with low security mode.

22.1.1 Hardware Operation

The SRTC is a real-time clock with enhanced security capabilities.

It provides an accurate and constant time, regardless of the main system power state and without the need to use an external on-board time source, such as an external RTC. The SRTC can wake up the system when a preset alarm threshold is reached.

22.2 Software Operation

The following sections describe the software operation of the SRTC driver.

22.2.1 IOCTL

The SRTC driver complies with the Linux RTC driver model. See the Linux documentation in <ltib_dir>/rpm/BUILD/linux/Documentation/rtc.txt for information on the RTC API.

Besides the initialization function, the SRTC driver provides IOCTL functions to set up the RTC timers and alarm functions. The following RTC IOCTLs are implemented by the SRTC driver:

- RTC_RD_TIME
- RTC_SET_TIME
- RTC_AIE_ON
- RTC_AIE_OFF
- RTC_ALM_READ
- RTC_ALM_SET

The driver information can be access by the proc file system. For example:

```
root@freescale /unit_tests$ cat /proc/driver/rtc
rtc_time        : 12:48:29
rtc_date        : 2009-08-07
alrm_time       : 14:41:16
alrm_date       : 1970-01-13
alarm_IRQ       : no
alrm_pending    : no
24hr            : yes
```

22.2.2 Keeping Alive in the Power Off State

To preserve the time when the device is in the power-off state, the SRTC clock source should be set to CKIL and the voltage input, NVCC_SRTC_POW, should remain active. Usually these signals are connected to the PMIC and software can configure the PMIC registers to enable the SRTC clock source and power supply.

Generally, when the main battery is removed and the device is in power-off state, a coin-cell battery is used as a backup power supply. To avoid SRTC time loss, the voltage of the coin-cell battery should be sufficient to power the SRTC. If the coin-cell battery is chargeable, it is recommended to automatically enable the coin-cell charger so that the SRTC is properly powered.

22.3 Driver Features

The SRTC driver includes the following features:

- Implementing all the functions required by Linux to provide the real-time clock and alarm interrupt.
- Reserveing time in power-ff state.
- Alarm wakes up the system from low-power modes.

22.3.1 Source Code Structure

The RTC module is implemented in the following directory:

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

Table below shows the RTC module files.

Table 22-1. RTC Driver Files

File	Description
rtc-snvs.c	SNVS RTC driver implementation file

The source file for the SRTC specifies the SRTC function implementations.

22.3.2 Menu Configuration Options

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

- Device Drivers > Real Time Clock > Freescale SNVS Real Time Clock

Chapter 23
Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound Driver

23.1 ALSA Sound Driver Introduction

The Advanced Linux Sound Architecture (ALSA), now the most popular architecture in Linux system, provides audio and MIDI functionality to the Linux operating system.

ALSA has the following significant features:

- Efficient support for all types of audio interfaces, from consumer sound cards to professional multichannel audio interfaces
- Fully modularized sound drivers
- SMP and thread-safe design
- User space library (alsa-lib) to simplify application programming and provide higher level functionality
- Support for the older Open Sound System (OSS) API, providing binary compatibility for most OSS programs

ALSA System on Chip (ASoC) layer is designed for SoC audio. The overall project goal of the ASoC layer provides better ALSA support for embedded system on chip processors and portable audio CODECs.

The ASoC layer also provides the following features:

- CODEC independence, allows reuse of CODEC drivers on other platforms and machines.
- Easy I2S/PCM audio interface setup between CODEC and SoC. Each SoC interface and CODEC registers its audio interface capabilities with the core.
- Dynamic Audio Power Management (DAPM). DAPM is an ASoC technology designed to minimize audio subsystem power consumption no matter what audio-use case is active. DAPM guarantees the lowest audio power state at all times and is completely transparent to user space audio components. DAPM is ideal for mobile devices or devices with complex audio requirements.

- Pop and click reduction. Pops and clicks can be reduced by powering the CODEC up/down in the correct sequence (including using digital mute). ASoC signals the CODEC when to change power states.
- Machine specific controls, allows machines to add controls to the sound card, for example, volume control for speaker amp.

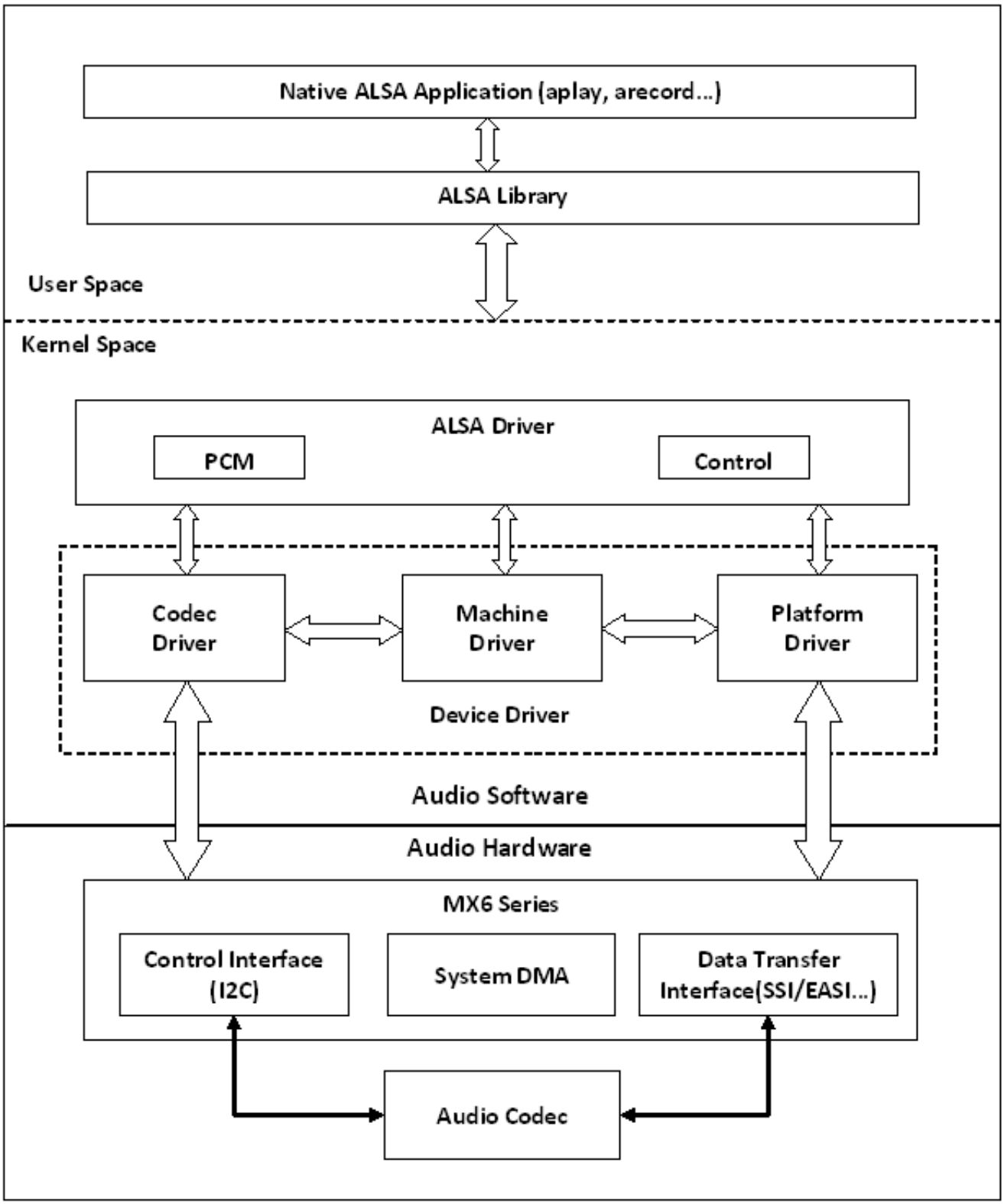


Figure 23-1. ALSA SoC Software Architecture

ASoC basically splits an embedded audio system into 3 components:

- 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.

More detailed information about ASoC can be found in the Linux kernel documentation in the linux source tree at linux/Documentation/sound/alsa/soc and at http://www.alsa-project.org/main/index.php/ASoC.

23.2 SoC Sound Card

Currently, the stereo CODEC (wm8962), 7.1 CODEC (cs42888), and AM/FM CODEC drivers are implemented by using SoC architecture.

These sound card drivers are built in independently. The stereo sound card supports stereo playback and capture. The 7.1 sound card supports up to eight channels of audio playback, while enabling ASRC, 7.1 sound card only supports 2 or 6 channels audio playback. The AM/FM sound card supports radio PCM capture.

NOTE

The 7.1 CODEC is only supported on the i.MX 6Quad Sabre-AI platform.

The AM/FM CODEC is only supported on the i.MX 6Quad Sabre-AI platform.

23.2.1 Stereo CODEC Features

The stereo CODEC supports the following features:

- Sample rates for playback and capture are 8KHz, 32 KHz, 44.1 KHz, 48 KHz, and 96 KHz
- Channels:
  - Playback: supports two channels.
  - Capture: supports two channels.
- 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

23.2.2 7.1 Audio Codec Features

- Sample rates for playback and record:
    - 48 KHz, 96 KHz, 192 KHz
    - Playback: 5.512k, 8k, 11.025k, 16k, 22k, 32k, 44.1k, 48k, 64k, 88.2k, 96k, 176.4k, 192k(ASRC enabled)
- Channels:
    - Playback: 2, 4, 6, 8 channels
    - Playback(ASRC enabled): 2, 6 channels
    - Capture: 2, 4 channels
- Audio formats:
    - Playback:
        - SNDRV_PCM_FMTBIT_S16_LE
        - SNDRV_PCM_FMTBIT_S20_3LE
        - SNDRV_PCM_FMTBIT_S24_LE
    - Playback(ASRC enabled):
        - SNDRV_PCM_FMTBIT_S16_LE
        - SNDRV_PCM_FMTBIT_S24_LE
    - Capture:
        - SNDRV_PCM_FMTBIT_S16_LE
        - SNDRV_PCM_FMTBIT_S20_3LE
        - SNDRV_PCM_FMTBIT_S24_LE

23.2.3 AM/FM Codec Features

- Supported sample rate for Capture: 48 KHz
- Supported channels:
    - Capture: supports two channels.
- Supported audio formats:
    - Capture: SNDRV_PCM_FMTBIT_S16_LE

23.2.4 Sound Card Information

The following is the registered sound card information, using the commands aplay -l and arecord -l. For example, the stereo sound card is registered as card 0.

```
root@freescale /$ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: wm8962audio [wm8962-audio], device 0: HiFi wm8962-0 []
 Subdevices: 1/1
 Subdevice #0: subdevice #0
```

23.3 Hardware Operation

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

23.3.1 Stereo Audio CODEC

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

The WM8962 ASoC CODEC driver exports the audio record/playback/mixer APIs according to the ASoC architecture.

The CODEC driver is generic and hardware independent code that configures the CODEC to provide audio capture and playback. It does not contain 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 WM8962 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 .

Headphone insertion/removal can be detected through a GPIO interrupt signal.

SSI dual FIFO features are enabled by default.

23.3.2 7.1 Audio Codec

The 7.1 audio codec includes 8-channel DAC and 4-channel ADC, which are controlled by the I2C interface. The audio data is transferred from the user data buffer to the ESAI fifo, through a DMA channel. The DMA channel is selected according to audio sample bits. The codec works in slave mode as the esai provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate. The ESAI supports up to eight audio output ports. While enabling ASRC, 7.1 audio codec supports 2-channel or 6-channel playback through ASRC. On the i.MX 6 Sabre-AI board, a cs42888 codec with 4 audio in port is used, each port receive two channels of data in the I2S format (network mode), providing 8-channel of playback functionality. This codec also has two audio output ports connected with ESAI, providing 4-channel of recording functionality.

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

- Codec DAI and PCM configuration
- Codec control I/O, using I2C
- Mixers and audio controls
- Codec audio operations
- DAI Digital mute control

The CS42888 codec is registered as an I2C client when the module initializes. The APIs are exported to the upper layer by the structure snd_soc_dai_ops.

23.3.3 AM/FM Codec

The AM/FM codec is a virtual codec, it only has a PCM interface connected to the Tuner device. The audio data is transferred from the user data buffer to or from the SSI FIFO through the DMA channel. The DMA channel is selected according to the audio sample bits. AUDMUX is used to set up the path between the SSI port and the output port which connects with the codec. The codec works in master mode as it provides the BCLK and LRCLK. The BCLK and LRCLK can be configured according to the audio sample rate.

23.4 Software Operation

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

23.4.1 ASoC Driver Source Architecture

File imx-pcm-dma-mx2.c is shared by the stereo ALSA SoC driver, the 7.1 ALSA SoC driver and other 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. wm8962.c registers the stereo CODEC and hifi DAI drivers. The direct hardware operations on the stereo codec are in wm8962.c. imx-wm8962.c is the machine layer code which creates the driver device and registers the stereo sound card.

The multi-channel codec is connected to the CPU through the ESAI interface. imx-esai.c registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip ESAI interface. cs42888.c registers the multi-channel CODEC and hifi DAI drivers. The direct hardware operations on the multi-channel CODEC are in cs42888.c. imx-cs42888.c is the machine layer code which creates the driver device and registers the stereo sound card.

The AM/FM 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. si4763.c registers the Tuner CODEC and Tuner DAI drivers. The direct hardware operations on the CODEC are in si4763.c. imx-si4763.c is the machine layer code which creates the driver device and registers the sound card.

The following table shows the stereo CODEC SoC driver source files. These files are under the <ltib_dir>/rpm/BUILD/linux/sound/soc directory.

Table 23-1. Stereo Codec SoC Driver Files

File	Description
imx/imx-wm8962.c	Machine layer for stereo CODEC ALSA SoC
imx/imx-pcm-dma-mx2.c	Platform layer for stereo CODEC ALSA SoC
imx/imx-pcm.h	Header file for PCM driver and AUDMUX register definitions
imx/imx-ssi.c	Platform DAI link for stereo CODEC ALSA SoC
imx/imx-ssi.h	Header file for platform DAI link and SSI register definitions
codecs/wm8962.c	CODEC layer for stereo CODEC ALSA SoC
codecs/wm8962.h	Header file for stereo CODEC driver

The following table lists the AM/FM CODEC SoC driver source files. These files are under the <ltib_dir>/rpm/BUILD/linux/sound/soc directory.

Table 23-2. AM/FM Codec SoC Driver Source Files

File	Description
imx/imx-si4763.c	Machine layer for AM/FM CODEC ALSA SoC
imx/imx-si4763.h	Header file for AM/FM CODEC ALSA SoC
imx/imx-pcm-dma-mx2.c	Platform layer for stereo CODEC ALSA SoC
imx/imx-pcm.h	Header file for pcm driver and AUDMUX register definitions
imx/imx-ssi.c	Platform DAI link for stereo CODEC ALSA SoC
imx/imx-ssi.h	Header file for platform DAI link and SSI register definitions
codecs/si4763.c	Codec layer for stereo CODEC ALSA SoC

The following table shows the multiple-channel ADC SoC driver source files. These files are also under the <ltib_dir>/rpm/BUILD/linux/sound/soc directory.

Table 23-3. CS42888 ASoC Driver Source File

File	Description
imx/imx-cs42888.c	Machine layer for mutliple-channel CODEC ALSA SoC
imx/imx-pcm-dma-mx2.c	Platform layer for mutliple-channel CODEC ALSA SoC
imx/imx-pcm.h	Header file for pcm driver
imx/imx-esai.c	Platform DAI link for mutliple-channel CODEC ALSA SoC
imx/imx-esai.h	Header file for platform DAI link
codecs/cs42888.c	CODEC layer for mutliple-channel codec ALSA SoC
codecs/cs42888.h	Header file for mutliple-channel CODEC driver

23.4.2 Sound Card Registration

The CODECs have the same registration sequence:

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

23.4.3 Device Open

The ALSA driver performs the following functions:

- 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.

23.4.4 Platform Data

struct mxc_audio_platform_data defined in include/linux/fsl_devices.h is used to pass the platform data of audio CODEC.

The value of platform data needs to be updated according to Hardware design.

Take wm8962 CODEC platform data as a example to show the parameter of mxc_audio_platform_data. See header file for the details of more variables.

- `ssi_num` indicates which SSI channel is used.
- `src_port` indicates which AUDMUX port is connected with SSI.
- `ext_port` indicates which AUDMUX port is connected with external audio CODEC.
- `hp_gpio`: The IRQ line used for headphone detection.
- `hp_active_low`: When headphone is inserted, the detection pin status. If pin voltage level is low, the value should be 1.
- `mic_gpio`: The IRQ line used for micphone detection
- `mic_active_low`: When micphone is inserted, the detection pin status, if pin voltage level is low, the value should be 1.
- `init`: The callback function to initialize audio CODEC. For example, configure the clock of audio CODEC.
- `clock_enable`: The callback function to enable or disable clock for audio CODEC.

23.4.5 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>. Select **Configure the Kernel** on the displayed screen and exit. When the next screen appears, select the following options to enable this module:

- SoC Audio supports for wm8962 CODEC. In menu configuration, this is 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, SoC Audio support for WM8962
- SoC Audio supports for i.MX cs42888. In menu configuration, this is option is available under Device drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio support for IMX - CS42888
- SoC Audio supports for AM/FM. In menu configuration, this is 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, SoC Audio support for IMX SI4763

23.5 Unit Test

This section shows how to use ALSA driver, and assume the rootfs is GNOME.

23.5.1 Stereo CODEC Unit Test

Stereo CODEC driver supports playback and record features. There are a default volume, and you may adjust volume by alsamixer command.

Playback feature may be tested by the following command:

- aplay [-Dplughw:0,0] audio.wav

Record feature supports analog micphone and digital micphone. The default is digital micphone if analog micphone isn't plug-in.

Because analog micphone is connected to IN3R port of WM8962 CODEC, the following amixer commands are needed to input into command line for enabling analog micphone.

- amixer sset 'MIXINR IN3R' on
- amixer sset 'INPGAR IN3R' on

The recording feature may be tested by the following command:

- arecord [-Dplughw:0,0] -r 44100 -f S16_LE -c 2 -d 5 record.wav

More usage for aplay/arecord/amixer may be obtained by the following commands.

- aplay --h
- arecord --h
- amixer --h

23.5.2 7.1 Audio Codec Unit Test

The 7.1 Audio codec driver supports multi-channel playback and record feature. The codec has a default volume, and you can adjust volume by alsamixer command.

The playback feature can be tested by the following command:

- aplay [-Dplughw:0,0] audio.wav

While enabling ASRC, the 7.1 audio codec should use the device 1 for playback. The codec has a default volume, and you can adjust volume by alsamixer command.

- aplay [-Dplughw:0,1] audio.wav

The recording feature supports line in and mic in simultaneously. While on i.MX 6 Sabre-AI board, LINE-IN (L/R) uses AIN1/AIN2, and MICS1/MICS2 uses AIN3/AIN4. By default, 2-ch record uses AIN1/AIN2, and 4-ch record uses AIN1/AIN2/AIN3/AIN4 together.

The recording feature can be tested by following command:

- arecord [-Dplughw:0,0] -r 48000 -f S16_LE -c 2 -d 5 record.wav

Note:The default ALSA config file, asound.conf located under /etc/, only supports stereo playback and record, which means, if you want to test 4,6,8-ch playback or 4-ch recording, and use aplay audio.wav or arecord -c 4 audio.wav(without -Dplughw), you will have to make slight changes to the configure file as following:

- Make sure that playback PCM uses dmix_48000 and capture PCM uses dsnoop_48000 under pcm.asymed{}.
- Add "channels x" to the end of struct pcm.dmix_48000{} if you want to playback x-ch wav file(x is greater than 2).
- Add "channels x" to the end of struct pcm.!dsnoop_48000{} if you want to record to x-ch wav(x is greater than 2).

If plug plughw is used to make a playback or record, examples are as follows:

- aplay: Dplughw:0,0 audio.wav or
- arecord: Dplughw:0,0 -c 4 -r 48000 -f S16_LE record.wav

You are not required to change asound.conf because this configuration file is not used here.

More usage for aplay/arecord/amixer can be obtained by the following commands.

- aplay --h
- arecord --h
- amixer --h

23.5.3 AM/FM Codec Unit Test

This test turns on the AM/FM radio tuner (SI4763). It also sets and gets the current station.

NOTE: An underrun error may occur sometimes.

This underrun behaviour is normal, since the test connects the AM/FM output to the audio codec by a simple pipe.

There is no synchronization method between them. Upper layers (such as gstreamer plugins) should be responsible for synchronization.

Input the following command in command line to start unit test:

- ./mxc_tuner_test.sh

The following infomation will be output to console window:

Welcome to radio menu.

1. Turn on the radio

2. Get current frequency

3. Set current frequency

4. Turn off the radio

9. Exit.

- To turn on the radio, select option 1.
- To get the current frequency, select option 2.
- To set the desire frecuency, select option 3 <enter> set the frequency <9740>.
- To turn off the radio, select option 4.
- To Exit select, option 9.

Chapter 24
Asynchronous Sample Rate Converter (ASRC) Driver

24.1 Introduction

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

24.1.1 Hardware Operation

ASRC includes the following features:

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

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

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

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

24.2 Software Operation

As an assistant component in the audio system, the ASRC driver implementation depends on the use cases in the platform.

Currently ASRC is used in following two scenarios:

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

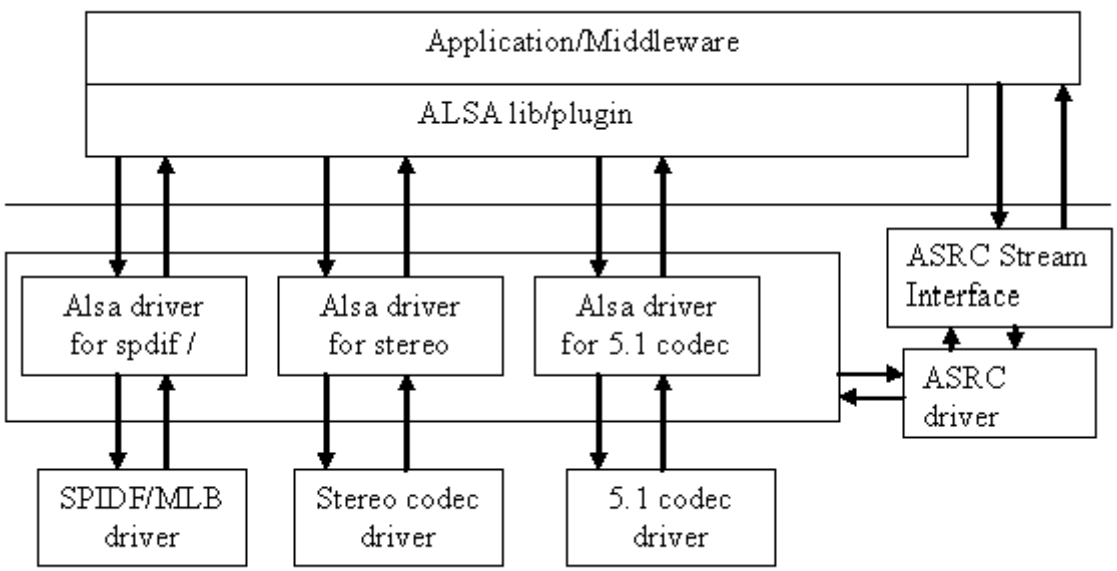


Figure 24-1. Audio Driver Interactions

As illustrated in figure above, the ASRC stream interface provides the interface for the user space. The ASRC registers itself under /dev/mxc_asrc and creates proc file /proc/driver/asrc when the module is inserted. proc is used to track the channel number for each

pair. If all the pairs are not used, users can adjust the channel number through the proc file. The total channels number should be 10. Otherwise, the adjusted value cannot be saved properly.

Now 7.1 audio codec driver supports calling ASRC driver for memroy > ASRC > perripheral(ESAI TX). The input audio file is convert into board defined sampling rate(for example, 48khz). This use case only supports 2-channel or 6-channel playback. To call this use case, perform the the following steps::

- Call aplay -l | grep ASRC to get the card number and device number of playback PCM. The device name is CS42888_ASRC. For example, the card number is 0 and the device number is 1.
- Play the audio file with the card0device1 device. For example, aplay -Dplughw:0,1 $AUDIO_FILE.

24.2.1 Sequence for Memory to ASRC to Memory

- Start the /dev/mxc_asrc device.
- Request ASRC pair. (ASRC_REQ_PAIR)
- Configure ASRC pair. (ASRC_CONIFG_PAIR)
- Start ASRC. (ASRC_START_CONV)
- Write the raw audio data (to be converted) into the user maintained input buffer. Fill asrc_convert_buffer struct with input/output buffer length and address. Driver would copy output data to user maintained output buffer address according to the output buffer size. Repeat this step until all data is converted. (ASRC_CONVERT)
- Stop ASRC conversion: (ASRC_STOP_CONV)
- Release ASRC pair. (ASRC_RELEASE_PAIR)
- Shut down the /dev/mxc_asrc device.

24.2.2 Sequence for Memory to ASRC to Peripheral

Memory to ASRC to peripheral audio path is involved in 7.1 audio codec driver. In 7.1 audio sound card, a new device with the name of CS42888_ASRC is specified for playback with ASRC. The steps below show the flow of calling ASRC to memroy to peripheral:

- The sound device (PCM) has been registered and start to enable the DMA channel in ALSA driver.
- Request ASRC pair. (asrc_req_pair)
- Configure ASRC pair. (asrc_config_pair)

- Enable the DMA channel from Memory to ASRC and from ASRC to Memory.
- Start DMA channel and start ASRC conversion. (asrc_start_conv)
- When audio data playback complete, stop DMA channel and ASRC. (asrc_stop_conv)
- Release ASRC pair. (asrc_release_pair)

24.3 Source Code Structure

Table below lists the source files available in the devices directory.

```
<ltib_dir>/rpm/BUILD/linux/drivers/mxc/asrc
<ltib_dir>/rpm/BUILD/linux/include/linux/
<ltib_dir>/rpm/BUILD/linux/sound/soc/imx/
<ltib_dir>/rpm/BUILD/linux/sound/soc/codec/
```

Table 24-1. ASRC Source File List

File	Description
mxc_asrc.c	ASRC driver implementation codes including stream interface
mxc_asrc.h	ASRC register definitions and export function declarations
imx-cs42888.c	memory to ASRC to ESAI TX implementation in 7.1 audio codec machine driver.
imx-pcm-dma-mx2.c	memroy to ASRC to ESAI TX implementation in 7.1 audio codec platform driver.
imx-esai.c	memroy to ASRC to ESAI TX implementation in 7.1 audio codec cpu driver.
cs42888.c	memory to ASRC to ESAI TX implementation in 7.1 audio codec codec driver.

24.3.1 Linux Menu Configuration Options

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

The ASRC driver can only be configured with build-in module.

24.4 Platform Data

struct mxc_asrc_platform_data defined in arch/arm/plat-mxc/include/mach/mxc_asrc.h is used to transfer the platform information of ASRC according to different SOC.

- channel_bits: indicates the channel bit information.
- clk_map_ver: The mapping relationships in different SOC are different. This version number can be used to indicate clock map information.

- asrc_core_clk: ASRC core clock information, which is used for ASRC register access.
- asrc_audio_clk: ASRC process clock, which is used for input/output clock source.

24.4.1 Programming Interface (Exported API and IOCTLs)

The ASRC Exported API allows the ALSA driver to use ASRC services.

The ASRC IOCTLs below are used for user space applications:

ASRC_REQ_PAIR:

Apply a pair from ASRC driver. Once a pair is allocated, ASRC core clock is enabled.

ASRC_CONFIG_PAIR:

Configure ASRC pair allocated. User is responsible for providing parameters defined in struct asrc_config. Items in asrc_config are as follows:

- pair: ASRC pair allocated by the IOCTL(ASRC_REQ_PAIR).
- channel_num: channel number.
- buffer_num: buffer number required by input and output buffer. The input/output buffers are allocated inside ASRC driver. The user is responsible for remapping it into user space.
- dma_buffer_size: buffer size for input and output buffers. The buffer size should be in the unit of page size. Usually, 4 KB is used.
- input_sample_rate: input sampling rate. Input sample rate should be 5.512k, 8k, 11.025k, 16k, 22k, 32k, 44.1k, 48k, 64k, 88.2k 96k, 176.4k, or 192k.
- output_sample_rate: output sampling rate. Output sampling rate should be 32k, 44.1k, 48k, 64k, 88.2k, 96k, 176.4k, or 192k.
- input_word_width: word width of input audio data. The input data word width can be 16 bit or 24 bit.
- output_word_width: word width of output audio data. The output data word width can be 16 bit or 24 bit.
- inclk: the input clock source can be ESAI RX clock, SSI1 RX clock, SSI2 RX clock, SPDIF RX clock, MLB_clock, ESAI TX clock, SSI1 TX clock, SSI2 TX clock, SPDIF TX clock, ASRCLK1 clock, or NONE. If using clock except NONE, the user should make sure that the clock is available.
- outclk: the output clock source is the same as the input clock source.

ASRC_CONVERT:

Convert the input data into output data according to the parameters set by ASRC_CONFIG_PAIR. Driver would copy input_buffer_length bytes data from the input_buffer_vaddr for conversion. After conversion, the driver fills the

output_buffer_length according to data number generated by ASRC and copy output_buffer_length to output_buffer_vaddr. However, before calling ASRC_CONVERT, the user needs to fill the output_buffer_length according to the ratio of input sample rate and output sample rate. If the generated buffer size is larger than the user filled output_buffer_size, the driver would only copy user filled output_buffer_size to output_buffer_vaddr. If the generated buffer size is smaller than user filled output_buffer_size (with the difference of less than 64 bytes), calling ASRC_CONVERT would fail.

- input_buffer_vaddr: virtual address of input buffer.
- output_buffer_vaddr: virtual address of output buffer.
- input_buffer_length: length of input buffer(bytes).
- output_buffer_length: length of output buffer(bytes).

ASRC_START_CONV:

Start ASRC pair convert.

ASRC_STOP_CONV:

Stop ASRC pair convert.

ASRC_STATUS:

Query ASRC pair status.

Chapter 25
The Sony/Philips Digital Interface (S/PDIF) Driver

25.1 Introduction

The Sony/Philips Digital Interface (S/PDIF) audio module is a stereo transceiver that allows the processor to receive and transmit digital audio. The S/PDIF transceiver allows the handling of both S/PDIF channel status (CS) and User (U) data. The frequency measurement block allows the S/PDIF RX section to derive the receive clock from the incoming S/PDIF stream.

25.1.1 S/PDIF Overview

The following figure shows the block diagram of the S/PDIF interface.

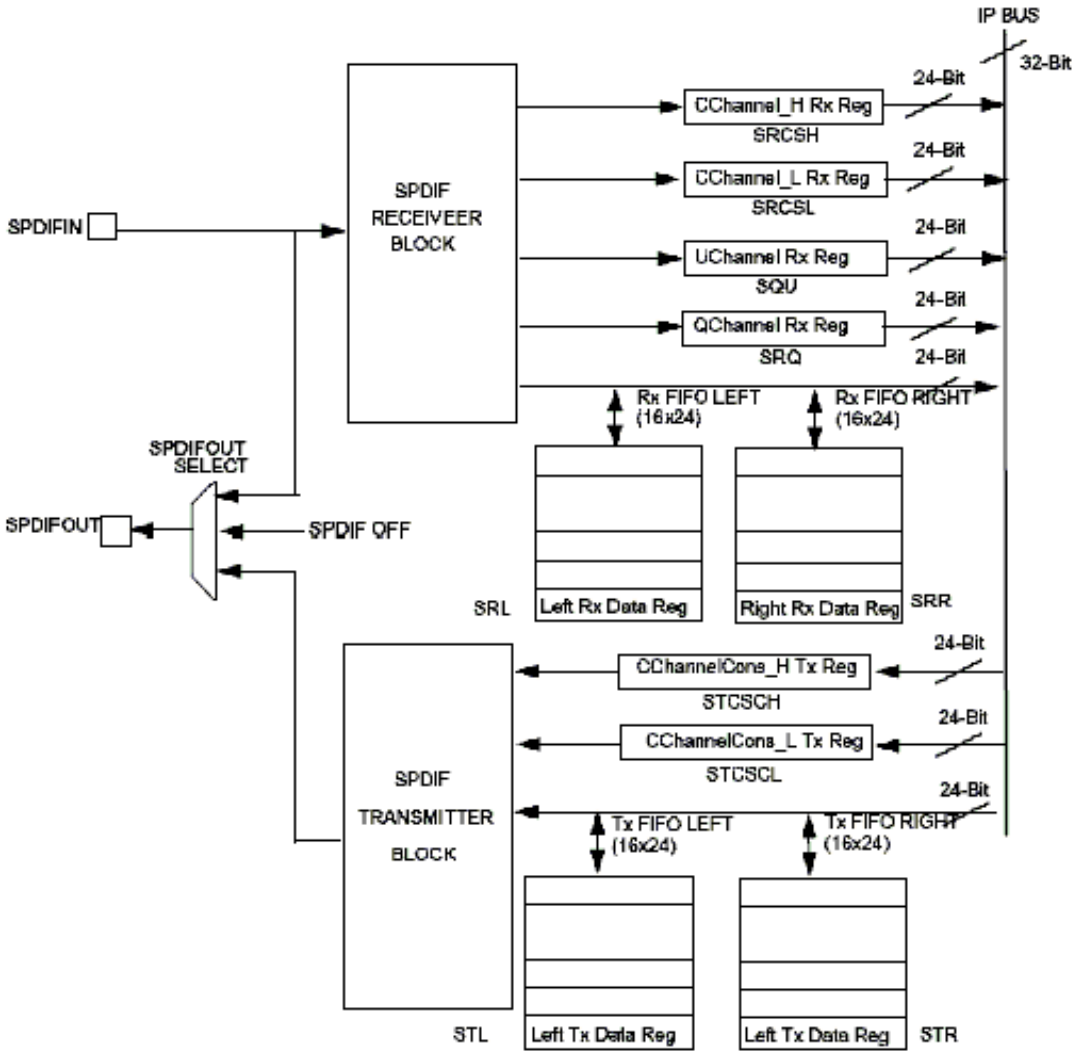


Figure 25-1. S/PDIF Transceiver Data Interface Block Diagram

25.1.2 Hardware Overview

The S/PDIF is composed of two parts:

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

output bitstream in the biphase mark format (IEC958), which consists of audio data, channel status and user bits.

In the S/PDIF transmitter, the IEC958 biphase bit stream is generated on both edges of the S/PDIF Transmit clock. The S/PDIF Transmit clock is generated by the S/PDIF internal clock dividers and the sources are from outside of the S/PDIF block. The S/PDIF receiver can recover the S/PDIF Rx clock from the S/PDIF stream. Figure 25-1 shows the clock structure of the S/PDIF transceiver.

25.1.3 Software Overview

The S/PDIF driver is designed at the ALSA System on Chip (ASoC) layer. The ASoC driver for S/PDIF provides one playback device for Tx and one capture device for Rx. The playback output audio format can be linear PCM data or compressed data with 16-bit, 20-bit, and 24-bit audio. The allowed sampling bit rates are 44.1, 48 and 32 KHz. The capture input audio format can be linear PCM data or compressed 24-bit data and the allowed sampling bit rates are from 16 to 96 KHz. The driver provides the same interface for PCM and compressed data transmission.

25.1.4 ASoC layer

The ASoC layer divides audio drivers for embedded platforms into separated layers that can be reused. ASoC divides an audio driver into a codec driver, a machine layer, a DAI (digital audio interface) layer, and a platform layer. The Linux kernel documentation has some concise description of these layers in linux/Documentation/sound/alsa/soc. In the case of the S/PDIF driver, you can reuse the platform layer (imx-pcm-dma-mx2.c) that is used by the ssi stereo codec driver.

25.2 S/PDIF Tx Driver

The S/PDIF Tx driver supports the following features:

- 32, 44.1 and 48 KHz sample rates.
- Signed 16 and 24-bit little Endian sample format. Due to S/PDIF SDMA feature, the 24-bit output sample file must have 32-bits in each channel per frame. Only the 24 LSBs are valid.
- In the ALSA subsystem, the supported format is defined as S16_LE and S24_LE.
- Two channels.

- Information query.
- The device ID can be determined by using the "aplay -l" utility to list out the playback audio devices.

  For example:

```
root@freescale ~$ aplay -l

**** List of PLAYBACK Hardware Devices ****

card 0: imxspdif [imx-spdif], device 0: IMX SPDIF mxc-spdif-0 []

  Subdevices: 1/1

  Subdevice #0: subdevice #0
```

- **NOTE**

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

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

```
#aplay -Dplughw:0,0 audio.wav
```

NOTE

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

The "iecset" utility provides a common method to set or dump the IEC958 status bits.

```
#ciecset -c 1
```

25.2.1 Driver Design

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

25.2.2 Provided User Interface

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

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

25.3 S/PDIF Rx Driver

The S/PDIF Rx driver supports the following features:

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

  In ALSA subsystem, the supported format is defined to S24_LE.

- Two channels.
- The device ID can be determined by using the arecord -l to list out record devices.

  For example:

  ```
  root@freescale ~$ arecord -l

  **** List of CAPTURE Hardware Devices ****

  card 0: cs42888audio [cs42888-audio], device 0: HiFi CS42888-0 []

    Subdevices: 1/1

    Subdevice #0: subdevice #0

  card 1: imxspdif [imx-spdif], device 0: IMX SPDIF mxc-spdif-0 []

    Subdevices: 1/1

    Subdevice #0: subdevice #0
  ```

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

  ```
  #arecord -Dplughw:1,0" -c 2 -r 44100 -f S24_LE record.wav
  ```

NOTE

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

The "iecset" utility provides a common method to set or dump the IEC958 status bits.

```
#iecset -c 1
```

25.3.1 Driver Design

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

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

For user bits reception, there are two modes for User Channel reception: CD and non-CD. The mode is determined by the USyncMode (bit 1 of CDText_Control register). The user can call the sound control interface to set the mode (see Table 25-1), but no matter what the mode is, the driver handles the user bits in the same way. For the S/PDIF Rx, the hardware block copies the Q bits from the user bits to the QChannel registers and puts the user bits in UChannel registers. The driver allocates two queue buffers for both U bits and Q bits. The U bits queue buffer is 96x2 bytes in size, the Q bits queue buffer is 12x2 bytes in size, and queue buffers are filled in the U/Q Full, Err and Sync interrupt handlers. This means that the user can get the previous ready U/Q bits while S/PDIF driver is reading new U/Q bits.

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

25.3.2 Provided User Interfaces

The S/PDIF Rx driver provides interfaces for user application as shown in table below.

Table 25-1. S/PDIF Rx Driver Interfaces

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

1. The mode column shows the interface attribute: r (read) or w (write)
2. The sound control type of interface is called by the snd_ctl_xxx() alsa-lib function

The user application can follow the program flow from Figure 25-2 to use the S/PDIF Rx driver.

1. The application opens the S/PDIF Rx PCM device, waits for the DPLL to lock the input bit stream, and gets the input sample rate. If the USyncMode needs to be set, set it before reading the U/Q bits.
2. Set the hardware parameters, including the channel number, format and capture sample rate which is obtained from the driver.
3. Call the preparation and triggering function to start S/PDIF Rx stream reading.
4. Call the reading function to get the data. During the reading process, applications can read the U/Q bits and channel status from the driver and validates the illegal bits.

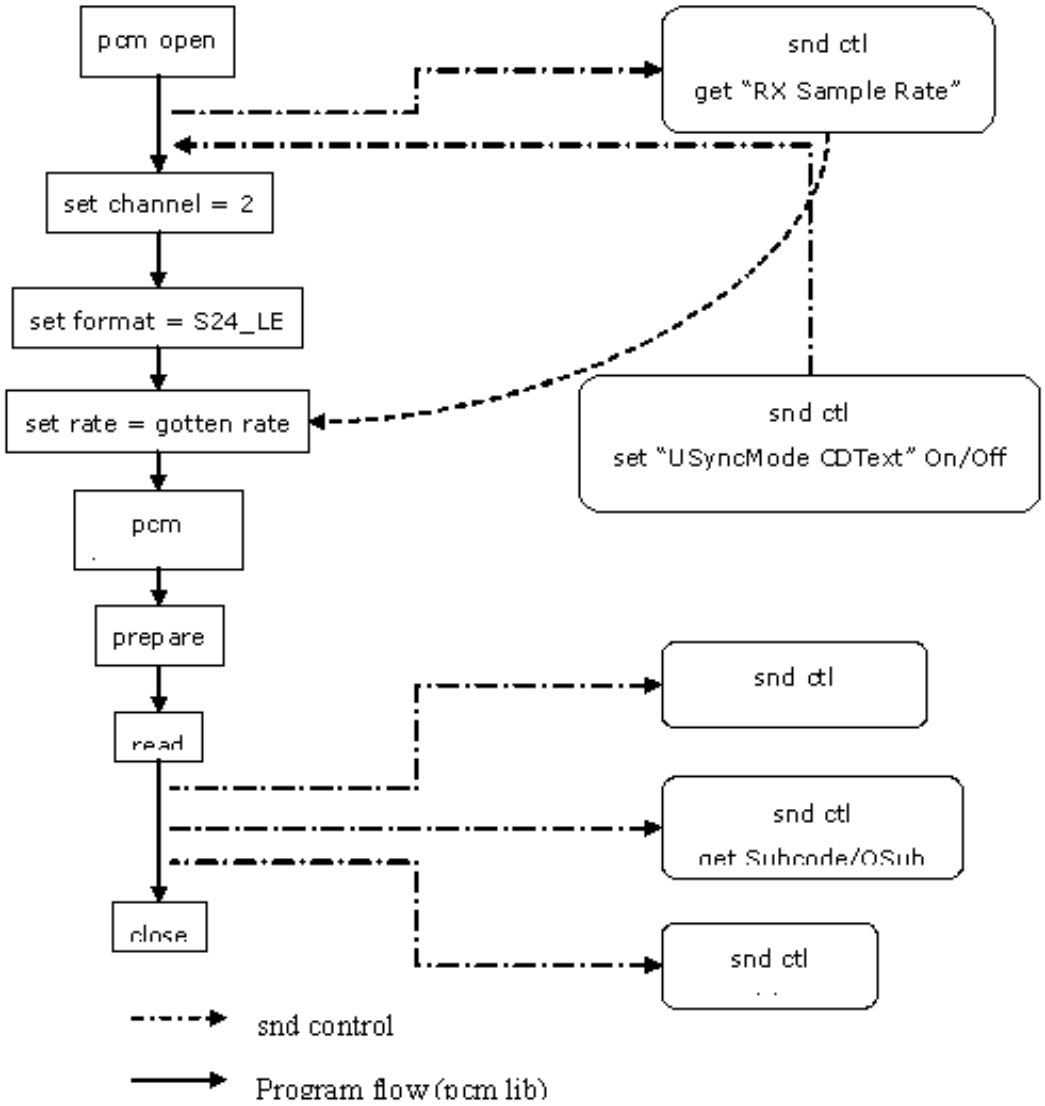


Figure 25-2. S/PDIF Rx Application Program Flow

25.4 Source Code Structure

The following table lists the source files for the driver.

These files are under the <ltib_dir>/rpm/BUILD/linux/ directory.

Table 25-2. S/PDIF Driver Files

File	Description
sound/soc/codecs/mxc_spdif.c	S/PDIF ALSA SOC codec driver
sound/soc/codecs/mxc_spdif.h	S/PDIF ALSA SOC codec driver header
sound/soc/imx/imx-spdif.c	S/PDIF ALSA SOC machine layer

Table continues on the next page...

Table 25-2. S/PDIF Driver Files (continued)

File	Description
sound/soc/imx/imx-spdif-dai.c	S/PDIF ALSA SOC DAI layer
sound/soc/imx/imx-pcm-dma-mx2.c	ALSA SOC platform layer
sound/soc/imx/imx-pcm.h	ALSA SOC platform layer header

25.5 Menu Configuration Options

The following Linux kernel configurations 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 displayed screen and exit. When the next screen appears, select the following options to enable this module:

- CONFIG_SND_MXC_SPDIF: Configuration option for the S/PDIF driver. In the menuconfig, this option is available under Device Drivers > Sound card support > Advanced Linux Sound Architecture > ALSA for SoC audio support > SoC Audio for Freescale i.MX CPUs > SoC Audio support for IMX - S/PDIF

25.6 Platform Data

struct mxc_spdif_platform_data is used to transfer board-specific data to the S/PDIF driver.

It is defined in include/linux/fsl_devices.h.

- spdif_tx : is 1 if TX is supported on the board.
- spdif_rx : is 1 if RX is supported on the board.
- spdif_clk_44100 : the 44.1KHz transmit clock for the STC register. -1 indicates that it does not support this sample rate.
- spdif_clk_48000 : the transmit clock used for 48KHz and 32KHz for the STC register. -1 indicates that it does not support these sample rates.
- spdif_div_44100 : 44.1KHz clock division factor in the STC register.
- spdif_div_48000 : 48KHz clock division factor in the STC register.
- spdif_div_32000 : 32KHz clock division factor in the STC register.
- spdif_rx_clk : rx clock source in mux in SRPC register. Leave as 0 to get clock from rx stream.
- spdif_core_clk : S/PDIF core clock.

25.7 Interrupts and Exceptions

S/PDIF Tx/Rx hardware block has many interrupts to indicate the success, exception and event.

The driver handles the following interrupts:

- DPLL Lock and Loss Lock: saves the DPLL lock status. This is used when getting the Rx sample rate.
- U/Q Channel Full and overrun/underrun: puts the U/Q channel register data into queue buffer, and update the queue buffer write pointer.
- U/Q Channel Sync: saves the ID of the buffer whose U/Q data is ready for read out.
- U/Q Channel Error: resets the U/Q queue buffer.

25.8 Unit Test Preparation

In order to prepare to run a unit test, perform the following actions:

- Set up the M-Audio Transit USB sound card by installing the M-Audio Transit driver on your PC.
- Install WaveLab tools on your PC.

25.8.1 Tx test step

1. Plug optical line into [lineloptical] port of M-Audio transit.

NOTE

Make sure that the [optical out] port of M-Audio transit has no output (red light off) after plugging the optical line.

2. Start WaveLab, press the record button on the toolbar, set the record file name, sample rate, channel number, and then start recording.
3. Run the following command on the board to play one wave file:

```
#aplay -D hw:[card id],[pcm id] audioXXkYYS.wav
```

- After finishing aplay, stop recording in WaveLab.
- Play the recorded wav file in wavelab to check if it works properly.

25.8.2 Rx test step

1. Plug optical line into [optical port] of M-Audio transit.
2. Start WaveLab, open a test wav file: audioXXkYYS.wav to play in loop.
3. Run the following command on the board to record one wave file. After finishing recording, you may play back the recorded wav file on other audio card on the board or PC.

```
#arecord -D hw:[card id],[pcm id] -c 2 -d 20 -r [sample rate in Hz] -f S24_LE record.wav
```

NOTE

The sample rate argument in the arecord command must be consistent with the wav file played on WaveLab.

Chapter 26
SPI NOR Flash Memory Technology Device (MTD) Driver

26.1 Introduction

The SPI NOR Flash Memory Technology Device (MTD) driver provides the support to the data Flash though the SPI interface.

By default, the SPI NOR Flash MTD driver creates static MTD partitions to support data Flash. If RedBoot partitions exist, they have higher priority than static partitions, and the MTD partitions can be created from the RedBoot partitions.

26.1.1 Hardware Operation

On some boards, the SPI NOR - AT45DB321D is equipped, while on some boards M25P32 is equipped. Check the SPI NOR type on the boards and then configure it properly.

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

The M25P32 is a 32 Mbit (4M x 8) Serial Flash memory, with advanced write protection mechanisms, accessed by a high speed SPI-compatible bus up to 75MHz. The memory is organized as 64 sectors, each containing 256 pages. Each page is 256 bytes wide. Thus, the whole memory can be viewed as consisting of 16384 pages, or 4,194,304 bytes. The memory can be programmed 1 to 256 bytes at a time using the Page Program instruction. The whole memory can be erased using the Bulk Erase instruction, or a sector at a time, using the Sector Erase instruction.

Unlike conventional Flash memories that are accessed randomly, these two SPI NOR access data sequentially. They operate from a single 2.7-3.6 V power supply for program and read operations. They are enabled through a chip select pin and accessed through a three-wire interface: Serial Input, Serial Output, and Serial Clock.

26.1.2 Software Operation

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

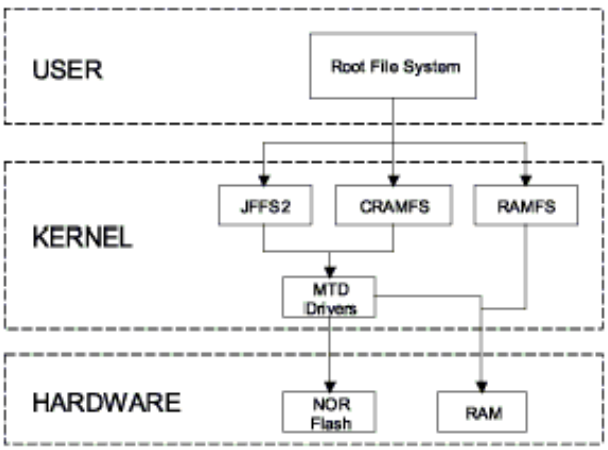


Figure 26-1. Components of a Flash-Based File System

The MTD subsystem for Linux is a generic interface to memory devices, such as Flash and RAM, providing simple read, write, and erase access to physical memory devices. Devices called mtdblock devices can be mounted by JFFS, JFFS2 and CRAMFS file systems. The SPI NOR MTD driver is based on the MTD data Flash driver in the kernel by adding SPI access. In the initialization phase, the SPI NOR MTD driver detects a data Flash by reading the JEDEC ID. Then the driver adds the MTD device. The SPI NOR MTD driver also provides the interfaces to read, write, and erase NOR Flash.

26.1.3 Driver Features

This NOR MTD implementation supports the following features:

- Provides necessary information for the upper layer MTD driver

26.1.4 Source Code Structure

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

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

Table below shows the driver files:

Table 26-1. SPI NOR MTD Driver Files

File	Description
m25p80.c	Source file

26.1.5 Menu Configuration Options

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

- CONFIG_MTD_M25P80: This config enables access to most modern SPI flash chips, used for program and data storage.
- Device Drivers > Memory Technology Device (MTD) support >Self-contained MTD device drivers > Support most SPI Flash chips (AT26DF, M25P, W25X, ...)

Chapter 27
MMC/SD/SDIO Host Driver

27.1 Introduction

The MultiMediaCard (MMC)/ Secure Digital (SD)/ Secure Digital Input Output (SDIO) Host driver implements a standard Linux driver interface to the ultra MMC/SD host controller (uSDHC) .

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 SD3.0 and SDIO 2.0 cards (so far we support SDIO v2.0 (AR6003 is verified)).
- Supports card insertion and removal detections.
- Supports the standard MMC commands.
- PIO and DMA data transfers.
- Power management.
- Supports 1/4/8-bit operations for MMC cards.
- Support eMMC4.4 SDR and DDR modes.
- Support SD3.0 SDR50 and SDR104 modes.

27.1.1 Hardware Operation

The MMC communication is based on an advanced 11-pin serial bus designed to operate in a low voltage range. The uSDHC module supports MMC along with SD memory and I/O functions. The uSDHC controls the MMC, SD memory, and I/O cards by sending commands to cards and performing data accesses to and from the cards. The SD memory card system defines two alternative communication protocols: SD and SPI. The uSDHC only supports the SD bus protocol.

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

There are four 32-bit registers used to store the response from the card in the uSDHC. The uSDHC reads these four registers to get the command response directly. The uSDHC uses a fully configurable 128x32-bit FIFO for read and write. The buffer is used as temporary storage for data being transferred between the host system and the card, and vice versa. The uSDHC data buffer access register bits hold 32-bit data upon a read or write transfer.

For receiving data, the steps are as follows:

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

For transmitting data, the steps are as follows:

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

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

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

27.1.2 Software Operation

The Linux OS contains an MMC bus driver which implements the MMC bus protocols. The MMC block driver handles the file system read/write calls and uses the low level MMC host controller interface driver to send the commands to the uSDHC.

The MMC driver is responsible for implementing standard entry points for `init`, `exit`, `request`, and `set_ios`. The driver implements the following functions:

- The `init` function `esdhc_pltfm_init()` initializes the platform hardware and set platform dependant flags or values to sdhci_host structure.
- The `exit` function `esdhc_pltfm_exit()` deinitializes the platform hardware and frees the memory allocated.
- The function `esdhc_pltfm_get_max_clock()` gets the maximum SD bus clock frequency supported by the platform.
- The function `esdhc_pltfm_get_min_clock()` gets the minimum SD bus clock frequency supported by the platform.
- `esdhc_pltfm_get_ro()` gets the card read only status.
- `plt_8bit_width()` handles 8 bit mode switching on the platform.
- `plt_clk_ctrl()` handles clock management on the platform.
- `esdhc_prepare_tuning()` handles the preparation for tuning. It's only used for SD3.0 UHS-I mode.
- `esdhc_post_tuning()` handles the post operation for tuning.
- `esdhc_set_clock()` handles the clock change request.
- `cd_irq()` it's the interrupt routine for card detect.

Figure below shows how the MMC-related drivers are layered.

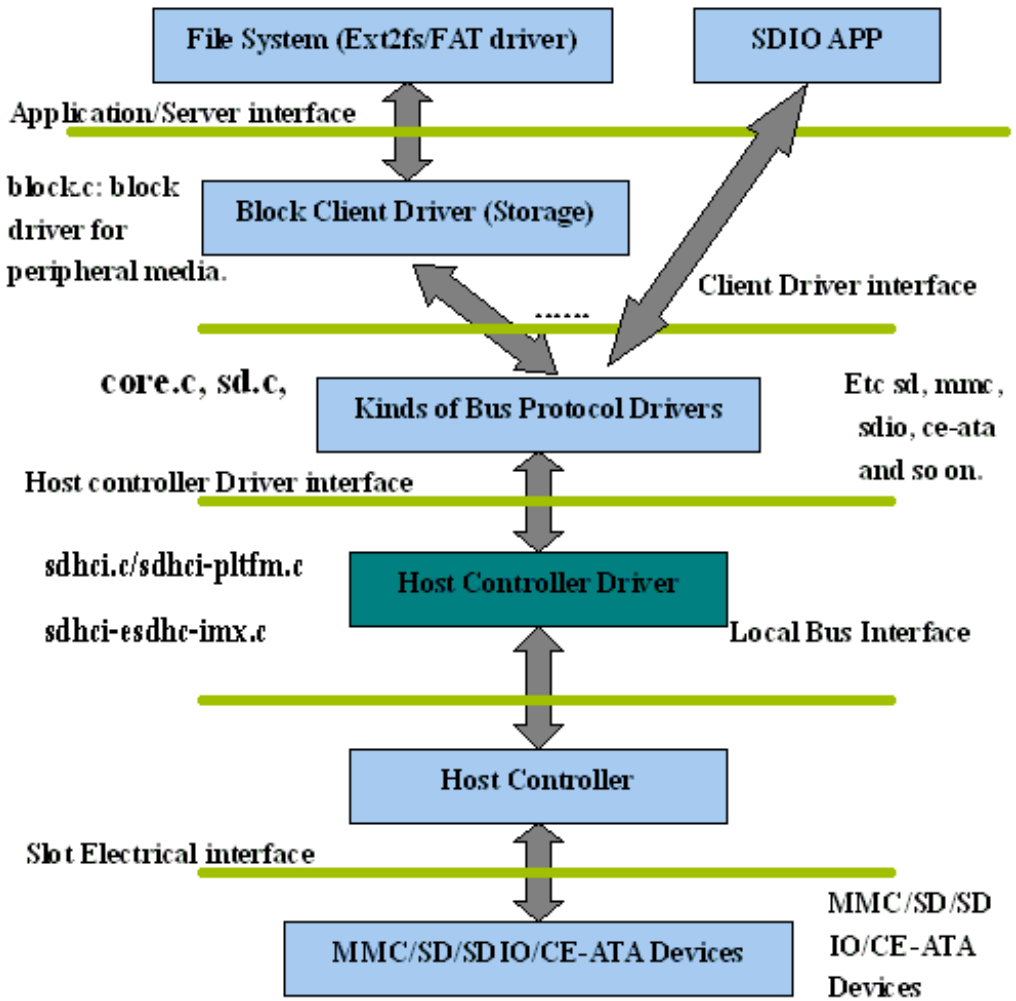


Figure 27-1. MMC Drivers Layering

27.2 Driver Features

The MMC driver supports the following features:

- Supports multiple uSDHC modules.
- Provides all the entry points to interface with the Linux MMC core driver.
- MMC and SD cards.
- SDIO cards.
- SD3.0 cards.
- Recognizes data transfer errors such as command time outs and CRC errors.
- Power management.
- It supports to be built as loadable or builtin module

27.2.1 Source Code Structure

Table below shows the uSDHC source files available in the source directory: <ltib_dir>/rpm/BUILD/linux/drivers/mmc/host/.

Table 27-1. uSDHC Driver Files MMC/SD Driver Files

File	Description
sdhci.c	sdhci standard stack code
sdhci-pltfm.c	sdhci platform layer
sdhci-esdhc-imx.c	uSDHC driver
sdhci-esdhc.h	uSDHC driver header file

27.2.2 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 builds 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 builds 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_SDHCI_ESDHC_IMX is used for the i.MX USDHC ports. In menuconfig, this option is found under:
  - Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller

  To compile SDHCI driver as a loadable module, several options should be selected as indicated below:
  - CONFIG_MMC_SDHCI=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support

    - CONFIG_MMC_SDHCI_PLTFM=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus.
    - CONFIG_MMC_SDHCI_ESDHC_IMX=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller

  To compile SDHCI driver as a builttin module, several options should be selected as indicated below:
    - CONFIG_MMC_SDHCI=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support
    - CONFIG_MMC_SDHCI_PLTFM=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus.
    - CONFIG_MMC_SDHCI_ESDHC_IMX=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller
- CONFIG_MMC_UNSAFE_RESUME is 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 > Assume MMC/SD cards are non-removable.

27.2.3 Platform Data

struct esdhc_platform_data defined in arch/arm/plat-mxc/include/mach/esdhc.h is used to pass platform informaton:

- .wp_gpio: GPIO used for write protect detection
- .cd_gpio: GPIO used for card detection
- .always_present: 1 indicates the card is inserted and non-removable, and the card detect is ignored
- .support_18v: indicate the board could provide 1.8v power to the card.
- .support_8bit: indicate 8 data pins are connected to the card slot.
- .platform_pad_change: callback function used to change the pad settings due to different SD bus clock frequency
- .keep_power_at_suspend: keep MMC/SD slot power when system enters suspend
- .delay_line: delay line setting for DDR mode

27.2.4 Programming Interface

This driver implements the functions required by the MMC bus protocol to interface with the i.MX uSDHC module.

See the Linux document generated from build: make htmldocs.

27.2.5 Loadable Module Operations

The SDHCI driver can be built as loadable or builtin module.

1. How to build SDHCI driver as loadable module.
   - CONFIG_MMC_SDHCI=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support
   - CONFIG_MMC_SDHCI_PLTFM=m, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus.
   - CONFIG_MMC_SDHCI_ESDHC_IMX=y, it can be found at Device Drivers > MMC/SD Card Support > Secure Digital Host Controller Interface support > SDHCI support on the platform specific bus > SDHCI platform support for the Freescale eSDHC i.MX controller
2. How to load and unload SDHCI module.

   Due to dependency, please load or unload the module following the module sequence shown below.

   run the following commands to load module:
   - load modules via insmod command, assuming the files of sdhci.ko and sdhci-platform.ko exist in current directory.

     ```
     $> insmod sdhci.ko
     $> insmod sdhci-platform.ko
     ```
   - load modules via modprobe command, please make sure the files of sdhci.ko and sdhci-platform.ko exist in corresponding kernel module lib directory.

     ```
     $> modprobe sdhci.ko
     $> modprobe sdhci-platform.ko
     ```

   run the following commands to unload module.:
   - unload modules via insmod command.

     ```
     $> rmsmod sdhci-platform
     $> rmsmod sdhci
     ```
   - unload modules via modprobe command.

```
$> modprobe -r sdhci-platform
$> modprobe -r sdhci
```

Chapter 28
NAND GPMI Flash Driver

28.1 Introduction

The NAND Flash Memory Technology Devices (MTD) driver is used in the Generic-Purpose Media Interface (GPMI) controller on the i.MX 6Dual/6Quad.

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.

28.1.1 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.MX 6Dual/6Quad contains a hardware error-correcting block.

28.2 Software Operation

MTDs in Linux covers 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.MX 6Dual/6Quad. 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 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

28.2.1 Basic Operations: Read/Write

The NAND driver exports the following callbacks:

```
mil_ecc_read_page (with ECC)
mil_ecc_write_page (with ECC)
mil_read_byte (without ECC)
mil_read_buf (without ECC)
mil_write_buf (without ECC)
mil_ecc_read_oob (with ECC)
mil_ecc_write_oob (with ECC)
```

These functions read the requested amount of data, with or without error correction. In the case of read, the mil_incoming_buffer_dma_begin 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 mil_incoming_buffer_dma_begin before processing the command NAND_CMD_PAGEPROG.

28.2.2 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 hardware accelerator-BCH.

For BCH, the page laylout of 2K page is (2k + 64), the page layout of 4K page is (4k + 218) the page layout of 8K page is (8K + 448).

28.2.3 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 application kobs-ng .

28.2.4 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-nfc-mil.c file:

```
mil_block_bad
mil_scan_bbt
```

28.3 Source Code Structure

The NAND driver is located in the drivers/mtd/nand/gpmi-nfc directory.

The following files are included in the NAND driver:

```
gpmi-nfc.c
hal-mx50.c
hal-mxs.c
gpmi-nfc.h
gpmi-regs.h
bch-regs.h
gpmi-regs-mx50.h
bch-regs-mx50.h
```

28.3.1 Menu Configuration Options

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

- CONFIG_IMX_HAVE_PLATFORM_GPMI_NFC = [Y]
- CONFIG_MTD_NAND_GPMI_NFC = [Y | M]

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

In addition, these UBI options must be enabled:

- CONFIG_MTD_UBI=y
- CONFIG_MTD_UBI_WL_THRESHOLD=4096
- CONFIG_MTD_UBI_BEB_RESERVE=1
- CONFIG_UBIFS_FS=y
- CONFIG_UBIFS_FS_LZO=y
- CONFIG_UBIFS_FS_ZLIB=y

Chapter 29
SATA Driver

29.1 Hardware Operation

The detailed hardware operation of SATA is detailed in the Synopsys DesignWare Cores SATA AHCI documentation, named SATA_Data_Book.pdf.

29.1.1 Software Operation

The details about the libata APIs, see the libATA Developer's Guide named libata.pdf pulished by Jeff Gazik.

The SATA AHCI driver is based on the LIBATA layer of the block device infrastructure of the Linux kernel . FSL integrated AHCI linux driver combined the standard AHCI drivers handle the details of the integrated freescale's SATA AHCI controller, while the LIBATA layer understands and executes the SATA protocols. The SATAdevice, such as a hard disk, is exposed to the application in user space by the /dev/sda* interface. Filesystems are built upon the block device. The AHCI specified integrated DMA engine, which assists the SATA controller hardware in the DMA transfer modes.

29.1.2 Source Code Structure Configuration

The source codes of freescale's AHCI sata driver is integrated into the plat-mxc relative files. <ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/ahci_sata.c.

The standard AHCI and AHCI platform drivers are used to do the actual sata operations.

The source codes of the standard AHCI and AHCI platform drivers are located in drivers/ata/ folder, named as ahci.c and ahci-platform.c.

29.1.3 Linux Menu Configuration Options

The following Linux kernel configurations are provided for SATA driver:

- CONFIG_SATA_AHCI_PLATFORM: Configure options for SATA driver. In the menuconfig this option is available under "Device Drivers --->Serial ATA (prod) and Parallel ATA (experimental) drivers -> Platform AHCI SATA support".
-

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

29.1.4 Board Configuration Options

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

29.2 Programming Interface

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

29.2.1 Usage Example2

NOTE

There may be a known error message when few kinds of SATA disks are initialized, such as:

ata1.00: serial number mismatch '090311PB0300QKG3TB1A' ! = ''

ata1.00: revalidation failed (errno=-19)

pls ignore that.

1. After building the kernel and the SATA AHCI driver and deploying, boot the target, and log in as root.
2. Make sure that the AHCI and AHCI paltform drivers are built in kernel or loaded into kernel. Use the following commands to load the drivers into kernel.

```
# insmod libata.ko
# insmod libahci.ko
# insmod ahci-platform.ko
```

You should see messages similar to the following:

```
ahci: SSS flag set, parallel bus scan disabled
ahci ahci.0: AHCI 0001.0100 32 slots 1 ports 3 Gbps 0x1 impl platform mode
ahci ahci.0: flags: ncq sntf stag pm led clo only pmp pio slum part ccc
scsi0 : ahci
ata1: SATA max UDMA/133 irq_stat 0x00000040, connection status changed irq 28
ata1: SATA link up 1.5 Gbps (SStatus 113 SControl 300)
ata1.00: ATA-8: Hitachi HTS545032B9A300, PB3OC60G, max UDMA/133
ata1.00: 625142448 sectors, multi 0: LBA48 NCQ (depth 31/32)
ata1.00: serial number mismatch '090311PB0300QKG3TB1A' != ''
ata1.00: revalidation failed (errno=-19)
ata1: limiting SATA link speed to 1.5 Gbps
ata1.00: limiting speed to UDMA/133:PIO3
ata1: SATA link up 1.5 Gbps (SStatus 113 SControl 310)
ata1.00: configured for UDMA/133
scsi 0:0:0:0: Direct-Access     ATA      Hitachi HTS54503 PB3O PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 625142448 512-byte logical blocks: (320 GB/298 GiB)
sd 0:0:0:0: [sda] Write Protect is off
sd 0:0:0:0: [sda] Write cache: enabled, read cache: enabled, doesn't support DPO or FUA
 sda: sda1 sda2 sda3
sd 0:0:0:0: [sda] Attached SCSI disk
```

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

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

29.2.2 Usage Example

Create Partitons

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

```
$fdisk -l /dev/sda
```

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

```
$fdisk /dev/sda
```

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

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

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

```
$dd if=/dev/zero of=/dev/sda1 bs=1024 count=5
```

The response should be as follows:

5+0 records in

5+0 records out

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

```
$dd if=file_name of=/dev/sda1
```

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

```
$dd if=/dev/sda1 of=output_file bs=1024 count=1
```

Files System Tests

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

```
$mkfs.ext2 /dev/sda1
$mkfs.vfat /dev/sda1
```

Mount the file system as follows:

```
$mkdir /mnt/sda1
$mount -t ext2 /dev/sda1 /mnt/sda1
```

After mounting, file/directory, operations can be performed in /mnt/sda1.

Unmount the filesystem as follows:

```
$umount /mnt/sda1
```

Chapter 30
Inter-IC (I2C) Driver

30.1 Introduction

I2C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices.

The I2C driver for Linux has two parts:

- I2C bus driver-low level interface that is used to talk to the I2C bus
- I2C chip driver-acts as an interface between other device drivers and the I2C bus driver

30.1.1 I2C Bus Driver Overview

The I2C bus driver is invoked only by the I2C chip driver and is not exposed to the user space.

The standard Linux kernel contains a core I2C module that is used by the chip driver to access the I2C bus driver to transfer data over the I2C bus. The chip driver uses a standard kernel space API that is provided in the Linux kernel to access the core I2C module. The standard I2C 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 I2C 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 I2C master mode

30.1.2 I2C Device Driver Overview

The I2C device driver implements all the Linux I2C data structures that are required to communicate with the I2C 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 I2C bus. Internally, these API functions use the standard I2C kernel space API to call the I2C core module. The I2C core module looks up the I2C bus driver and calls the appropriate function in the I2C 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.

30.1.3 Hardware Operation

The I2C module provides the functionality of a standard I2C master and slave.

It is designed to be compatible with the standard Philips I2C 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

30.2 Software Operation

The I2C driver for Linux has two parts: an I2C bus driver and an I2C chip driver.

30.2.1 I2C Bus Driver Software Operation

The I2C 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 I2C bus. The algorithm structure contains a pointer to a function that is called whenever the I2C chip driver wants to communicate with an I2C device.

During startup, the I2C bus adapter is registered with the I2C core when the driver is loaded. Certain architectures have more than one I2C module. If so, the driver registers separate i2c_adapter structures for each I2C module with the I2C core. These adapters are unregistered (removed) when the driver is unloaded.

After transmitting each packet, the I2C 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 I2C bus driver uses wait queues for its operation, other device drivers should be careful not to call the I2C API methods from an interrupt mode.

30.2.2 I2C Device Driver Software Operation

The I2C driver controls an individual I2C device on the I2C bus. A structure, i2c_driver, describes the I2C 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 I2C devices, after the driver is loaded. The attach_adapter callback function is called whenever a new I2C bus driver is loaded in the system. When the I2C 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.

30.3 Driver Features

The I2C driver supports the following features:

- I2C communication protocol
- I2C master mode of operation

NOTE

The I2C driver does not support the I2C slave mode of operation.

30.3.1 Source Code Structure

Table below shows the I2C bus driver source files available in the directory:

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

Table 30-1. I2C Bus Driver Files

File	Description
i2c-imx.c	I2C bus driver source file

30.3.2 Menu Configuration Options

To get to the Linux kernel configuration option provided for this module, use the ./ltib -c command when located in the <ltib dir>.

On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

Device Drivers > I2C support > I2C Hardware Bus support > IMX I2C interface.

30.3.3 Programming Interface

The I2C device driver can use the standard SMBus interface to read and write the registers of the device connected to the I2C bus.

For more information, see <ltib_dir>/rpm/BUILD/linux/include/linux/i2c.h.

30.3.4 Interrupt Requirements

The I2C module generates many kinds of interrupts.

The highest interrupt rate is associated with the transfer complete interrupt as shown in table below.

Table 30-2. I2C Interrupt Requirements

Parameter	Equation	Typical	Best Case
Rate	Transfer Bit Rate/8	25,000/sec	50,000/sec
Latency	8/Transfer Bit Rate	40 Î¼s	20 Î¼s

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 I2C interface).

Chapter 31
Enhanced Configurable Serial Peripheral Interface (ECSPI) Driver

31.1 Introduction

The ECSPI driver implements a standard Linux driver interface to the ECSPI controllers.

It supports the following features:

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

31.1.1 Hardware Operation

ECSPI is used for fast data communication with fewer software interrupts than conventional serial communications.

Each ECSPI is equipped with a data FIFO and is a master/slave configurable serial peripheral interface module, allowing the processor to interface with external SPI master or slave devices.

The primary features of the ECSPI includes:

- Master/slave-configurable
- Four chip select signals to support multiple peripherals
- Up to 32-bit programmable data transfer
- 64 x 32-bit FIFO for both transmit and receive data
- Configurable polarity and phase of the Chip Select (SS) and SPI Clock (SCLK)

31.2 Software Operation

The following sections describe the ECSPI software operation.

31.2.1 SPI Sub-System in Linux

The ECSPI driver layer is located between the client layer (SPI-NOR Flash are examples of clients) and the hardware access layer. Figure below shows the block diagram for SPI subsystem in Linux.

The SPI requests go into I/O queues. Requests for a given SPI device are executed in FIFO order and they complete asynchronously through completion callbacks. There are also some simple synchronous wrappers for those calls including the ones for common transaction types such as writing a command and then reading its response.

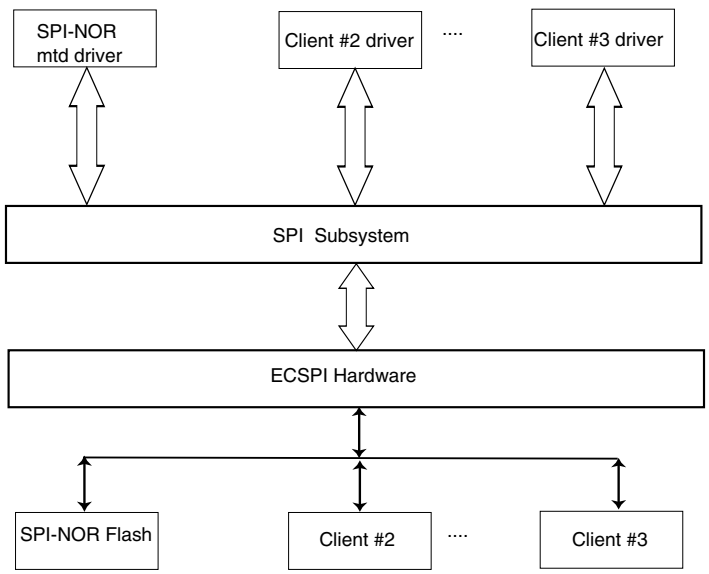


Figure 31-1. SPI Subsystem

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

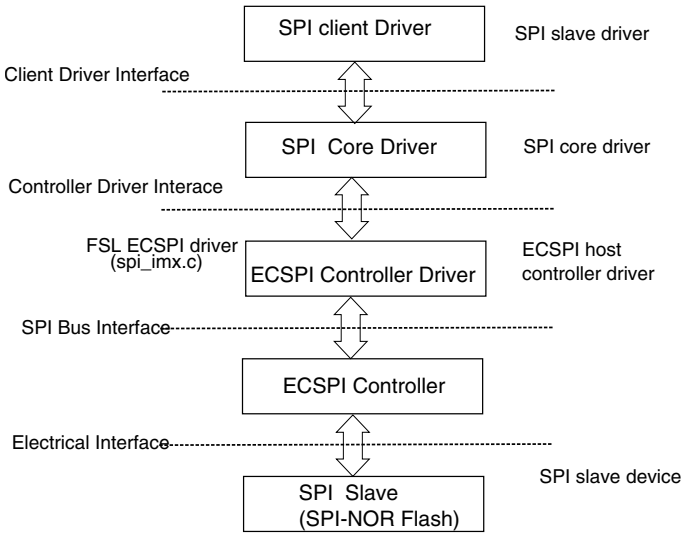


Figure 31-2. Layering of SPI Drivers in SPI Subsystem

31.2.2 Software Limitations

The ECSPI driver limitations are as follows:

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

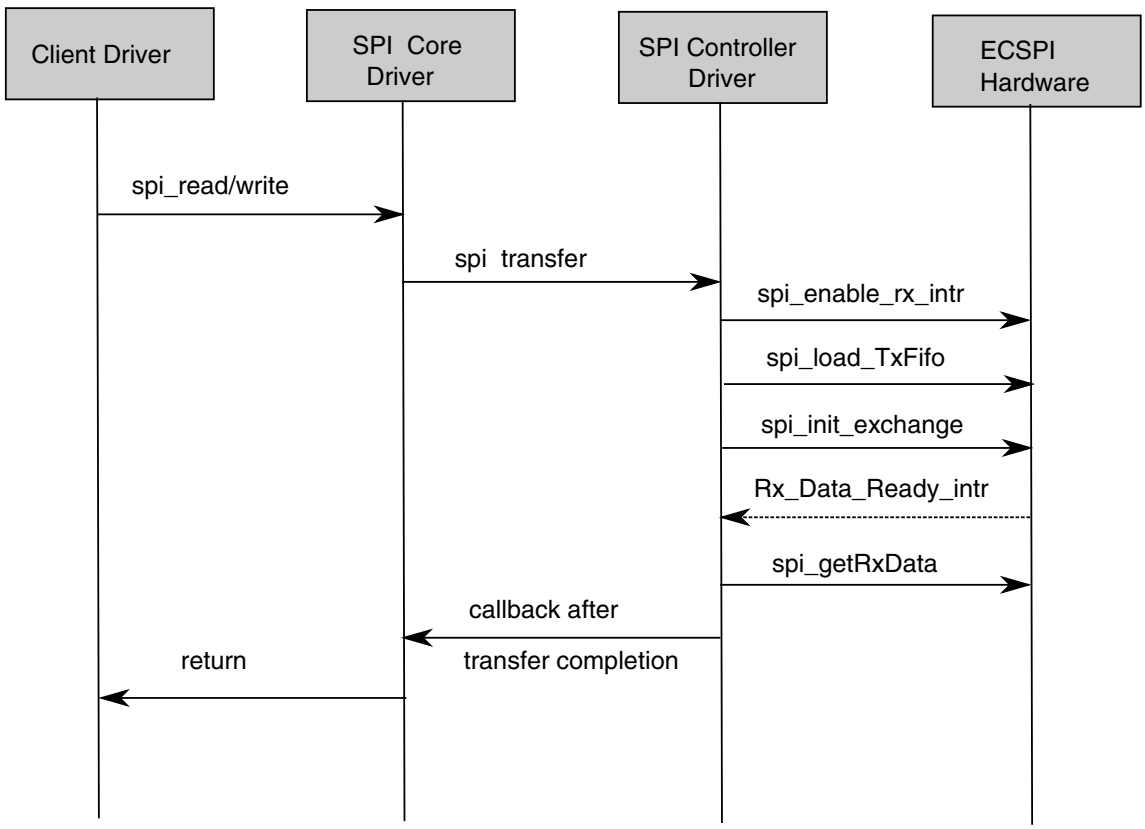
31.2.3 Standard Operations

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

- Init function spi_imx_init() registers the device_driver structure.
- Probe function spi_imx_probe() performs initialization and registration of the SPI device specific structure with SPI core driver. The driver probes for memory and IRQ resources. Configures the IOMUX to enable ECSPI I/O pins, requests for IRQ and resets the hardware.
- Chip select function spi_imx_chipselect() configures the hardware ECSPI for the current SPI device. Sets the word size, transfer mode, data rate for this device.
- SPI transfer function spi_imx_transfer() handles data transfers operations.
- SPI setup function spi_imx_setup() initializes the current SPI device.
- SPI driver ISR spi_imx_isr() is called when the data transfer operation is completed and an interrupt is generated.

31.2.4 ECSPI Synchronous Operation

Figure below shows how the ECSPI provides synchronous read/write operations.


Client Driver
SPI Core Driver
SPI Controller Driver
ECSPI Hardware
spi_read/write
spi transfer
spi_enable_rx_intr
spi_load_TxFifo
spi_init_exchange
Rx_Data_Ready_intr
spi_getRxData
callback after
transfer completion
return

31.3 Driver Features

The ECSPI module supports the following features:

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

31.3.1 Source Code Structure

Table below shows the source files available in the devices directory:

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

Table 31-1. CSPI Driver Files

File	Description
spi_imx.c	SPI Master Controller driver

31.3.2 Menu Configuration Options

To get to the Linux kernel configuration options provided for this module, use the ./ltib -c command when located in the <ltib dir>.

On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following options to enable this module:

- CONFIG_SPI build support for the SPI core. In menuconfig, this option is available under:
  - Device Drivers > SPI Support.
- CONFIG_BITBANG is the Library code that is automatically selected by drivers that need it. SPI_IMX selects it. In menuconfig, this option is available under:
  - Device Drivers > SPI Support > Utilities for Bitbanging SPI masters.
- CONFIG_SPI_IMX implements the SPI master mode for ECSPI. In menuconfig, this option is available under:
  - Device Drivers > SPI Support > Freescale i.MX SPI controllers.

31.3.3 Programming Interface

This driver implements all the functions that are required by the SPI core to interface with the ECSPI hardware.

For more information, see the Linux document generated from build: make htmldocs.

31.3.4 Interrupt Requirements

The SPI interface generates interrupts.

ECSPI interrupt requirements are listed in table below.

Table 31-2. ECSPI Interrupt Requirements

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

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

Chapter 32
FlexCAN Driver

32.1 Driver Overview

FlexCAN is a communication controller implementing the CAN protocol according to the CAN 2.0B protocol specification.

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

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

32.1.1 Hardware Operation

For the information on hardware operations, see the *i.MX 6 Multimedia Applications Processor Reference Manual*.

32.1.2 Software Operation

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

The CAN network device driver interface.

The CAN network device driver interface provides a generic interface to setup, configure and monitor CAN network devices. The user can then configure the CAN device, like setting the bit-timing parameters, via the netlink interface using the program "ip" from the "IPROUTE2" utility suite.

Starting and stopping the CAN network device.

A CAN network device is started or stopped as usual with the command "ifconfig canX up/down" or "ip link set canX up/down". Be aware that you *must* define proper bit-timing parameters for real CAN devices before you can start it to avoid error-prone default settings:

- ip link set canX up type can bitrate 125000

The iproute2 tool also provides some other configuration capbilities for can bus such as bit-timing setting. For details, please refer to kernel doc: Documentation/networking/can.txt

32.1.3 Source Code Structure

Table below shows the driver source file available in the directory, <ltib_dir>/rpm/BUILD/linux/drivers/net/can/

Table 32-1. FlexCAN Driver Files

File	Description
flexcan.c	flexcan driver

32.1.4 Linux Menu Configuration Options

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

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

  Networking > CAN bus subsystem support.

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

Networking > CAN bus subsystem support > Raw CAN Protocol (raw access with CAN-ID filtering).

- CONFIG_CAN_BCM - Build support for Broadcast Manager CAN protocol. In menuconfig, this option is available under

  Networking > CAN bus subsystem support > Broadcast Manager CAN Protocol (with content filtering).

- CONFIG_CAN_VCAN - Build support for Virtual Local CAN interface (also in Ethernet interface). In menuconfig, this option is available under

  Networking > CAN bus subsystem support > CAN Device Driver > Virtual Local CAN Interface (vcan).

- CONFIG_CAN_DEBUG_DEVICES - Build support to produce debug messages to the system log to the driver. In menuconfig, this option is available under

  Networking > CAN bus subsystem support > CAN Device Driver > CAN devices debugging messages.

- CONFIG_CAN_FLEXCAN - Build support for FlexCAN device driver. In menuconfig, this option is available under

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

Chapter 33
Media Local Bus Driver

33.1 Introduction

MediaLB is an on-PCB or inter-chip communication bus specifically designed to standardize a common hardware interface and software API library.

This standardization allows an application or multiple applications to access the MOST Network data or to communicate with other applications with minimum effort. MediaLB supports all the *MOST Network data transport* methods: synchronous stream data, asynchronous packet data, and control message data. MediaLB also supports an isochronous data transport method. For detailed information about the MediaLB, see the Media Local Bus Specification.

33.1.1 MLB Device Module

The MediaLB module implements the Physical Layer and Link Layer of the MediaLB specification, interfacing the i.MX to the MediaLB controller.

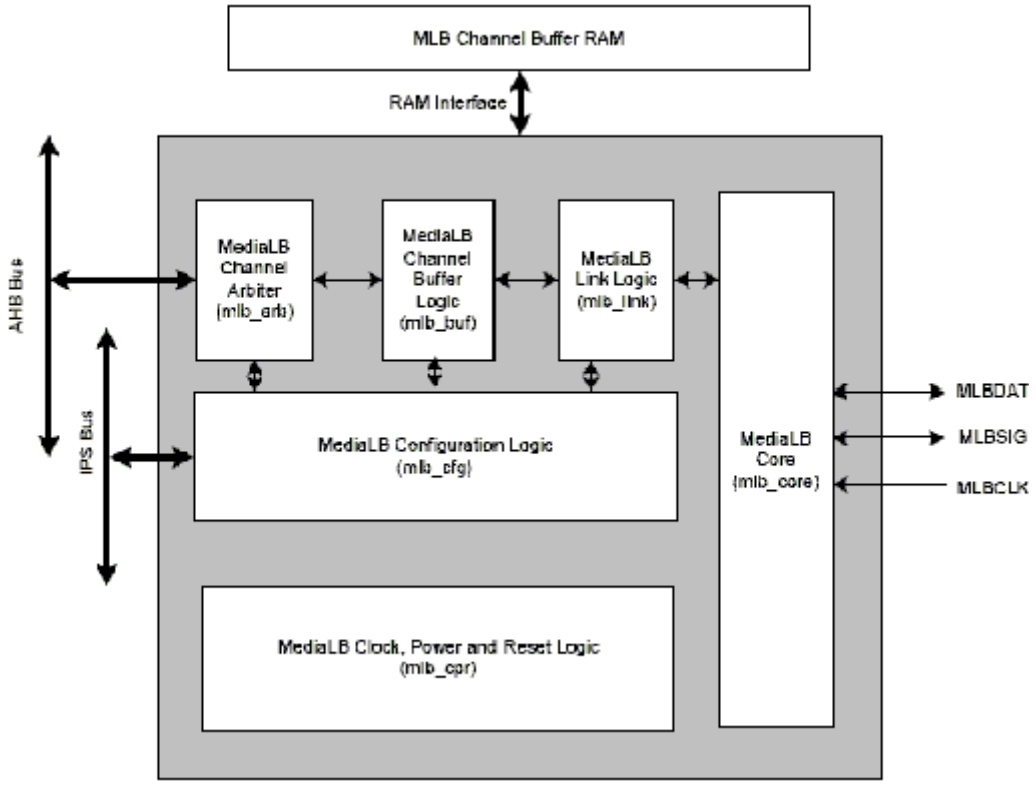


Figure 33-1. MLB Device Top-Level Block Diagram

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

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

33.1.2 Supported Feature

- Synchronous, asynchronous, control and isochronous channel.
- Up to 16 logical channels and 31 physical channels running at a maximum speed of 1024Fs
- Transmission of commands and data and reception of receive status when functioning as the transmitting device associated with a logical channel address

- Reception of commands and data and transmission as receive status responses when functioning as the receiving device associated with a logical channel address
- MediaLB lock detection
- System channel command handling

33.1.3 Modes of Operation

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

33.1.4 MLB Driver Overview

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

33.2 MLB Driver

Functionality of the MLB driver is described in supported features, MLB driver architecture, and software operation.

33.2.1 Supported Features

- 256Fs, 512Fs and 1024Fs frame rates
- Asynchronous and control channel types
- The following configurations to MLB device module:
  - Frame rate
  - Device address
  - Channel address
- MLB channel exception get interface. All the channel exceptions are sent and handled by the application.

33.2.2 MLB Driver Architecture

The MLB driver is a common linux character driver and the architecture is shown in figure below.

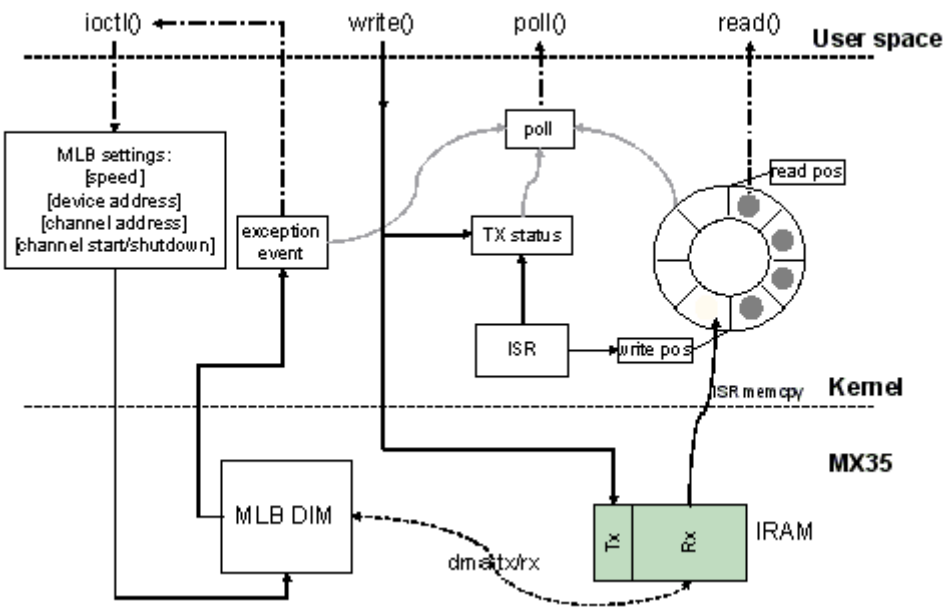


Figure 33-2. MLB Driver Architecture Diagram

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

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

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

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

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

33.2.3 Software Operation

The MLB driver provides a common interface to application.

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

`MLB_SET_FPS`

Argument type: unsigned int

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

`MLB_GET_VER`

Argument type: unsigned long

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

`MLB_SET_DEVADDR`

Argument type: unsigned char

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

`MLB_CHAN_SETADDR`

Argument type: unsigned int

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

`MLB_CHAN_STARTUP`

Startup the corresponding type of channel for transmit and reception.

`MLB_CHAN_SHUTDOWN`

Shutdown the corresponding type of channel.

`MLB_CHAN_GETEVENT`

Argument type: unsigned long

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

```
MLB_EVT_TX_PROTO_ERR_CUR
MLB_EVT_TX_BRK_DETECT_CUR
MLB_EVT_RX_PROTO_ERR_CUR
MLB_EVT_RX_BRK_DETECT_CUR
```

33.3 Driver Files

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

Table 33-1. MLB Driver Source File List

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

33.4 Menu Configuration Options

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

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

Chapter 34
ARC USB Driver

34.1 Introduction

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 OTG core supported
- High Speed Host Only core(Host1), high speed, full speed, and low devices are supported.
- High Speed Inter-Chip core(Host2 & Host3)
- Host mode-Supports HID (Human Interface Devices), MSC (Mass Storage Class)
- Peripheral mode-Supports MSC, and CDC (Communication Devices Class) drivers which include ethernet and serial support
- Embedded DMA controller

34.1.1 Architectural Overview

The USB host system is composed of a number of hardware and software layers.

Figure below shows a conceptual block diagram of the building block layers in a host system that support USB 2.0.

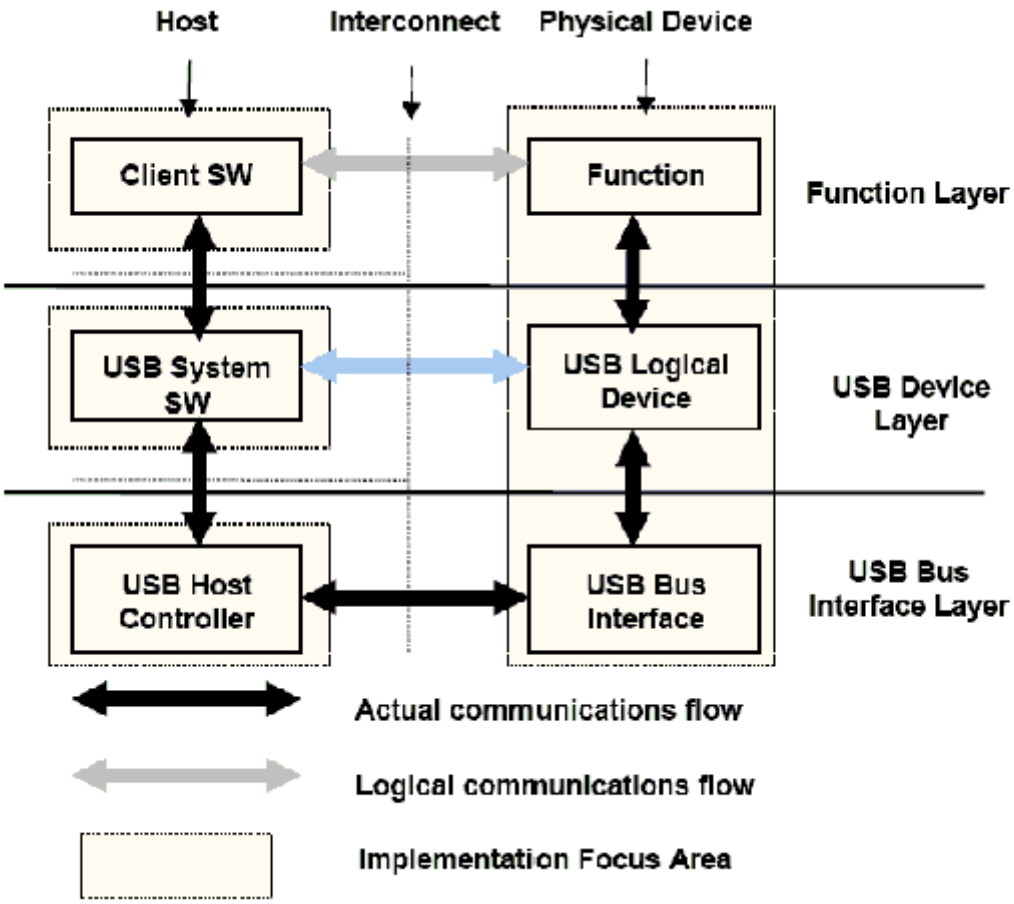


Figure 34-1. USB Block Diagram

34.2 Hardware Operation

For information on hardware operations, refer to the EHCI spec.ehci-r10.pdf.

The spec is available at http://www.usb.org/developers/docs/

34.2.1 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, ID dynamic switch host/device modes are supported. Full OTG functions are temporarily not supported.

34.2.2 Source Code Structure

Table below shows the source files available in the source directory, <ltib_dir>/rpm/BUILD/linux/drivers/usb.

Table 34-1. USB Driver Files

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

Table continues on the next page...

Table 34-1. USB Driver Files (continued)

File	Description
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 below shows the platform related source files.

Table 34-2. USB Platform Source Files

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

Table below shows the platform-related source files in the directory:

`<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/`

Table 34-3. USB Platform Header Files

File	Description
usb_dr.c	Platform-related initialization
usb_h1.c	Platform-related initialization
usb_h2.c	Platform-related initialization
usb_h3.c	Platform-related initialization

Table below shows the common platform source files in the directory:

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

Table 34-4. USB Common Platform Files

File	Description
isp1504xc.c	ULPI PHY driver (USB3317 uses the same driver as ISP1504)
utmixc.c	Internal UTMI transceiver driver
usb_hsic_xcvr.c	HSIC featured phy's interface
usb_common.c	Common platform related part of USB driver
usb_wakeup.c	Handle USB wakeup events

34.2.3 Menu Configuration Options

To get to the Linux kernel configuration options available for this module, 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 Y.

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

  By default, this option is Y.

- CONFIG_USB_EHCI_ARC_OTG-Build support for selecting the ARC EHCI OTG host. In menuconfig, this option is available under

  Device drivers > USB support > EHCI HCD (USB 2.0) support > Support for DR host port on Freescale controller.

  By default, this option is Y.

- CONFIG_USB_EHCI_ARC_HSIC Freescale HSIC USB Host Controller

  By default, this option is N.

- CONFIG_USB_EHCI_ROOT_HUB_TT-Some EHCI chips have vendor-specific extensions to integrate transaction translators, so that no OHCI or UHCI companion controller is needed. 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_ARC && USB_EHCI_HCD.

- 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_IMX_USB_CHARGER Freescale i.MX 6 USB Charger Detection

  By default, this option is N.

- CONFIG_USB_OTG-OTG Support, support dual role with ID pin detection.

  By default, this option is Y.

- CONFIG_MXC_OTG-USB OTG pin detect support for Freescale USB OTG Controller

  By default, this option is Y.

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

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

  By default, this option is M.

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

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

  By default, this option is Y.

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

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

By default, this option is M.

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

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

  By default, this option is M.

34.2.4 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.

34.3 System WakeUp

Both host and device connect/disconnect event can be system wakeup source, as well the device remote wakeup.

But all the wakeup functions depend on the USB PHY power supply, including 1p1, 2p5, 3p3, no power supply, all the wakeup function behavior will be unpredictable.

For host remote wake feature, there is a limitation that our system clock needs a short time to be stable after resume, if the resume signal sent by the connected device only last very short time (less than the time need to make clock stable), the remote wakeup may fail. At such case, we should not turn off some clocks to decrease the time needs to be stable to fix such issue.

34.3.1 USB Wakeup usage

USB wakeup usage is outlined in three procedures: how to enable USB wakeup system, what kinds of wakeup events USB supports, and how to close USB child device power.

34.3.2 How to Enable USB WakeUp System Ability

For otg port:

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup
```

For device-only port:

```
echo enabled > /sys/devices/platform/fsl-usb2-udc/power/wakeup
```

For host-only port:

```
echo enabled > /sys/devices/platform/fsl-ehci.x/power/wakeup
(x is the port num)
```

For USB child device:

```
echo enabled > /sys/bus/usb/devices/1-1/power/wakeup
```

34.3.3 WakeUp Events Supported by USB

USBOTG port is used as an example.

Device mode wakeup:

connect wakeup: when USB line connects to usb port, the other port is connected to PC (Wakeup signal: vbus change)

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup
```

Host mode wakeup:

connect wakeup: when USB device connects to host port (Wakeup signal: ID/(dm/dp) change)

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup
```

disconnect wakeup: when USB device disconnects to host port (Wakeup signal: ID/(dm/dp) change)

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup
```

remote wakeup: press USB device (i.e. press USB key on the USB keyboard) when USB device connects to host port (Wakeup signal: ID/(dm/dp) change):

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup
echo enabled > /sys/bus/usb/devices/1-1/power/wakeup
```

NOTE

For the hub on board, it is necessary to enable hub's wakeup first. For remote wakeup, it is necessary to perform the three steps outlined below:

```
echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup (enable the roothub's wakeup)
echo enabled > /sys/bus/usb/devices/1-1/power/wakeup (enable the second level hub's wakeup)
(1-1 is the hub name)
echo enabled > /sys/bus/usb/devices/1-1.1/power/wakeup (enable the USB device wakeup, that
device connects at second level hub)
(1-1.1 is the USB device name)
```

34.3.4 How to Close the USB Child Device Power

The following code string outlines how to close the USB child device power:

```
echo auto > /sys/bus/usb/devices/1-1/power/control
echo auto > /sys/bus/usb/devices/1-1.1/power/control (If there is a hub at usb device)
```

Chapter 35
i.MX 6 PCI Express Root Complex Driver

35.1 Introduction

PCI Express hardware module, contained in i.MX 6 SoC, can either be configured to act as a Root Complex or a PCIe Endpoint.

This chapter describes the PCI Express Root Complex implementation on i.MX 6Dual/6Quad/6Solo/6DualLite SOC's families.

It also describes the drivers that need to be configured and operated on the i.MX 6 PCI Express device as Root Complex.

35.1.1 PCIe

PCI Express (PCIe) is Third Generation I/O Interconnect, targeting low cost, high volume, multi-platform interconnection usages. It has the concepts with earlier PCI and PCI-X and offers backwards compatibility for existing PCI software with following differences:

- PCIe is a point-to-point interconnect
- Serial link between devices
- Packet based communication
- Scalable performance via aggregated Lanes from X1 to X16
- Need PCIe switch to have connection between more than two PCIe devices

35.1.2 Terminology and Conventions

Following terminologies and conventions are used in this document:

- Bridge

A Function that virtually or actually connects a PCI/PCI-X segment or PCI Express Port with an internal component interconnect or with another PCI/PCI-X bus segment or PCI Express Port.

- Downstream
  - 1. The relative position of an interconnect/System Element (Port/component) that is farther from the Root Complex. The Ports on a Switch that are not the Upstream Port are Downstream Ports. All Ports on a Root Complex are Downstream Ports. The Downstream component on a Link is the component farther from the Root Complex.
  - 2. A direction of information flow where the information is flowing away from the Root Complex.
- Endpoint

  One of several defined System Elements. A Function that has a Type 00h Configuration Space header.

- Host

  The entity comprising of one (or more) Central Processing Unit(s) (CPU) and resources, such as Memory (RAM) that can be shared across multiple PCIe nodes connected through a Root Complex.

- Lane

  A set of differential signal pairs, one pair for transmission and one pair for reception.

- Link

  The collection of two Ports and their interconnecting Lanes. A Link is a dual simplex communications path between two components.

- PCIe Fabric

  A topology comprised of various PCI Express nodes, also referred as devices. A device in the fabric can be Root Complex, Endpoint, PCIe-PCI/PCI-X Bridge or a Switch.

- Port
  - 1. Logically, an interface between a component and a PCI Express Link.
  - 2. Physically, a group of Transmitters and Receivers located on the same chip that define a Link.
- Root Complex

  RC A defined System Element that includes a Host Bridge, zero or more Root Complex Integrated Endpoints, zero or more Root Complex Event Collectors, and one or more Root Ports

- Root Port

  A PCI Express Port on a Root Complex that maps a portion of the Hierarchy through an associated virtual PCI-PCI Bridge.

- Upstream
  - 1. The relative position of an interconnect/System Element (Port/component) that is closer to the Root Complex. The Port on a Switch that is closest topologically to the Root Complex is the Upstream Port. The Port on a component that contains only Endpoint or Bridge Functions is an Upstream Port. The Upstream component on a Link is the component closer to the Root Complex.
- http://intellinuxwireless.org/?n=Info

Any element of the fabric which is relatively closer towards RC is treated as 'Upstream'. All PCIe Endpoint ports (including termination points for bridges) and Switch ports, which are closer to RC are called Upstream Ports on that device. A Upstream Flow is the communication moving towards RC.

35.1.3 PCIe Topology on i.MX 6 in PCIe RC Mode

There is one PCIe port on the i.MX 6.

The following figure describes the diagram of the PCIe RC port on i.MX 6.

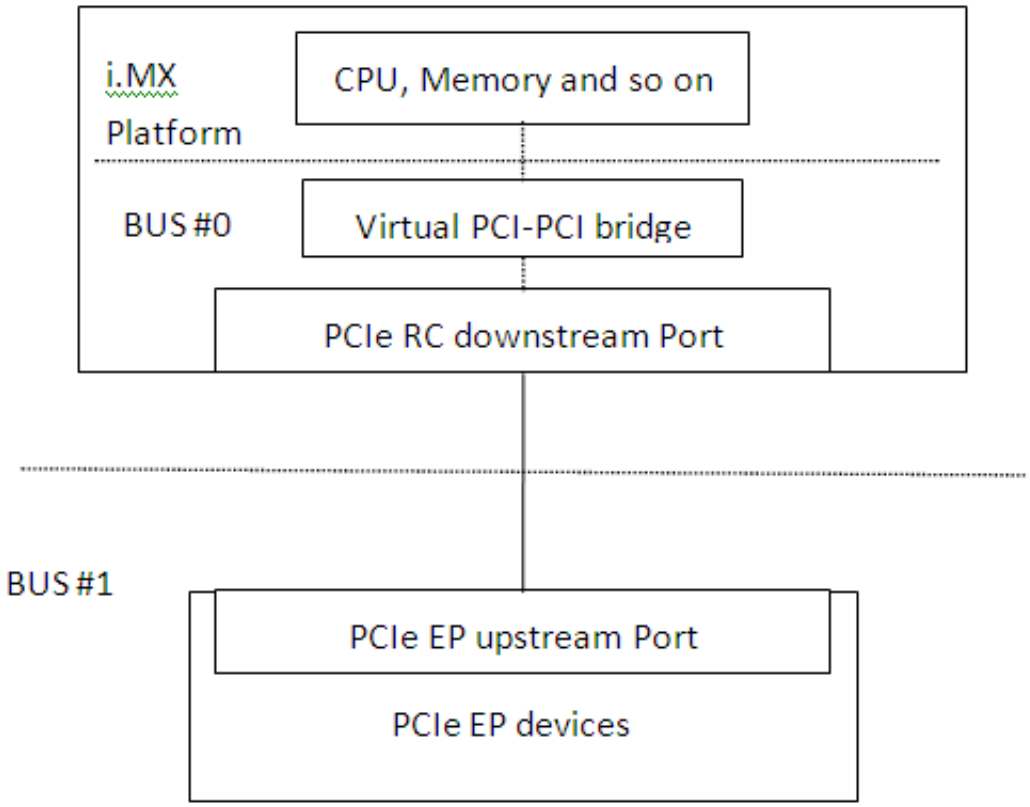


Figure 35-1. diagram of the PCIe RC port on i.MX 6

PCI Enumeration Mapping

Since PCI Express is point to point topology, to maintain compatibility with legacy PCI Bus - Device notion used for Software Enumeration, we introduce following concepts which allow identifying various nodes and their internals (e.g., PCIe Switches) in terms of PCI devices/functions:

- Host Bridge: A bridge, integrated into RC to have PCI compatible connection to Host. The PCI side of this bridge is Bus #0 always. This means, the device on this bus will be the host itself.
- Virtual PCI-PCI Bridge: Each PCI Express port which is part of RC or a Switch is treated as a virtual PCI-PCI bridge. This means each port has a primary and secondary PCI bus and the downstream is mapped into the remote configuration space.
- Root port associated virtual bridge has Bus #0 on the primary side with secondary bus on the downstream.
- Each PCIe Switch is viewed as collection of as many virtual PCI-PCI bridges as number of downstream ports, connected to a virtual PCI bus which is actually secondary bus of another PCI-PCI bridge forming the upstream port of the switch.
- The upstream port of each EP can either be part of the secondary bus segment of virtual PCI-PCI Bridge representing downstream port of a switch or of the root port.

35.1.4 Features

Listed below are the various features supported by i.MX 6 as a PCI Express Root Complex driver.

- Express Base Specification Revision 2.0 compliant
- Gen1 operation with x1 link supporting 5 GT/s raw transfer rate in single direction
- Support Legacy Interrupts (INTx), and MSI
- Configurable Max_Payload_Size size (128 bytes to 4 KB)
- 4-KB maximum Request size
- It fits into Linux PCI Bus framework to provide PCI compatible software enumeration support
- In addition, it provides interface to Endpoint Drivers to access the respective devices detected downstream.
- The same interface can be used by the PCI Express Port Bus Driver framework in Linux to perform AER, ASP etc handling
- Interrupt handling facility for EP drivers either as Legacy Interrupts (INTx).
- Access to EP I/O BARs through generic I/O accessories in Linux PCI subsystem.
- Seamless handling of PCIe errors

NOTE

Out of the above, MSI, I/O access, Port Bus Driver integration are currently incomplete.

35.2 Linux PCI Subsystem and RC driver

In Linux, the PCI implementation can roughly be divided into following main components: PCI BIOS architecture specific Linux implementation, Host Controller (RC) Module, and Core.

- PCI BIOS Architecture specific Linux implementation to kick off PCI bus initialization. It interfaces with PCI Host Controller code as well as the PCI Core to perform bus enumeration and allocation of resources such as memory and interrupts. The successful completion of BIOS execution assures that all the PCI devices in the system are assigned parts of available PCI resources and their respective drivers (referred as Slave Drivers). PCI can take control of them using the facilities provided by PCI Core. It is possible to skip resource allocation (if they were assigned before Linux was booted, for example PC scenario).

- Host Controller (RC) Module handles hardware (SoC + Board) specific initialization and configuration and it invokes PCI BIOS. It should provide callback functions for BIOS as well as PCI Core, which will be called during PCI system initialization and accessing PCI bus for configuration cycles. It provides resources information for available memory/IO space, INTx interrupt lines, MSI. It should also facilitate IO space access (as supported) through in _x_ () out _x_ () You may need to provide indirect memory access (if supported by h/w) through read _x_ () write _x_ ()
- Core creates and initializes the data structure tree for bus devices as well as bridges in the system, handles bus/device numberings, creates device entries and proc/sysfs information, provides services for BIOS and slave drivers and provides hot plug support (optional/as supported by h/w). It targets (EP) driver interface query and initializes corresponding devices found during enumeration. It also provides MSI interrupt handling framework and PCI express port bus support. It provides Hot-Plug support (if supported), advanced error reporting support, power management event support, and virtual Channel support to run on PCI express ports (if supported).

35.2.1 RC driver source files

The driver files are present at the following path relative to extracted kernel source directory.

arch/arm/mach-mx6/pcie.c (RC driver source)

arch/arm/mach-mx6/include/mach/pci.h (Define the platform data structure for RC driver)

35.2.2 Kernel configurations

Root Complex is not supported by the default kernel configurations on i.MX 6 boards.

To set the default configuration, execute the following command as follows:

```
make CROSS_COMPILE=arm-none-linux-gnueabi- ARCH=imx6_defconfig
```

Configure the Root Complex to be built in:

```
Prompt: PCI Express support
Defined at arch/arm/mach-mx6/Kconfig:171
Depends on: ARCH_MXC [=y] && ARCH_MX6 [=y]
Location:
-> System Type
-> Freescale MXC Implementations
Selects: PCI [=y]
```

NOTE

PCI Express support can't be built as a module.

35.3 System Resource: Memory Layout

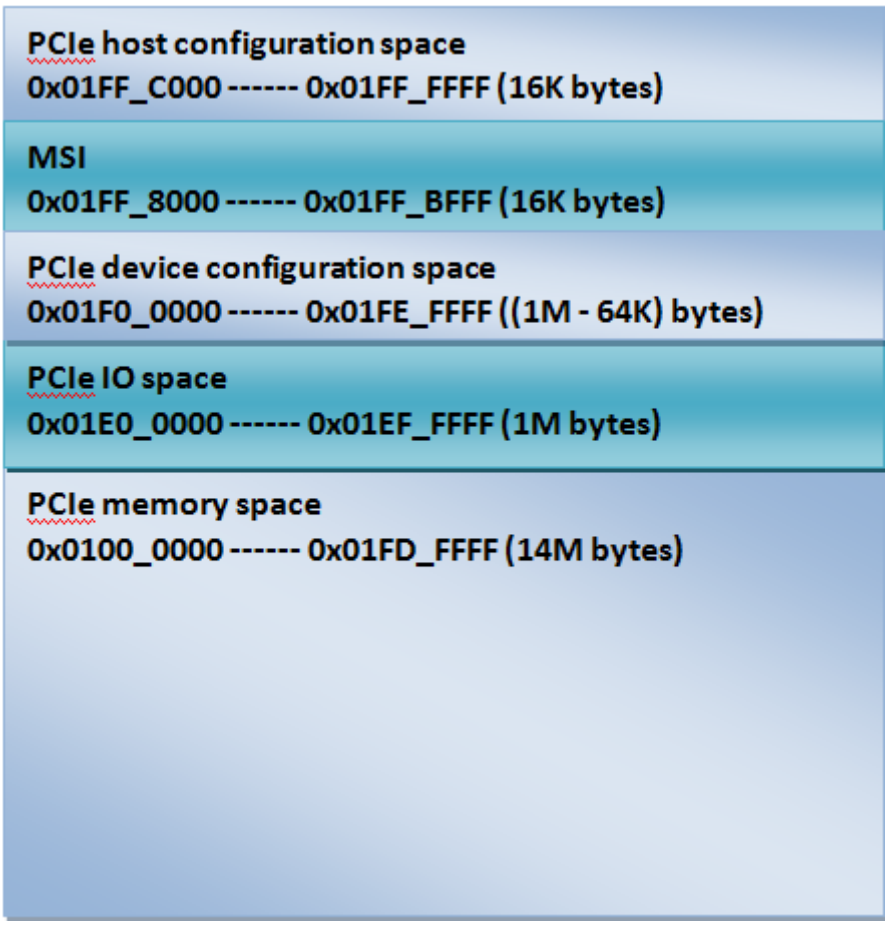


Figure 35-2.

- IO and memory spaces are two address spaces used by the devices to communicate with their device driver running in the Linux kernel on CPU.
- The upper 16Kbytes PCIe host configuration space.
  - This memory segment is used to map the configuration space of PCIe RC. SW can access PCIe RC core configuration space through the DBI interface.
- PCIe device configuration space.
  - Used to map the configuration spaces of PCIe EP devices that are inserted to the RC downstream port.

35.3.1 System Resource: Interrupt lines

i.MX 6 Root Complex driver uses interrupt line 155 for legacy interrupts.

35.4 Using PCIe Endpoint and running Tests

Perform the following steps to use PCIe endpoint and run test:

Configure the driver according to PCIe Endpoint device.

Run "make menuconfig" after run "make ARCH=arm imx6_defconfig".

```
Kernel configuration:
* -> System Type
       -> Freescale MXC Implementations
Select the PCI Express support.
```

Implement the following configurations according to the PCIe EP devices:

- PCIe to USB card driver

```
Symbol: USB_XHCI_HCD [=y]
Type  : tristate
Prompt: xHCI HCD (USB 3.0) support (EXPERIMENTAL)
  Defined at drivers/usb/host/Kconfig:20
  Depends on: USB_SUPPORT [=y] && USB [=y] && PCI [=y] && EXPERIMENTAL [=y]
  Location:
    -> Device Drivers
      -> USB support (USB_SUPPORT [=y])
```

- Intel CT gigabit network card driver

```
Symbol: E1000E [=y]
Type  : tristate
Prompt: Intel(R) PRO/1000 PCI-Express Gigabit Ethernet support
  Defined at drivers/net/Kconfig:2139
  Depends on: NETDEVICES [=y] && NETDEV_1000 [=y] && PCI [=y] && (!SPARC32 || BROKEN [=n])
  Location:
    -> Device Drivers
      -> Network device support (NETDEVICES [=y])
        -> Ethernet (1000 Mbit) (NETDEV_1000 [=y])
```

- Generic IEEE 802.11 Networking Stack (mac80211) used by WIFI devices

```
Symbol: MAC80211 [=y]
Type  : tristate
Prompt: Generic IEEE 802.11 Networking Stack (mac80211)
  Defined at net/mac80211/Kconfig:1
  Depends on: NET [=y] && WIRELESS [=y] && CFG80211 [=y]
  Location:
    -> Networking support (NET [=y])
      -> Wireless (WIRELESS [=y])
```

- Intel iwl4965 or iwl6300 card driver

```
Symbol: IWL4965
[=n]

   Type  :
tristate

   Prompt: Intel Wireless WiFi 4965AGN
(iwl4965)
```

```
     Defined at drivers/net/wireless/iwlegacy/Kconfig:
65
     Depends on: NETDEVICES [=y] && WLAN [=y] && PCI [=n] && MAC80211
[=n]

Location:

       -> Device
Drivers

         -> Network device support (NETDEVICES
[=y])
           -> Wireless LAN (WLAN [=y])
     Selects: IWLWIFI_LEGACY [=n]
```

To enable the wifi driver, we need to enable one of the two options: IWL4965 or IWLAGN. You must choose one, but not both.

```
CONFIG_IWLAGN:


Select to build the driver supporting the:
Intel Wireless WiFi Link Next-Gen AGN
```

This option enables support with the following hardware:

```
      Intel Wireless WiFi Link 6250AGN Adapter
      Intel 6000 Series Wi-Fi Adapters (6200AGN and 6300AGN)
      Intel WiFi Link 1000BGN
      Intel Wireless WiFi 5150AGN
      Intel Wireless WiFi 5100AGN, 5300AGN, and 5350AGN
      Intel 6005 Series Wi-Fi Adapters
      Intel 6030 Series Wi-Fi Adapters
      Intel Wireless WiFi Link 6150BGN 2 Adapter
      Intel 100 Series Wi-Fi Adapters (100BGN and 130BGN)
      Intel 2000 Series Wi-Fi Adapters
```

- WIFI firmware configurations:

In order to install the mandatory required firmware by Intel IWL WIFI devices, please refer to the following link for guidance http://intellinuxwireless.org/?n=Info

35.4.1 Ensuring PCIe System Initialization

Run 'lspci' after login the consol. There should be the following similar message if the PCIe link is established.

root@freescale ~$ lspci

00:00.0 PCI bridge: Unknown device 16c3:abcd (rev 01)

01:00.0 Network controller: Intel Corporation Unknown device 4237

35.4.2 Tests

Run different tests according the different PCIe EP devices.

- Intel Iwl6300 mini-PCIe x1 WIFI card
  - Iperf, netperf
  - Overnight different packet ping
- Intel CT gigabit standard PCIe X1 network card
  - NFS mount/data IO through NFS
  - Iperf, netperf
  - Overnight different packet ping
- PCIe to USB3.0 standard PCIe X1 card
  - General tests
    - * Block storage device, recognization,
    - * Partition creation, format and so on.
    - * Hundreds MB data read/write by copy command
  - Stress tests
    - `./iozone -a -n 2000m -g 2000m -i 0 -i 1 -f /mnt/src/iozone.tmpfile -Rb ./iozone`

35.4.3 Known Issues

- You can connect an external WIFI antenna to enlarge the WIFI signal strength if the WIFI card tests cannot work properly.

35.5 i.MX 6Quad SD PCIe RC/EP Validation System

35.5.1 Hardware Setup

There are two i.MX 6Quad SABRE-SD boards: one is used as PCIe RC; and the other is used as PCIe EP. They are connected by two mini_PCIe-to-standard_PCIe adaptors, two PEX cable adaptors, and then one PCIe cable.

35.5.2 Software Configurations

When building the RC image, make sure that:

```
CONFIG_IMX_PCIE=y
# CONFIG_IMX_PCIE_EP_MODE_IN_EP_RC_SYS is not set
```

```
CONFIG_IMX_PCIE_RC_MODE_IN_EP_RC_SYS=y
```

When building the EP image, make sure that:

```
CONFIG_IMX_PCIE=y
CONFIG_IMX_PCIE_EP_MODE_IN_EP_RC_SYS=y
# CONFIG_IMX_PCIE_RC_MODE_IN_EP_RC_SYS is not set
```

35.5.3 Features

Set up the link between RC and EP by their stand-alone 125MHz running internally.

In the EP system, EP can access the reserved DDR memory (default address: 0x40000000) of the PCIe RC system by the interconnection between PCIe EP and PCIe RC.

NOTE

- The layout of the 1G DDR memory on the SD board is 0x1000_0000-0x4FFF_FFFF). Use mem=768M in the kernel command line to reserve the 0x4000_0000-0x4FFF_FFFF DDR memory space for the EP access test.
- Boot up the PCIe EP system, and then boot up the PCIe RC system.
- Example of the RC kernel command line:

```
noinitrd console=ttymxc0,115200,mem=768M root=/dev/nfs
nfsroot=<your_rootfs> ip=dhcp rw
```

35.5.4 Results

When the ARM core is used as the bus master (define EP_SELF_IO_TEST in pcie.c driver):

Regarding to the log listed in the following table, the data size of each TLP when the cache is enabled, is about 4 times of the data size in write, and 2 times of the data size in read, when the cache is disabled.

	ARM core used as the bus master, and the cache is disabled	ARM core used as the bus master, and the cache is enabled
Data size in one write TLP	8 bytes	32 bytes
Write speed	~109 MB/s	~298 MB/s

Table continues on the next page...

	ARM core used as the bus master, and the cache is disabled	ARM core used as the bus master, and the cache is enabled
Data size in one read TLP	32 bytes	64 bytes
Read speed	~29 MB/s	~100 MB/s

When the cache is enabled:

```
                PCIe EP: Starting data transfer...
                PCIe EP: Data transfer is successful, tv_count1 54840us, tv_count2
162814us.
                PCIe EP: Data write speed is 298 MB/s.
                PCIe EP: Data read speed is 100 MB/s.
```

The snapshot of the analyzer log is as follows:

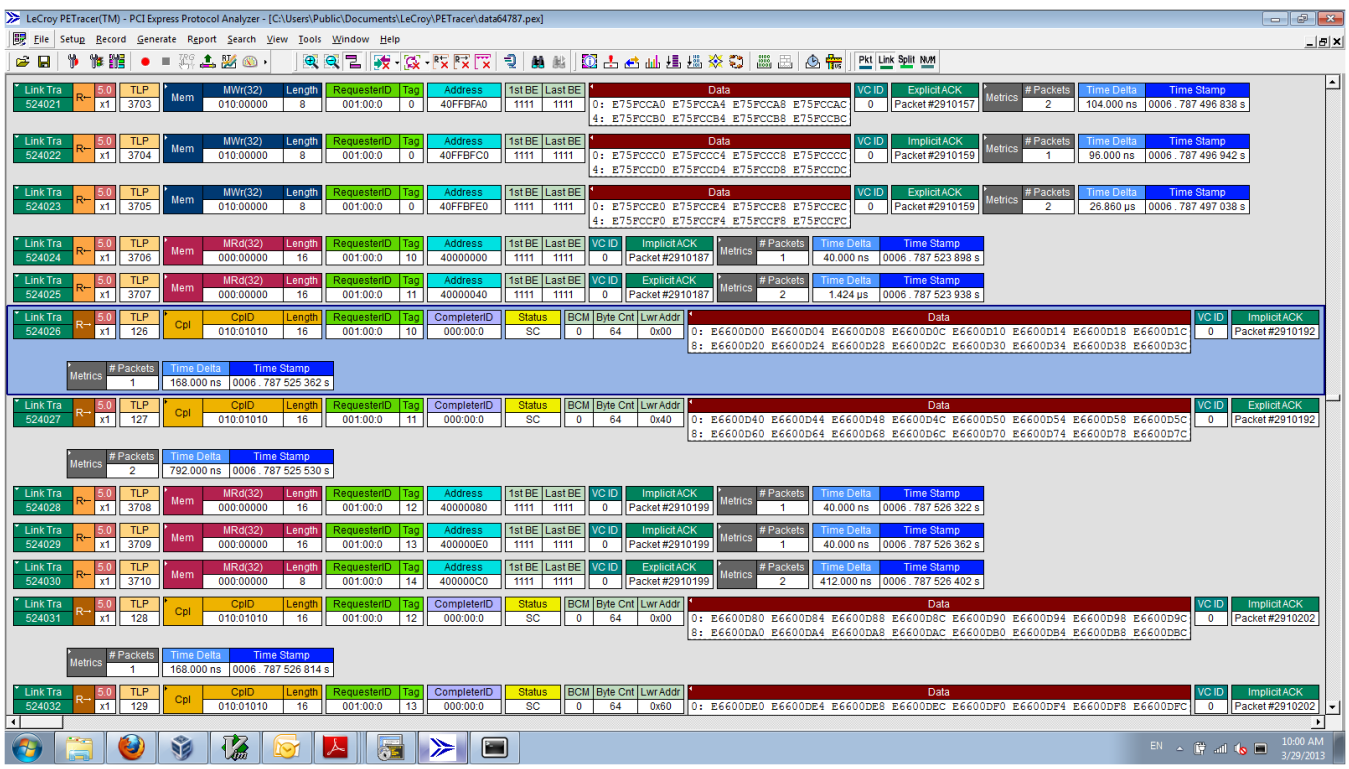


Figure 35-3. Analyzer log for enabled cache

When the cache is disable:

```
                PCIe EP: Starting data transfer...
                PCIe EP: Data transfer is successful, tv_count1 149616us, tv_count2
552099us.
                PCIe EP: Data write speed is 109 MB/s.
                PCIe EP: Data read speed is 29 MB/s.
```

The snapshot of the write/read log is as follows:

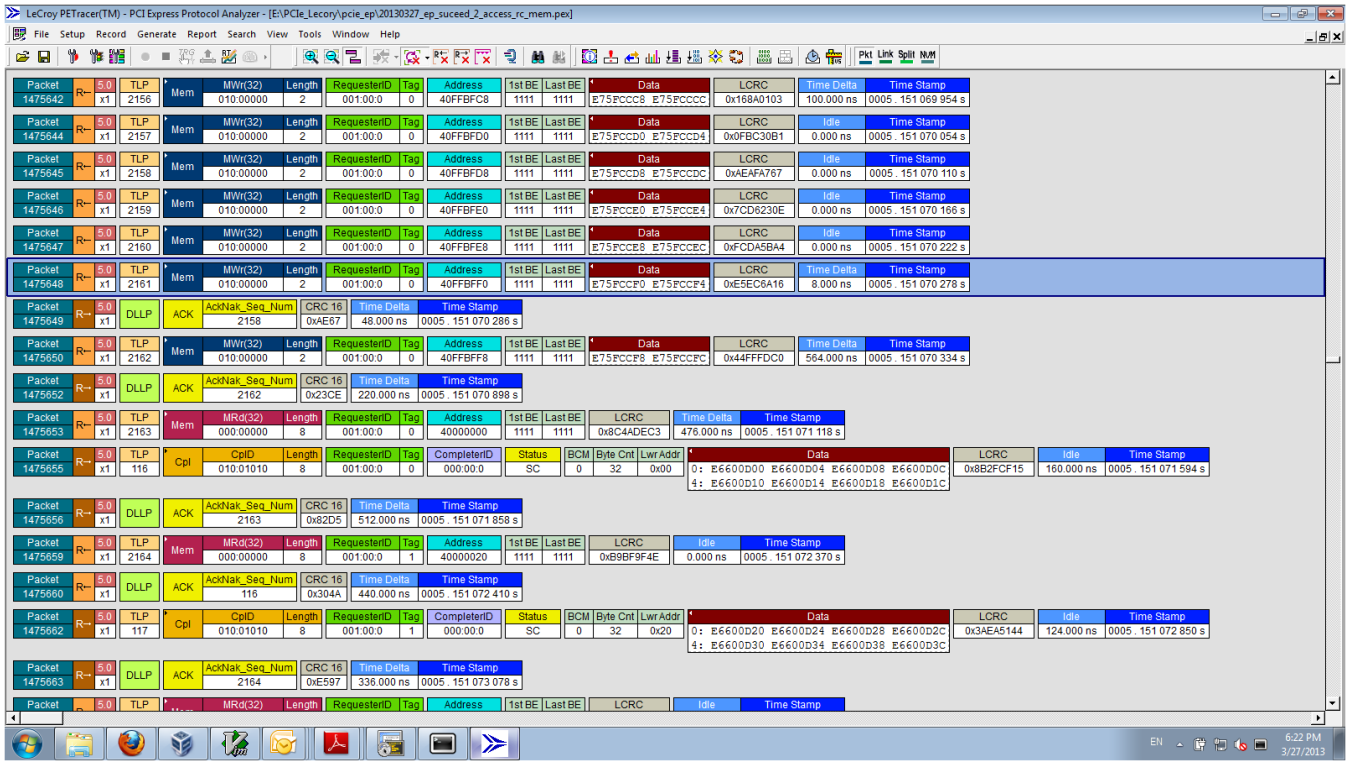


Figure 35-4. Write/Read log for disabled cache

Chapter 36
WEIM NOR Driver

36.1 Introduction

The Wireless External Interface Module (WEIM) NOR driver supports the Parallel NOR flash.

36.2 Hardware Operation

By default, there is a parallel NOR in the i.MX 6Quad/6Dual SABRE-AI boards. The parallel NOR has more pins than the SPI NOR. On some boards, the M29W256GL7AN6E is equipped. Refer to the datasheet for details on the parallel NOR.

36.3 Software Operation

Similar to the SPI NOR, the parallel NOR uses the MTD subsystem. Because the parallel NOR is very small, you may only use the jffs2 but cannot use the UBIFS for it.

36.4 Source Code

To set the proper timing only for the parallel NOR, refer to `mx6q_setup_weimcs()` in arch/arm/mach-mx6/board-mx6q_sabreauto.c.

36.5 Enabling the WEIM NOR

Add `weim-nor` to the kernel command line to enable the WEIM NOR. The WEIM NOR has pin conflict with some other modules, such as the SPI.

Chapter 37
Fast Ethernet Controller (FEC) Driver

37.1 Introduction

The Fast Ethernet Controller (FEC) driver performs the full set of IEEE 802.3/Ethernet CSMA/CD media access control and channel interface functions.

The FEC requires an external interface adapter and transceiver function to complete the interface to the Ethernet media. It supports half or full-duplex operation on 10 Mbps, 100 Mbps or 1000 Mbps related Ethernet networks.

The FEC driver supports the following features:

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

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

37.2 Hardware Operation

The FEC is an Ethernet controller that interfaces the system to the LAN network.

The FEC supports different standard MAC-PHY (physical) interfaces for connection to an external Ethernet transceiver. The FEC supports the 10/100 Mbps MII, and 10/100 Mbps RMII. In addition, the FEC supports 1000 Mbps RGMII, which uses 4-bit reduced GMII operating at 125 MHz.

A brief overview of the device functionality is provided here. For details see the FEC chapter of the *i.MX 6 Multimedia Applications Processor Reference Manual*.

In MII mode, there are 18 signals defined by the IEEE 802.3 standard and supported by the EMAC. MII, RMII and RGMII modes uses a subset of the 18 signals. These signals are listed in table below.

Table 37-1. Pin Usage in MII, RMII and RGMII Modes

Direction	EMAC Pin Name	MII Usage	RMII Usage	RGMII Usage
In/Out	FEC_MDIO	Management Data Input/Output	Management Data Input/output	Management Data Input/Output
Out	FEC_MDC	Management Data Clock	General output	Management Data Clock
Out	FEC_TXD[0]	Data out, bit 0	Data out, bit 0	Data out, bit 0
Out	FEC_TXD[1]	Data out, bit 1	Data out, bit 1	Data out, bit 1
Out	FEC_TXD[2]	Data out, bit 2	Not Used	Data out, bit 2
Out	FEC_TXD[3]	Data out, bit 3	Not Used	Data out, bit 3
Out	FEC_TX_EN	Transmit Enable	Transmit Enable	Transmit Enable
Out	FEC_TX_ER	Transmit Error	Not Used	Not Used
In	FEC_CRS	Carrier Sense	Not Used	Not Used
In	FEC_COL	Collision	Not Used	Not Used
In	FEC_TX_CLK	Transmit Clock	Not Used	Synchronous clock reference (REF_CLK, can connect from PHY)
In	FEC_RX_ER	Receive Error	Receive Error	Not Used
In	FEC_RX_CLK	Receive Clock	Not Used	Synchronous clock reference (REF_CLK, can connect from PHY)
In	FEC_RX_DV	Receive Data Valid	Receive Data Valid and generate CRS	RXDV XOR RXERR on the falling edge of FEC_RX_CLK.
In	FEC_RXD[0]	Data in, bit 0	Data in, bit 0	Data in, bit 0
In	FEC_RXD[1]	Data in, bit 1	Data in, bit 1	Data in, bit 1
In	FEC_RXD[2]	Data in, bit 2	Not Used	Data in, bit 2
In	FEC_RXD[3]	Data in, bit 3	Not Used	Data in, bit 3

The MII management interface consists of two pins, FEC_MDIO, and FEC_MDC. The FEC hardware operation can be divided in the parts listed below. For detailed information consult the *i.MX 6 Multimedia Applications Processor Reference Manual*.

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

- Before transmitting, the controller waits for carrier sense to become inactive, then determines if carrier sense stays inactive for 60 bit times. If the transmission begins after waiting an additional 36 bit times (96 bit times after carrier sense originally became inactive), both buffer (TXB) and frame (TXF) interrupts may be generated as determined by the settings in the EIMR.
- Reception-The FEC receiver is designed to work with almost no intervention from the host and can perform address recognition, CRC checking, short frame checking, and maximum frame length checking. When the driver enables the FEC receiver by asserting ECR[ETHER_EN], it immediately starts processing receive frames. When FEC_RX_DV asserts, the receiver checks for a valid PA/SFD header. If the PA/SFD is valid, it is stripped and the frame is processed by the receiver. If a valid PA/SFD is not found, the frame is ignored. In MII mode, the receiver checks for at least one byte matching the SFD. Zero or more PA bytes may occur, but if a 00 bit sequence is detected prior to the SFD byte, the frame is ignored.
- After the first six bytes of the frame have been received, the FEC performs address recognition on the frame. During reception, the Ethernet controller checks for various error conditions and once the entire frame is written into the FIFO, a 32-bit frame status word is written into the FIFO. This status word contains the M, BC, MC, LG, NO, CR, OV, and TR status bits, and the frame length. Receive Buffer (RXB) and Frame Interrupts (RXF) may be generated if enabled by the EIMR register. When the receive frame is complete, the FEC sets the L bit in the RxBD, writes the other frame status bits into the RxBD, and clears the E bit. The Ethernet controller next generates a maskable interrupt (RXF bit in EIR, maskable by RXF bit in EIMR), indicating that a frame has been received and is in memory. The Ethernet controller then waits for a new frame.
- Interrupt management-When an event occurs that sets a bit in the EIR, an interrupt is generated if the corresponding bit in the interrupt mask register (EIMR) is also set. The bit in the EIR is cleared if a one is written to that bit position; writing zero has no effect. This register is cleared upon hardware reset. These interrupts can be divided into operational interrupts, transceiver/network error interrupts, and internal error interrupts. Interrupts which may occur in normal operation are GRA, TXF, TXB, RXF, RXB. Interrupts resulting from errors/problems detected in the network or transceiver are HBERR, BABR, BABT, LC, and RL. Interrupts resulting from internal errors are HBERR and UN. Some of the error interrupts are independently counted in the MIB block counters. Software may choose to mask off these interrupts as these errors are visible to network management through the MIB counters.
- PHY management-phylib was used to manage all the FEC phy related operation such as phy discovery, link status, and state machine.MDIO bus will be created in FEC driver and registered into the system.You can refer to Documentation/networking/phy.txt under linux source directory for more information.

37.2.1 Software Operation

The FEC driver supports the following functions:

- Module initialization-Initializes the module with the device specific structure
- Rx/Tx transmition
- Interrupt servicing routine
- PHY management
- FEC management such init/start/stop
- i.MX 6 FEC module use little-endian format

37.2.2 Source Code Structure

Table below shows the source files.

They are available in the

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

Table 37-2. FEC Driver Files

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

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

37.2.3 Menu Configuration Options

To get to the Linux kernel configuration option provided for this module, use the ./ltib -c command when located in the <ltib dir>.

On the screen displayed, select **Configure the Kernel** and exit. When the next screen appears, select the following option to enable this module:

- CONFIG_FEC is provided for this module. This option is available under:
  - Device Drivers > Network device support > Ethernet (10, 100 or 1000 Mbit) > FEC Ethernet controller.
  - To mount NFS-rootfs through FEC, disable the other Network config in the menuconfig if need.

37.3 Programming Interface

Table 37-2 lists the source files for the FEC driver.

The following section shows the modifications that were required to the original Ethernet driver source for porting it to the i.MX device.

37.3.1 Device-Specific Defines

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

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

```
/*
 *      Define the buffer descriptor structure.
 */
struct bufdesc {
          unsigned short            cbd_datlen;       /* Data length */
          unsigned short            cbd_sc;           /* Control and status info */
          unsigned long             cbd_bufaddr;      /* Buffer address */
#ifdef CONFIG_ENHANCED_BD
          unsigned long cbd_esc;
          unsigned long cbd_prot;
          unsigned long cbd_bdu;
          unsigned long ts;
          unsigned short res0[4];
#endif
};
/*
 *      Define the register access structure.
 */
#define FEC_IEVENT              0x004 /* Interrupt event reg */
#define FEC_IMASK               0x008 /* Interrupt mask reg */
#define FEC_R_DES_ACTIVE        0x010 /* Receive descriptor reg */
#define FEC_X_DES_ACTIVE        0x014 /* Transmit descriptor reg */
#define FEC_ECNTRL              0x024 /* Ethernet control reg */
#define FEC_MII_DATA            0x040 /* MII manage frame reg */
#define FEC_MII_SPEED           0x044 /* MII speed control reg */
#define FEC_MIB_CTRLSTAT        0x064 /* MIB control/status reg */
#define FEC_R_CNTRL             0x084 /* Receive control reg */
#define FEC_X_CNTRL             0x0c4 /* Transmit Control reg */
#define FEC_ADDR_LOW            0x0e4 /* Low 32bits MAC address */
#define FEC_ADDR_HIGH           0x0e8 /* High 16bits MAC address */
#define FEC_OPD                 0x0ec /* Opcode + Pause duration */
#define FEC_HASH_TABLE_HIGH     0x118 /* High 32bits hash table */
#define FEC_HASH_TABLE_LOW      0x11c /* Low 32bits hash table */
#define FEC_GRP_HASH_TABLE_HIGH 0x120 /* High 32bits hash table */
#define FEC_GRP_HASH_TABLE_LOW  0x124 /* Low 32bits hash table */
#define FEC_X_WMRK              0x144 /* FIFO transmit water mark */
#define FEC_R_BOUND             0x14c /* FIFO receive bound reg */
#define FEC_R_FSTART            0x150 /* FIFO receive start reg */
#define FEC_R_DES_START         0x180 /* Receive descriptor ring */
#define FEC_X_DES_START         0x184 /* Transmit descriptor ring */
#define FEC_R_BUFF_SIZE         0x188 /* Maximum receive buff size */
```

```
#define FEC_MIIGSK_CFGR         0x300 /* MIIGSK config register */
#define FEC_MIIGSK_ENR          0x308 /* MIIGSK enable register */
```

37.3.2 Getting a MAC Address

The following statement gets the MAC address through the OCOTP (IC Identification) by default for i.MX 6.

The MAC address can be set through bootloader such as u-boot. FEC driver will use it to confiure the MAC address for network devices. i.MX 6 user needs to provide MAC address by kernel command line so that user can use sb_loader to load kernel and run it without bootloader interaction.

Due to certain pin conflicts (FEC RMII mode need to use GPIO_16 or RGMII_TX_CTL pin as reference clock input/output channel), the one of the both pins cannot connect to branch lines for other modules use because the branch lines have serious influence on clock.

Chapter 38
ENET IEEE-1588 Driver

38.1 Hardware Operation

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.

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'.

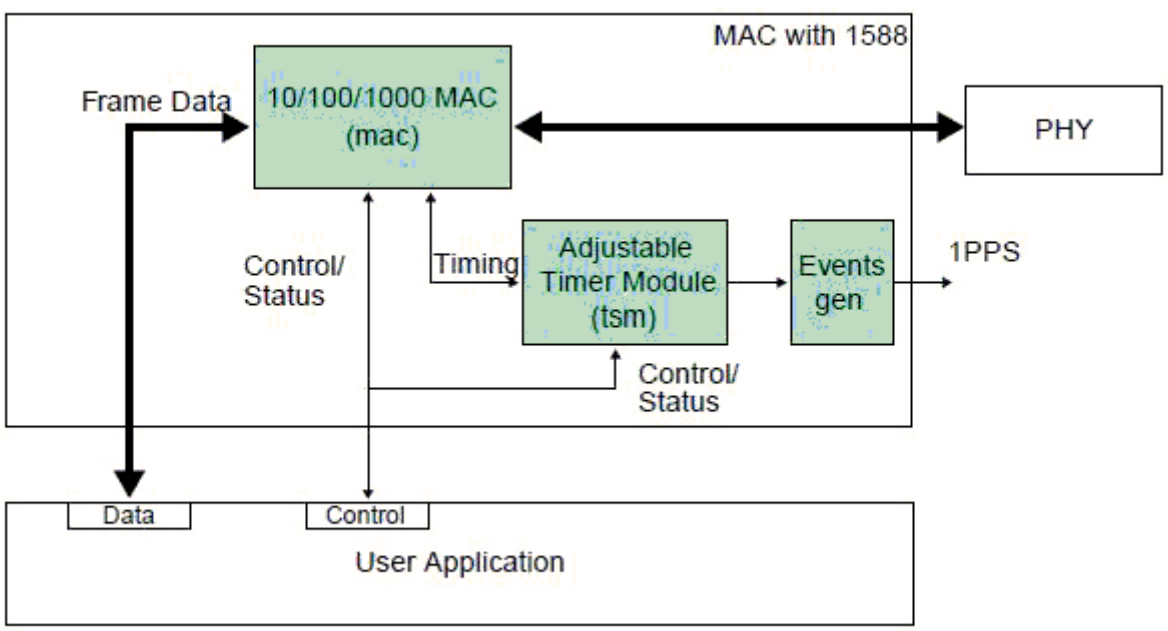


Figure 38-1. IEEE 1588 Functions Overview

38.1.1 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.

38.1.2 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.

38.2 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.

38.2.1 Source Code Structure

Table below lists the source files available in the <ltib_dir>/rpm/BUILD/linux/drivers/net directory.

Table 38-1. 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.

38.2.2 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.

38.3 Programming Interface

The 1588 driver complies with the IXXAT protocol stack interface.

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

38.3.1 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_Req 0x2
#define DEFAULT_msg_Peer_Delay_Resp 0x3
/* PTP message version */
#define PTP_1588_MSG_VER_1 1
#define PTP_1588_MSG_VER_2 2
```

38.3.2 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 39
Universal Asynchronous Receiver/Transmitter (UART) Driver

39.1 Introduction

The low-level UART driver interfaces the Linux serial driver API to all the UART ports.

It has the following features:

- Interrupt-driven and SDMA-driven transmit/receive of characters
- Standard Linux baud rates up to 4 Mbps
- Transmit and receive characters with 7-bit and 8-bit character lengths
- Transmits one or two stop bits
- Supports TIOCMGET IOCTL to read the modem control lines. Only supports the constants TIOCM_CTS and TIOCM_CAR, plus TIOCM_RI in DTE mode only
- Supports TIOCMSET IOCTL to set the modem control lines. Supports the constants TIOCM_RTS and TIOCM_DTR only
- Odd and even parity
- XON/XOFF software flow control. Serial communication using software flow control is reliable when communication speeds are not too high and the probability of buffer overruns is minimal
- CTS/RTS hardware flow control-both interrupt-driven software-controlled hardware flow and hardware-driven hardware-controlled flow
- Send and receive break characters through the standard Linux serial API
- Recognizes frame and parity errors
- Ability to ignore characters with break, parity and frame errors
- Get and set UART port information through the TIOCGSSERIAL and TIOCSSERIAL TTY IOCTL. Some programs like setserial and dip use this feature to make sure that the baud rate was set properly and to get general information on the device. The UART type should be set to 52 as defined in the serial_core.h header file.
- Serial IrDA

- Power management feature by suspending and resuming the URT ports
- Standard TTY layer IOCTL calls

All the UART ports can be accessed from the device files /dev/ttymxc0 to /dev/ttymxc1. Autobaud detection is not supported.

NOTE

If you want to use the DMA support for UART please also enable the RTS/CTS for it. The DMA may be abnormal if you do not enable the RTS/CTS.

39.2 Hardware Operation

Refer to the *i.MX 6Dual/6Quad Applications Processor Reference Manual* to determine the number of UART modules available in the device.

Each UART hardware port is capable of standard RS-232 serial communication and has support for IrDA 1.0.

Each UART contains a 32-byte transmitter FIFO and a 32-half-word deep receiver FIFO. Each UART also supports a variety of maskable interrupts when the data level in each FIFO reaches a programmed threshold level and when there is a change in state in the modem signals. Each UART can be programmed to be in DCE or DTE mode.

39.2.1 Software Operation

The Linux OS contains a core UART driver that manages many of the serial operations that are common across UART drivers for various platforms.

The low-level UART driver is responsible for supplying information such as the UART port information and a set of control functions to the core UART driver. These functions are implemented as a low-level interface between the Linux OS and the UART hardware. They cannot be called from other drivers or from a user application. The control functions used to control the hardware are passed to the core driver through a structure called uart_ops, and the port information is passed through a structure called uart_port. The low level driver is also responsible for handling the various interrupts for the UART ports, and providing console support if necessary.

Each UART can be configured to use DMA for the data transfer. These configuration options are provided in the mxc_uart.h header file. The user can specify the size of the DMA receive buffer. The minimum size of this buffer is 512 bytes. The size should be a multiple of 256. The driver breaks the DMA receive buffer into smaller sub-buffers of

256 bytes and registers these buffers with the DMA system. DMA transmit buffer size is fixed at 1024 bytes. The size is limited by the size of the Linux UART transmit buffer (1024).

The driver requests two DMA channels for the UARTs that need DMA transfer. On a receive transaction, the driver copies the data from the DMA receive buffer to the TTY Flip Buffer.

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

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

39.2.2 Driver Features

The UART driver supports the following features:

- Baud rates up to 4 Mbps
- Recognizes frame and parity errors only in interrupt-driven mode; does not recognize these errors in DMA-driven mode
- Sends, receives, and appropriately handles break characters
- Recognizes the modem control signals
- Ignores characters with frame, parity, and break errors if requested to do so
- Implements support for software and hardware flow control (software-controlled and hardware-controlled)
- Get and set the UART port information; certain flow control count information is not available in hardware-driven hardware flow control mode
- Implements support for Serial IrDA
- Power management
- Interrupt-driven and DMA-driven data transfer

39.2.3 Source Code Structure

Table below shows the UART driver source files that are available in the directory:

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

Table 39-1. UART Driver Files

File	Description
imx.c	Low level driver

Table below shows the header files associated with the UART driver.

Table 39-2. UART Global Header Files

File	Description
<ltib_dir>/rpm/BUILD/linux/ arch/arm/plat-mxc/ include/mach/imx-uart.h	UART header that contains UART configuration data structure definitions

39.3 Configuration

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

39.3.1 Menu Configuration Options

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

- CONFIG_SERIAL_IMX -Used for the UART driver for the UART ports. In menuconfig, this option is available under

  Device Drivers > Character devices > Serial drivers > IMX serial port support.

  By default, this option is Y.

- CONFIG_SERIAL_IMX_CONSOLE-Chooses the Internal UART to bring up the system console. This option is dependent on the CONFIG_SERIAL_IMX option. In the menuconfig this option is available under

  Device Drivers > Character devices > Serial drivers > IMX serial port support > Console on IMX serial port

  By default, this option is Y.

39.3.2 Source Code Configuration Options

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

39.3.3 Chip Configuration Options

39.3.4 Board Configuration Options

For i.MX 6Quad, the board specific configuration options for the driver is set within:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx6/board-mx6q_arm2.c
```

39.4 Programming Interface

The UART driver implements all the methods required by the Linux serial API to interface with the UART port.

The driver implements and provides a set of control methods to the Linux core UART driver. For more information about the methods implemented in the driver, see the API document.

39.4.1 Interrupt Requirements

The UART driver interface generates only one interrupt.

The status is used to determine which kinds of interrupt occurs, such as RX or TX.

With the SDMA enabled, the DMA RX interrupt occurs only when the received data fills all the 4K buffer. The DMA TX interrupt occurs when the data is sent out.

Chapter 40
AR6003 WiFi

40.1 Hardware Operation

The officially supported WiFi chip with FSL BSP is AR6003 from Atheros.

The Atheros AR6003 is a single chip, small form factor IEEE 802.11 a/b/g/n MAC/ baseband/ radio optimized for low-power mobile applications.

40.1.1 Software Operation

FSL BSP uses the open source ath6kl driver from kernel 3.0.35 for AR6003.

40.1.2 Driver features

AR6003 is a single stream, SDIO based 802.11 chipset from Atheros optimized for mobile and embedded devices. ath6kl is a cfg80211 driver for AR6003 and supports both the station and AP mode of operation.

Station mode supports 802.11 a/b/g/n with HT20 on 2.4/5GHz and HT40 only on 5GHz. Some of the other features include WPA/WPA2,WPS, WMM, WMM-PS, and BT coexistence. AP mode can be operated only in b/g mode with support for a subset of features mentioned above.

The driver supports cfg80211 but comes with its own set of wext ioctls which have historically supported some of our customers with features like BT 3.0 and AP mode of operation.

For further details, refer to http://wireless.kernel.org/en/users/Drivers/ath6kl

The driver requires firmware that runs on the chip's network processor. The majority of it is stored in ROM. The binaries that are downloaded and executed from RAM are as follows:

1) Patch against the code in ROM for bug fixes and feature enhancements.

2) Code to copy the data from the OTP region of the memory into RAM.

3) Calibration file carrying board specific data.

The above files need to be present in the directory '/lib/firmware/ath6k/AR6003/hw2.0/' for the driver to initialize the chip upon enumeration. The files can be downloaded from the link specified at the following location http://wireless.kernel.org/en/users/Drivers/ath6kl

This driver is only provided in the interim while we work on the mac80211 replacement, ath6k. Once the mac80211 driver achieves feature parity with the ath6kl driver, the ath6kl will be deprecated and removed from staging.

40.1.3 Source Code Structure

The AR6003 driver source files are available in the directory, <ltib_dir>/rpm/BUILD/linux/drivers/staging/ath6kl/

40.1.4 Linux Menu Configuration Options

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

CONFIG_ATH6K_LEGACY-Build support for AR6003 support (non mac80211).

Note: There are also a few other options under CONFIG_ATH6K_LEGACY. By default you may not need to use them. Refer to the option help for details.

Chapter 41
Bluetooth Driver

41.1 Introduction

The Bluetooth driver provides synchronous and asynchronous wireless connection among multiple devices.

The synchronous oriented channel provides voice transmission. The asynchronous channel allows more time delay in data transmission. The synchronous and asynchronous data transfer between the host and Bluetooth chip is performed by different hardware interfaces. The SSI interface is used to transfer voice from the host to the Bluetooth chip. UART or USB is used for asynchronous data communication.

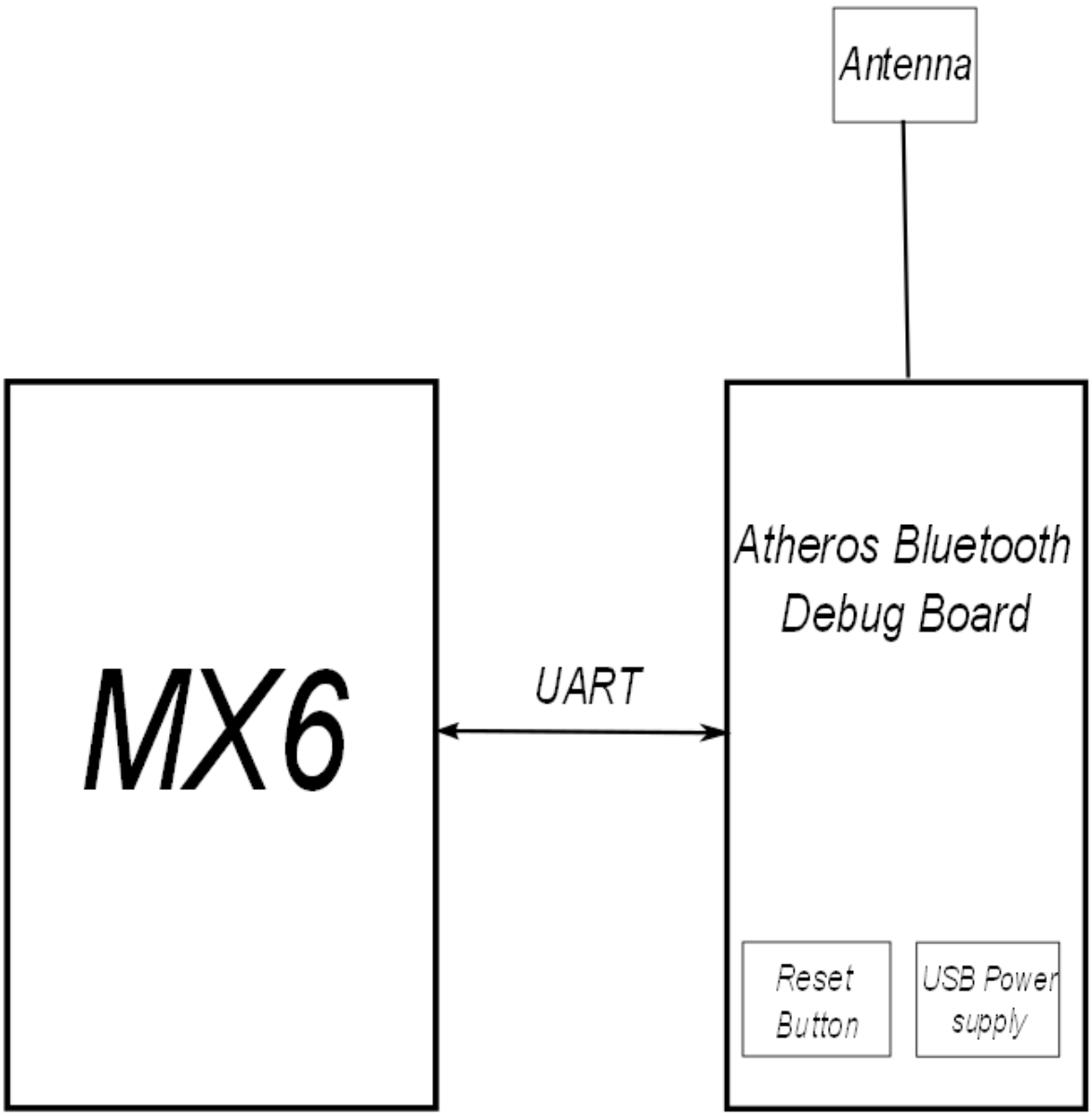
Based on the wireless connection, many services can be supported by profiles defined by the Bluetooth Group. On the i.MX platform, the A2DP and AVRCP profile is used to play music (mp3, wav, and so forth). The FTP profile provides access to the file system on another device. The SPP profile emulates a serial cable to provide a simply implemented wireless replacement for the existing RS-232 based serial communications applications. The handset profile is reserved for future support, so the SSI interface is reserved. The UART interface is used for communication between the host and the Bluetooth chip.

41.1.1 Hardware Operation

The platform uses the Atheros Bluetooth debug board.

Atheros Bluetooth debug board is a Bluetooth module that integrates Atheros Bluetooth soc on it with a mini usb port used to get power supply from external USB. Also there is a reset button on the board which is used to give a hardware reset to the SoC core.

Figure below illustrates the hardware interface between i.MX 6 and the Atheros Bluetooth module. UART is used for data communication.


Antenna
MX6
UART
Atheros Bluetooth
Debug Board
Reset
Button
USB Power
supply

41.2 Software Operation

BlueCore™ Host Software (BCHS) is a Bluetooth protocol provided by a third-party company, Cambridge Silicon Radio (CSR).

The porting of BCHS to Linux is divided into:

- A **user space port**, in which the BCHS protocol stack runs in user space together with the application.
- A **kernel space port**, in which the BCHS protocol stack runs in kernel space and the application runs in user space.

There are two ways to set up the user space port:

- The application and the BCHS protocol stack are running within the same process.
- The application and the BCHS protocol stack are running in two different processes.

In i.MX platform, the BCHS protocol stack runs in user space. And the application runs in the same process, as shown in figure below.

Encoding is used to minimize the bandwidth required for transferring the audio data. Thus, the encoding compresses the audio before transmission over the air. The A2DP profile mandates support for SBC encoding, and other codecs, such as MP3 and WMA, are optional. The A2DP source checks the capabilities of sink and then configures sink to select the dedicated codec.

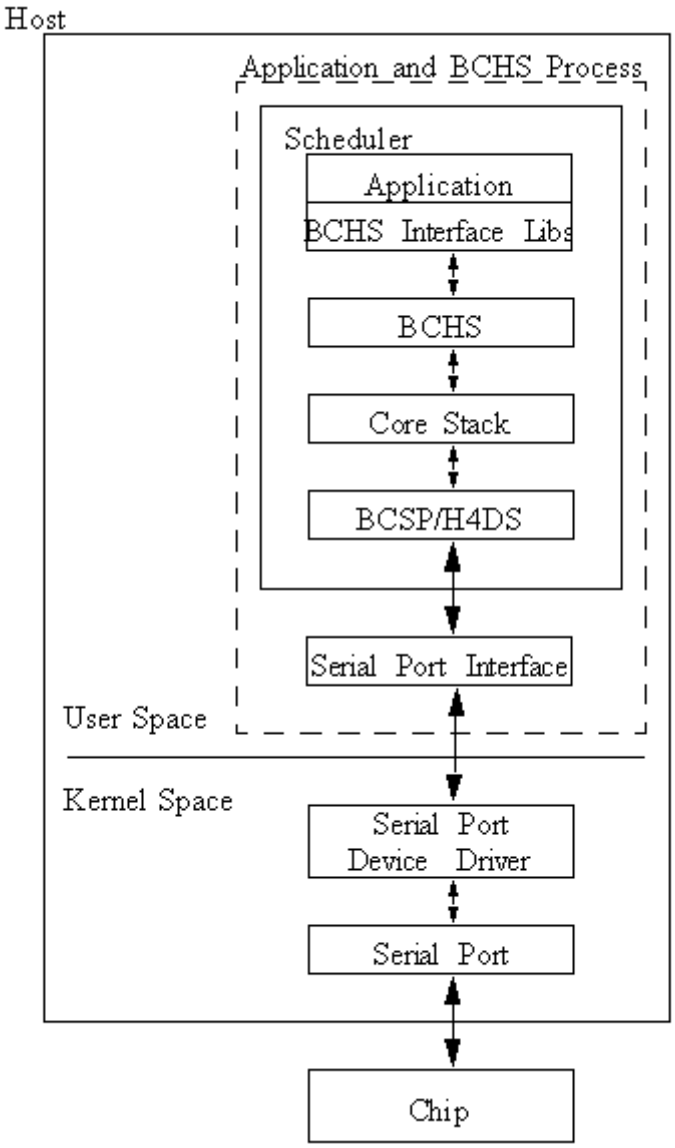


Figure 41-2. BCHS Protocol Stack

41.2.1 UART Control

For user space porting, first configure the universal asynchronous receiver transmitter (UART). On the i.MX platforms, UART2 is used for communication between the CPU chip and the Bluetooth chip. The BCHS protocol opens /dev/ttymxc1 and configures the device according to profile requirements.

The minimum baud rate for the A2DP profile is 460.8 kbps; 921.6 kbps baudrate is recommended. Table below maps the relationship between the UART baud rate and maximum SBC bit rate.

Table 41-1. UART Mapping

Baud Rate (kbps)	Max SBC bit rate (kbps)
115.2	75
230.4	150
460.8	300
600.0	400
921.6	410

The following table describes the UART configuration files.

41.2.2 Reset and Power control

Currently we use an external bluetooth debug board for the bluetooth communication, The bt module needs an usb cable connected with it to get the power supply, also in the debug board there is a reset button used to reset the whole bt module.

41.2.3 Configuration

To get to the Bluetooth configuration, use the command ./ltib -c when located in the <ltib dir>. In the screen, select Configure Kernel, exit, and a new screen will appear.

The Linux kernel configuration option CONFIG_MXC_BLUETOOTH is provided for the MXC processors. In the menuconfig this option is available under Device Drivers –> MXC support drivers -> MXC Bluetooth support -> MXC Bluetooth support. By default, this option is M for all architectures.

Chapter 42
Pulse-Width Modulator (PWM) Driver

42.1 Introduction

The pulse-width modulator (PWM) has a 16-bit counter and is optimized to generate sound from stored sample audio images and generate tones.

The PWM has 16-bit resolution and uses a 4x16 data FIFO to generate sound. The software module is composed of a Linux driver that allows privileged users to control the backlight by the appropriate duty cycle of the PWM Output (PWMO) signal.

42.1.1 Hardware Operation

Figure below shows the PWM block diagram.

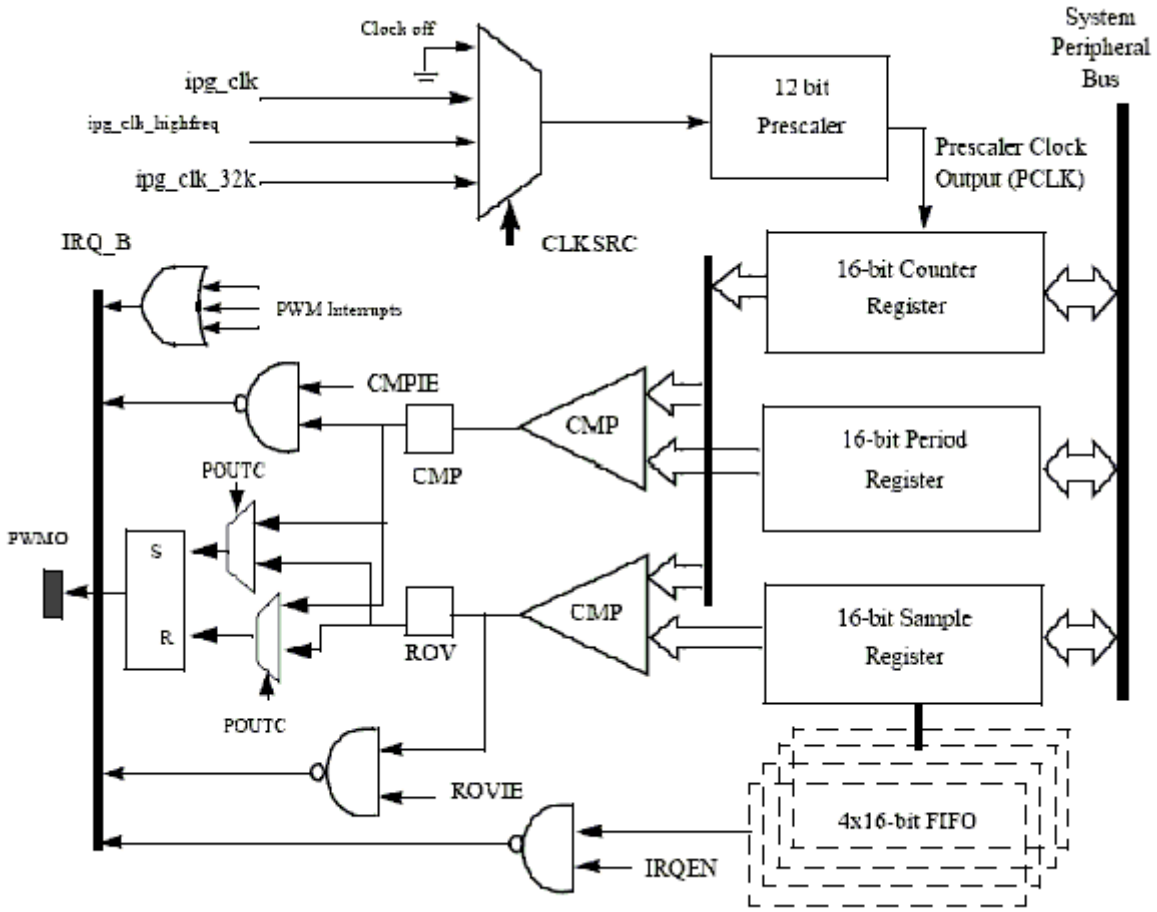


Figure 42-1. PWM Block Diagram

The PWM follows IP Bus protocol for interfacing with the processor core. It does not interface with any other modules inside the device except for the clock and reset inputs from the Clock Control Module (CCM) and interrupt signals to the processor interrupt handler. The PWM includes a single external output signal, PMWO. The PWM includes the following internal signals:

- Three clock inputs
- Four interrupt lines
- One hardware reset line
- Four low power and debug mode signals
- Four scan signals
- Standard IP slave bus signals

42.1.2 Clocks

The clock that feeds the prescaler can be selected from:

- High frequency clock-provided by the CCM. The PWM can be run from this clock in low power mode.
- Low reference clock-32 KHz low reference clock provided by the CCM. The PWM can be run from this clock in the low power mode.
- Global functional clock-for normal operations. In low power modes this clock can be switched off.

The clock input source is determined by the CLKSRC field of the PWM control register. The CLKSRC value should only be changed when the PWM is disabled.

42.1.3 Software Operation

The PWM device driver reduces the amount of power sent to a load by varying the width of a series of pulses to the power source. One common and effective use of the PWM is controlling the backlight of a QVGA panel with a variable duty cycle.

Table below provides a summary of the interface functions in source code.

Table 42-1. PWM Driver Summary

Function	Description
struct pwm_device *pwm_request(int pwm_id, const char *label)	Request a PWM device
void pwm_free(struct pwm_device *pwm)	Free a PWM device
int pwm_config(struct pwm_device *pwm, int duty_ns, int period_ns)	Change a PWM device configuration
int pwm_enable(struct pwm_device *pwm)	Start a PWM output toggling
int pwm_disable(struct pwm_device *pwm)	Stop a PWM output toggling

The function pwm_config() includes most of the configuration tasks for the PWM module, including the clock source option, and period and duty cycle of the PWM output signal. It is recommended to select the peripheral clock of the PWM module, rather than the local functional clock, as the local functional clock can change.

42.1.4 Driver Features

The PWM driver includes the following software and hardware support:

- Duty cycle modulation
- Varying output intervals
- Two power management modes-full on and full of

42.1.5 Source Code Structure

Table below lists the source files and headers available in the following directories:

```
<ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/pwm.c
<ltib_dir>/rpm/BUILD/linux/include/linux/pwm.h
```

Table 42-2. PWM Driver Files

File	Description
pwm.h	Functions declaration
pwm.c	Functions definition

42.1.6 Menu Configuration Options

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

- System Type > Enable PWM driver
- Select the following option to enable the Backlight driver:

  Device Drivers > Graphics support > Backlight & LCD device support > Generic PWM based Backlight Driver

Chapter 43
Watchdog (WDOG) Driver

43.1 Introduction

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

Some platforms may have two WDOG modules with one of them having interrupt capability.

43.1.1 Hardware Operation

Once the WDOG timer is activated, it must be serviced by software on a periodic basis.

If servicing does not take place in time, the WDOG times out. Upon a time-out, the WDOG either asserts the wdog_b signal or a wdog_rst_b system reset signal, depending on software configuration. The watchdog module cannot be deactivated once it is activated.

43.1.2 Software Operation

The Linux OS has a standard WDOG interface that allows support of a WDOG driver for a specific platform.

WDOG can be suspended/resumed in STOP/DOZE and WAIT modes independently. Since some bits of the WGOD registers are only one-time programmable after booting, ensure these registers are written correctly.

43.2 Generic WDOG Driver

The generic WGOD driver is implemented in the <ltib_dir>/rpm/BUILD/linux/drivers/watchdog/imx2_wdt.c file.

It provides functions for various IOCTLs and read/write calls from the user level program to control the WDOG.

43.2.1 Driver Features

This WDOG implementation includes the following features:

- Generates the reset signal if it is enabled but not serviced within a predefined timeout value (defined in milliseconds in one of the WDOG source files)
- Does not generate the reset signal if it is serviced within a predefined timeout value
- Provides IOCTL/read/write required by the standard WDOG subsystem

43.2.2 Menu Configuration Options

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

- CONFIG_IMX2_WDT-Enables Watchdog timer module. This option is available under Device Drivers > Watchdog Timer Support > IMX2+ Watchdog.

43.2.3 Source Code Structure

Table below shows the source files for WDOG drivers that are in the following directory:

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

Table 43-1. WDOG Driver Files

File	Description
imx2_wdt.c	WDOG function implementations

Watchdog system reset function is located under <ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxc/system.c

43.2.4 Programming Interface

The following IOCTLs are supported in the WDOG driver:

- WDIOC_GETSUPPORT
- WDIOC_GETSTATUS
- WDIOC_GETBOOTSTATUS
- WDIOC_KEEPALIVE
- WDIOC_SETTIMEOUT
- WDIOC_GETTIMEOUT

For detailed descriptions about these IOCTLs, see <ltib_dir>/rpm/BUILD/linux/Documentation/watchdog.

Chapter 44
OProfile

44.1 Introduction

OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.

OProfile is released under the GNU GPL. It consists of a kernel driver, a daemon for collecting sample data, and several post-profiling tools for turning data into information.

44.1.1 Overview

OProfile leverages the hardware performance counters of the CPU to enable profiling of a wide variety of interesting statistics, which can also be used for basic time-spent profiling.

All code is profiled: hardware and software interrupt handlers, kernel modules, the kernel, shared libraries, and applications.

44.1.2 Features

OProfile has the following features.

- Unobtrusive-No special recompilations or wrapper libraries are necessary. Even debug symbols (-g option to gcc) are not necessary unless users want to produce annotated source. No kernel patch is needed; just insert the module.
- System-wide profiling-All code running on the system is profiled, enabling analysis of system performance.
- Performance counter support-Enables collection of various low-level data and association for particular sections of code.
- Call-graph support-With an 2.6 kernel, OProfile can provide gprof-style call-graph profiling data.

- Low overhead-OProfile has a typical overhead of 1-8% depending on the sampling frequency and workload.
- Post-profile analysis-Profile data can be produced on the function-level or instruction-level detail. Source trees, annotated with profile information, can be created. A hit list of applications and functions that utilize the most CPU time across the whole system can be produced.
- System support-Works with almost any 2.2, 2.4 and 2.6 kernels, and works on based platforms.

44.1.3 Hardware Operation

OProfile is a statistical continuous profiler.

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

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

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

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

44.2 Software Operation

44.2.1 Architecture Specific Components

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

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

44.2.2 oprofilefs Pseudo Filesystem

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

44.2.3 Generic Kernel Driver

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

44.2.4 OProfile Daemon

The OProfile userspace daemon takes the raw data provided by the kernel and writes it to the disk. It takes the single data stream from the kernel and logs sample data against a number of sample files (available in /var/lib/oprofile/samples/current/). For the benefit of

the separate functionality, the names and paths of these sample files are changed to reflect where the samples were from. This can include thread IDs, the binary file path, the event type used, and more.

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

44.2.5 Post Profiling Tools

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

44.3 Requirements

OProfile has the following requirements.

- Add Oprofile support with Cortex-A8 Event Monitor

44.3.1 Source Code Structure

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

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

Table 44-1. OProfile Source Files

File	Description
op_arm_model.h	Header File with the register and bit definitions
common.c	Source file with the implementation required for all platforms

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

44.3.2 Menu Configuration Options

The following Linux kernel configurations are provided for this module.

To get to the Oprofile configuration, use the command ./ltib -c from the <ltib dir>. On the screen, first go to Package list and select oprofile. Then return to the first screen and, select **Configure Kernel**, then exit, and a new screen appears.

- CONFIG_OPROFILE-configuration option for the oprofile driver. In the menuconfig this option is available under
- General Setup > Profiling support (EXPERIMENTAL) > OProfile system profiling (EXPERIMENTAL)

44.3.3 Programming Interface

This driver implements all the methods required to configure and control PMU and L2 cache EVTMON counters.

More information, see the Linux document generated from build: make htmldocs.

44.3.4 Interrupt Requirements

The number of interrupts generated with respect to the OProfile driver are numerous. The latency requirements are not needed.

The rate at which interrupts are generated depends on the event.

44.3.5 Example Software Configuration

The following steps show and example of how to configure the OProfile:

1. Use the command ./ltib -c from the <ltib dir>. On the screen, first go to Package list and select oprofile. The current version in ltib is 0.9.5.
2. Then return to the first screen and select Configure Kernel, follow the instruction from Menu Configuration Options, to enable Oprofile in the kernel space.
3. Save the configuration and start to build.
4. Copy Oprofile binaries to target rootfs. Copy vmlinux to /boot directory and run Oprofile

```
root@ubuntu:/boot# opcontrol --separate=kernel --vmlinux=/boot/vmlinux
root@ubuntu:/boot# opcontrol --reset
Signalling daemon... done
root@ubuntu:/boot# opcontrol --setup --event=CPU_CYCLES:100000
```

```
root@ubuntu:/boot# opcontrol --start
Profiler running.
root@ubuntu:/boot# opcontrol --dump
root@ubuntu:/boot# opreport
Overflow stats not available
CPU: ARM V7 PMNC, speed 0 MHz (estimated)
Counted CPU_CYCLES events (Number of CPU cycles) with a unit mask of 0x00 (No un
it mask) count 100000
CPU_CYCLES:100000|
  samples|      %|
------------------
        4 22.2222 grep
        CPU_CYCLES:100000|
          samples|      %|
        ------------------
                4 100.000 libc-2.9.so
        2 11.1111 cat
        CPU_CYCLES:100000|
          samples|      %|
        ------------------
                1 50.0000 ld-2.9.so
                1 50.0000 libc-2.9.so
...
root@ubuntu:/boot# opcontrol --stop
Stopping profiling.
```

Chapter 45
CAAM (Cryptographic Acceleration and Assurance Module)

45.1 CAAM Device Driver Overview

This section discusses implementation specifics of the kernel driver components supporting CAAM (Cryptographic Acceleration and Assurance Module) within the Linux kernel.

CAAM's base driver packaging can be categorized on two distinct levels:

- Configuration and Job Execution Level
- API Interface Level

Configuration and Job Execution Level consists of:

- a control and configuration module which maps the main register page and writes global or system required configuration information.
- a module that feeds jobs through job rings, and reports status.

API Interface Level consists of:

- An interface to the Scatterlist Crypto API supporting asynchronous single-pass authentication-encryption operations, and common blockciphers - `caamalg`.
- An interface to the Scatterlist Crypto API supporting asynchronous hashes - `caamhash`.
- An interface to the hwrng API supporting use of the Random Number Generator - `caamrng`.

45.2 Configuration and Job Execution Level

This section has two parts:

- Control/Configuration Driver
- Job Ring Driver

45.3 Control/Configuration Driver

The control and configuration driver is responsible for initializing and setting up the master register page, initializing early-on feature initialization, providing limited debug and monitoring capability, and generally ensuring that all other dependent driver subsystems can connect to a correctly-configured device.

Step by step, it performs the following actions at startup:

- Allocates a private storage block for this level.
- Maps a virtual address to the full CAAM register page.
- Maps a virtual address for the SNVS register page.
- Maps a virtual (cache coherent) address for Secure Memory.
- Registers the security violation interrupt.
- Selects the correct DMA address size for the platform, and sets DMA address masks to match.
- Identifies other pertinent interrupt connections
- Initializes all job ring instances
- If the system configuration includes a DPAA Queue Interface, that interface has frame-pop enabled.

NOTE

i.MX 6 configurations do not contain this logic.

- If the instance contains a TRNG, it's oscillator/entropy configuration is set and then "kickstarted".
- Configuration information is sent to the system console to indicate that the driver is alive, and what configuration it has assumed.
- If CONFIG_DEBUG_FS is selected in the kernel configuration, then entries are added to enable debugfs views to useful registers in the performance monitor. Register views are accessible under the caam/ctl directory at the debugfs root entry.

45.4 Job Ring Driver

The Job Ring driver is responsible for providing job execution service to higher-level drivers. It takes care of overall management of both input and output rings and interrupt service driving the output ring.

One driver call is available for higher layers to use for queueing jobs to a ring for execution:

```
int caam_jr_enqueue(struct device *dev, u32 *desc, void (*cbk)(struct device
*dev, u32 *desc, u32 status, void *areq), void *areq);
```

Arguments:

`dev` Pointer to the struct device associated with the job ring for use. In the current configuration, one or more struct device entries exist in the controller's private data block, one for each ring.

`desc` Pointer to a CAAM job descriptor to be executed. The driver will map the descriptor prior to execution, and unmap it upon completion. However, since the driver can't reasonably know anything about the data referenced by the descriptor, it is the caller's responsibility to map/flush any of this data prior to submission, and to unmap/invalidate data after the request completes.

`cbk` Pointer to a callback function that will be called when the job has completed processing.

`areq` Pointer to metadata or context data associated with this request. Often, this can contain referenced data mapping information that request postprocessing (via the callback) can use to clean up or release resources once complete.

Callback Function Arguments:

`dev` Pointer to the struct device associated with the job ring for use.

`desc` Pointer to the original descriptor submitted for execution.

`status` Completion status received back from the CAAM DECO that executed the request. Nonzero only if an error occurred. Strings describing each error are enumerated in error.c.

`areq` Metadata/context pointer passed to the original request.

Returns:

- Zero on successful job submission
- -EBUSY if the input ring was full
- -EIO if driver could not map the job descriptor

45.5 API Interface Level

caamalg module provides a connection through the Scatterlist Crypto API both for common symmetric blockciphers, and for single-pass authentication-encryption services. This table lists all installed authentication-encryption algorithms by their common name, driver name, and purpose. Note that certain platforms, such as i.MX 6, contain a low-power MDHA accelerator, which cannot support SHA384 or SHA512.

Name	Driver Name	Purpose
authenc(hmac(md5),cbc(aes))	authenc-hmac-md5-cbc-aescaam	Single-pass authentication/encryption using MD5 and AES-CBC
authenc(hmac(sha1),cbc(aes))	authenc-hmac-sha1-cbc-aescaam	Single-pass authentication/encryption using SHA1 and AES-CBC
authenc(hmac(sha224),cbc(aes))	authenc-hmac-sha224-cbcaes-caam	Single-pass authentication/encryption using SHA224 and AES-CBC
authenc(hmac(sha256),cbc(aes))	authenc-hmac-sha256-cbcaes-caam	Single-pass authentication/ encryptionusing SHA256 and AES-CBC
authenc(hmac(sha384),cbc(aes))	authenc-hmac-sha384-cbcaes-caam	Single-pass authentication/encryption using SHA384 and AES-CBC
authenc(hmac(sha512),cbc(aes))	authenc-hmac-sha512-cbcaes-caam	Single-pass authentication/encryption using SHA512 and AES-CBC
authenc(hmac(md5),cbc(des3_ede))	authenc-hmac-md5-cbcdes3_ede-caam	Single-pass authentication/encryption using MD5 and Triple-DES-CBC
authenc(hmac(sha1),cbc(des3_ede))	authenc-hmac-sha1-cbcdes3_ede-caam	Single-pass authentication/encryption using SHA1 and Triple-DES-CBC
authenc(hmac(sha224),cbc(des3_ede))	authenc-hmac-sha224-cbcdes3_ede-caam	Single-pass authentication/encryption using SHA224 and Triple-DES-CBC
authenc(hmac(sha256),cbc(des3_ede))	authenc-hmac-sha256-cbcdes3_ede-caam	Single-pass authentication/encryption using SHA256 and Triple-DES-CBC
authenc(hmac(sha384),cbc(des3_ede))	authenc-hmac-sha384-cbcdes3_ede-caam	Single-pass authentication/encryption using SHA384 and Triple-DES-CBC
authenc(hmac(sha512),cbc(des3_ede))	authenc-hmac-sha512-cbc-des3_ede-caam	Single-pass authentication/encryption using SHA512 and Triple-DES-CBC
authenc(hmac(md5),cbc(des))	authenc-hmac-md5-cbc-descaam	Single-pass authentication/encryption using MD5 and Single-DES-CBC
authenc(hmac(sha1),cbc(des))	authenc-hmac-sha1-cbc-descaam	Single-pass authentication/encryption using SHA1 and Single-DES-CBC
authenc(hmac(sha224),cbc(des))	authenc-hmac-sha224-cbcdes-caam	Single-pass authentication/encryption using SHA224 and Single-DES-CBC
authenc(hmac(sha256),cbc(des))	authenc-hmac-sha256-cbcdes-caam	Single-pass authentication/encryption using SHA256 and Single-DES-CBC
authenc(hmac(sha384),cbc(des))	authenc-hmac-sha384-cbcdes-caam	Single-pass authentication/encryption using SHA384 and Single-DES-CBC
authenc(hmac(sha512),cbc(des))	authenc-hmac-sha512-cbcdes-caam	Single-pass authentication/encryption using SHA512 and Single-DES-CBC

This table lists all installed symmetric key blockcipher algorithms by their common name, driver name, and purpose.

Name	Driver Name	Purpose
cbc(aes)	cbc-aes-caam	AES with a CBC mode wrapper
cbc(des3_ede)	cbc-3des-caam	Triple DES with a CBC mode wrapper
cbc(des)	cbc-des-caam	Single DES with a CBC mode wrapper

Use of these services through the API is exemplified in the common conformance/ performance testing module in the kernel's crypto subsystem, known as tcrypt, visible in the kernel source tree at crypto/tcrypt.c.

The `caamhashmodule` provides a connection through the Scatterlist Crypto API both for common asynchronous hashes.

This table lists all installed asynchronous hashes by their common name, driver name, and purpose. Note that certain platforms, such as i.MX 6, contain a low-power MDHA accelerator, which cannot support SHA384 or SHA512.

Name	Driver Name	Purpose
sha1	sha1-caam	SHA1-160 Hash Computation
sha224	sha224-caam	SHA224 Hash Computation
sha256	sha256-caam	SHA256 Hash Computation
sha384	sha384-caam	SHA384 Hash Computation
sha512	sha512-caam	SHA512 Hash Computation
md5	md5-caam	MD5 Hash Computation
hmac(sha1)	hmac-sha1-caam	SHA1-160 Hash-based Message Authentication Code
hmac(sha224)	hmac-sha224-caam	SHA224 Hash-based Message Authentication Code
hmac(sha256)	hmac-sha256-caam	SHA256 Hash-based Message Authentication Code
hmac(sha384)	hmac-sha384-caam	SHA384 Hash-based Message Authentication Code
hmac(sha512)	hmac-sha512-caam	SHA512 Hash-based Message Authentication Code
hmac(md5)	hmac-md5-caam	MD5 Hash-based Message Authentication Code

Use of these services through the API is exemplified in the common conformance/ performance testing module in the kernel's crypto subsystem, known as tcrypt, visible in the kernel source tree at crypto/tcrypt.c.

The `caamrng` module installs a mechanism to use CAAM's random number generator to feed random data into a pair of buffers that can be accessed through `/dev/hw_random`.

`/dev/hw_random` is commonly used to feed the kernel's own entropy pool, which can be used internally, as an entropy source for other random data "devices".

For more information regarding support for this service, see `rng-tools` available in http://sourceforge.net/projects/gkernel/files/rng-tools.

45.6 Driver Configuration

Configuration of the driver is controlled by the following kernel confguration parameters (found under Cryptographic API -> Hardware Crypto Devices):

`CRYPTO_DEV_FSL_CAAM`

Enables building the base controller driver and the job ring backend.

`CRYPTO_DEV_FSL_CAAM_RINGSIZE`

Selects the size (e.g. the maximum number of entries) of job rings. This is selectable as a power of 2 in the range of 2-9, allowing selection of a ring depth ranging from 4 to 512 entries.

The default selection is 9, resulting in a ring depth of 512 job entries.

`CRYPTO_DEV_FSL_CAAM_INTC`

Enables the use of the hardware's interrupt coalescing feature, which can reduce the amount of interrupt overhead the system incurs during periods of high utilization. Leaving this disabled forces a single interrupt for each job completion, simplifying operation, but increasing overhead.

`CRYPTO_DEV_FSL_CAAM_INTC_COUNT_THLD`

If coalescing is enabled, selects the number of job completions allowed to queue before an interrupt is raised. This is selectable within the range of 1 to 255. Selecting 1 effectively defeats the coalescing feature. Any selection of a size greater than the job ring size will force a situation where the interrupt times out before ever raising an interrupt.

The default selection is 255.

`CRYPTO_DEV_FSL_CAAM_INTC_TIME_THLD`

If coalescing is enables, selects the count of bus clocks (divided by 64) before a coalescing timeout where, if the count threshold has not been met, an interrupt is raised at the end of the time period. The selection range is an integer from 1 to 65535.

The default selection is 2048.

`CRYPTO_DEV_FSL_CAAM_CRYPTO_API`

Enables Scatterlist Crypto API support for asynchronous blockciphers and for single-pass autentication-encryption operations through the API using CAAM hardware for acceleration.

`CRYPTO_DEV_FSL_CAAM_AHASH_API`

Enables Scatterlist Crypto API support for asynchronous hashing through the API using CAAM hardware for acceleration.

`CRYPTO_DEV_FSL_CAAM_RNG_API`

Enables use of the CAAM Random Number generator through the hwrng API. This can be used to generate random data to feed an entropy pool for the kernels pseudo-random number generator.

`CRYPTO_DEV_FSL_CAAM_RNG_TEST`

Enables a captive test to ensure that the CAAM RNG driver is operating and buffering random data.

45.7 Limitations

- Components of the driver do not currently build and run as modules. This may be rectified in a future version.
- Interdependencies exist between the controller and job ring backends, therefore they all must run in the same system partition. Future versions of the driver may separate out the job ring back-end as a standalone module that can run independently (and support independent API and SM instances) in it's own system partition.
- The full CAAM register page is mapped by the controller driver, and derived pointers to selected subsystems are calculated and passed to higher-layer driver components. Partition-independent configurations will have to map their own subsystem pointers instead.
- Upstream variants of this driver support only Power architecture. This ARM-specific port is not upstreamed at this time, although portions may be upstreamed at some point.
- TRNG kickstart may need to be moved to the bootloader in a future release, so that the RNG can be used earlier.
- The Job Ring driver has a registration and de-registration functions that are not currently necessary (and may be rewritten in future editions to provide for shutdown notifications to higher layers.

45.8 Limitations in the Existing Implementation Overview

This chapter describes a prototype of a Keystore Management Interface intended to provide access to CAAM Secure Memory.

Secure memory provides a controlled and access-protected area where critical system security parameters can be stored and processed in a running system without bus-level exposure of clear secrets. Secrets can be imported into and exported from secure memory, but never exported from secure memory in their cleartext form. Instead, secrets may be exported from secure memory in a covered form, using keys never visible to the outside.

This driver, with it's kernel-level API, exposes a basic interface to allow kernel-level services access to secure memory functionality. It is split into two pieces:

- Keystore Initialization and Maintenance Interfaces
- Keystore Access Interface

The initialization and maintenance services exist to initialize and define the instance of a keystore interface. Likewise, the access interface allows kernel-level services to use the API for management of security parameters.

45.9 Initialize Keystore Management Interface

Installs a set of pointers to functions that implement an underlying physical interface to the keystore subsystem.

In the present release, a default (and hidden) suite of functions implement this interface. Future implementations of this API may provide for the installation of an alternate interface. If this occurs, an alternate to this call can be provided.

```
void sm_init_keystore(struct device *dev);
```

Arguments:

`dev` points to a `struct device` established to manage resources for the secure memory subsystem.

45.10 Detect Available Secure Memory Storage Units

Returns the number of available units ("pages") that can be accessed by the local instance of this driver. Intended for use as a resource probe.

```
u32 sm_detect_keystore_units(struct device *dev);
```

Arguments:

`dev` Points to a `struct device` established to manage resources for the secure memory subsystem.

Returns: Number of detected units available for use, 0 through n - 1 may be used with subsequent calls to all other API functions.

45.11 Establish Keystore in Detected Unit

Sets up an allocation table in a detected unit that can be used for the storage of keys (or other secrets). The unit will be divided into a series of fixed-size slots, each one of which is marked available in the allocation table. The size of each slot is a build-time selectable parameter.

No calls to the keystore access interface can occur until `sm_establish_keystore()` has been called.

`sm_establish_keystore()` should follow a call to `sm_detect_keystore_units()`.

```
int sm_establish_keystore(struct device *dev, u32 unit);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

Returns:

- Zero on successful return
- -EINVAL if the keystore subsystem was not initialized
- -ENOSPC if no memory was available for the allocation table and associated context data.

45.12 Release Keystore

Releases all resources used by this keystore unit. No further calls to the keystore access interface can be made.

```
void sm_release_keystore(struct device *dev, u32 unit);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

45.13 Allocate a Slot from the Keystore

Allocate a slot from the keystore for use in all other subsequent operations by the keystore access interface.

```
int sm_keystore_slot_alloc(struct device *dev, u32 unit, u32 size, u32*slot);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

`size` Desired size of data for storage in the allocated slot.

`slot` Pointer to the variable to receive the allocated slot number, once known.

Returns:

- Zero for successful completion.
- -EKEYREJECTED if the requested size exceeds the selected slot size.

45.14 Load Data into a Keystore Slot

Load data into an allocated keystore slot so that other operations (such as encapsulation) can be carried out upon it.

```
int sm_keystore_slot_load(struct device *dev, u32 unit, u32 slot, constu8 *key_data, u32
key_length);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

`key_length` Length (in bytes) of information to write to the slot.

`key_data` Pointer to buffer with the data to be loaded. Must be a contiguous buffer.

Returns:

- Zero for successful completion.
- -EFBIG if the requested size exceeds that which the slot can hold.

45.15 Demo Image Update

Encapsulate data written into a keystore slot as a Secure Memory Blob.

```
int sm_keystore_slot_encapsulate(struct device *dev, u32 unit, u32
inslot, u32 outslot, u16 secretlen, u8 *keymod, u16 keymodlen);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

`inslot` Slot holding the input secret, loaded into that slot by sm_keystore_slot_load(). Note that the slot containing this secret should be overwritten or deallocated as soon as practical, since it contains cleartext at this point.

`outslot` Allocated slot to hold the encapsulated output as a Secure Memory Blob.

`secretlen` Length of the secret to be encapsulated, not including any blob storage overhead (blob key, MAC, etc.).

`keymod` Key modifier component to be used for encapsulation. The key modifier allows an extra secret to be used in the encapsulation process. The same modifier will also be required for decapsulation.

`keymodlen` Lenth of key modifier in bytes.

Returns:

- Zero on success
- CAAM job status if a failure occurs

45.16 Decapsulate Data in the Keystore

Decapsulate data in the keystore into a Black Key Blob for use in other cryptographic operations. A Black Key Blob allows a key to be used "covered" in main memory without exposing it as cleartext.

```
int sm_keystore_slot_decapsulate(struct device *dev, u32 unit, u32
inslot, u32 outslot, u16 secretlen, u8 *keymod, u16 keymodlen);
```

Arguments:

`dev` Points to a struct device established to manage resourcesfor the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

`inslot` Slot holding the input data, processed by a prior call to `sm_keystore_slot_encapsulate()`, and containing a Secure Memory Blob.

`outslot` Allocated slot to hold the decapsulated output data in the form of a Black Key Blob.

`secretlen` Length of the secret to be decapsulated, without any blob storage overhead.

`keymod` Key modified component specified at the time of encapsulation.

`keymodlen` Lenth of key modifier in bytes.

Returns:

- Zero on success
- CAAM job status if a failure occurs

45.17 Read Data From a Keystore Slot

Extract data from a keystore slot back to a user buffer. Normally to be used after some other operation (e.g. decapsulation) occurs.

```
int sm_keystore_slot_read(struct device *dev, u32 unit, u32 slot, u32
key_length, u8 *key_data);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to sm_detect_keystore_units().

`slot` Allocated slot to read from.

`key_length` Length (in bytes) of information to read from the slot.

`key_data` Pointer to buffer to hold the extracted data. Must be a contiguous buffer.

Returns:

- Zero for successful completion.
- -EFBIG if the requested size exceeds that which the slot can hold.

45.18 Release a Slot back to the Keystore

Release a keystore slot back to the available pool. Information in the store is wiped clean before the deallocation occurs.

```
int sm_keystore_slot_dealloc(struct device *dev, u32 unit, u32 slot);
```

Arguments:

`dev` Points to a struct device established to manage resources for the secure memory subsystem.

`unit` One of the units detected with a call to `sm_detect_keystore_units()`.

`slot` Number of the allocated slot to be released back to the store.

Returns:

- Zero for successful completion.
- -EINVAL if an unallocated slot is specified.

Configuration of the Secure Memory Driver / Keystore API is dependent on the following kernel configuration parameters:

```
CRYPTO_DEV_FSL_CAAM_SM
```

Turns on the secure memory driver in the kernel build.

```
CRYPTO_DEV_FSL_CAAM_SM_SLOTSIZE
```

Configures the size of a secure memory "slot".

Each secure memory unit is block of internal memory, the size of which is implementation dependent. This block can be subdivided into a number of logical "slots" of a size which can be selected by this value. The size of these slots needs to be set to a value that can hold the largest secret size intended, plus the overhead of blob parameters (blob key and MAC, typically no more than 48 bytes).

The values are selectable as powers of 2, limited to a range of 32 to 512 bytes. The default value is 7, for a size of 128 bytes.

`CRYPTO_DEV_FSL_CAAM_SM_TEST`

Enables operation of a captive test / example module that shows how one might use the API, while verifying it's functionality. The test module works along this flow:

- Creates a number of known clear keys (3 sizes).
- Allocated secure memory slots.
- Inserts those keys into secure memory slots and encapsulates.
- Decapsulates those keys into black keys.
- Enrcrypts DES, AES128, and AES256 plaintext with black keys. Since this uses symmetric ciphers, same-key encryption/decryption results will be equivalent.
- Decrypts enciphered buffers with equivalent clear keys.
- Compares decrypted results with original ciphertext and compares. If they match, the test reports OK for each key case tested.

Normal output is reported at the console as follows:

```
platform caam_sm.0: caam_sm_test: 8-byte key test match OK platform
caam_sm.0: caam_sm_test: 16-byte key test match OK platform caam_sm.0:
caam_sm_test: 32-byte key test match OK
```

- The secure memory driver is not implemented as a kernel module at this point in time.
- Implementation is presently limited to kernel-mode operations.
- One instance is possible at the present time. In the future, when job rings can run independently in different system partitions, a multiple instance secure memory driver should be considered.
- All storage requests are limited to the storage size of a single slot (which is of a build-time configurable length). It may be possible to allow a secret to span multiple slots so long as those slots can be allocated contiguously.
- Slot size is fixed across all pages/partitions.
- Encapsulation/Decapsulation interfaces could allow for authentication to be specified; the underlying interface does not request it.
- Encapsulation/Decapsulation interfaces return a job status; this status should be translated into a meaningful error from `errno.h`

45.19 CAAM/SNVS - Security Violation Handling Interface Overview

This chapter describes a prototype of a driver component and control interface for SNVS Security Violations. It provides a means of installing, managing, and executing application defined handlers meant to process security violation events as a response to their occurrence in a system.

SNVS allows for the continuous monitoring of a number of possible attack vectors in a running system. If the occurrence of one of these attach vectors is sensed, (e.g. a Security Violation has been detected), SNVS can, along with erasing critical security parameters and transitioning to a failure state. generate an interrupt indicating that the violation has occurred. This interrupt can dispatch an application-defined routine to take cleanup action as a consequence of the violation, such that an orderly shutdown of security services might occur.

Therefore, the purpose of this interface is to allow system-level services to install handlers for these types of events. This will allow the system designer to select how he wants to respond to specific security violation causes using a simple function call written to his system-specific requirements.

45.20 Operation

For existing platforms, 6 security violation interrupt causes are possible within SNVS. 5 of these violation causes are normally wired for use, and these causes are defined as:

- SECVIO_CAUSE_CAAM_VIOLATION - Violation detected inside CAAM/SNVS
- SECVIO_CAUSE JTAG_ALARM - JTAG activity detected
- SECVIO_CAUSE_WATCHDOG - Watchdog expiration
- SECVIO_CAUSE_EXTERNAL_BOOT - External bootload activity
- SECVIO_CAUSE_TAMPER_DETECT - Tamper detection logic triggered

Each of these causes can be associated with an application-defined handler through the API provided with this driver. If no handler is specified, then a default handler will be called. This handler does no more than to identify the interrupt cause to the system console.

45.21 Configuration Interface

The following interface can be used to define or remove application-defined violation handlers from the driver's dispatch table.

45.22 Install a Handler

```
int caam_secvio_install_handler(struct device *dev, enum secvio_cause
cause, void (*handler)(struct device *dev, u32 cause, void *ext), u8
*cause_description, void *ext);
```

Arguments:

`dev` Points to SNVS-owning device.

`cause` Interrupt source cause from the above list of enumerated causes.

`handler` Application-defined handler, gets called with dev, source cause, and locally-defined handler argument

`cause_description` Points to a string to override the default cause name, this can be used as an alternate for error messages and such. If left NULL, the default description string is used. ext pointer to any extra data needed by the handler.

Returns:

- Zero on success.
- -EINVAL if an argument was invalid or unusable.

45.23 Remove an Installed Driver

```
int caam_secvio_remove_handler(struct device *dev, enum secvio_cause
cause);
```

Arguments:

`dev` Points to SNVS-owning device.

`cause` Interrupt source cause.

Returns:

- Zero on success.
- -EINVAL if an argument was invalid or unusable.

45.24 Driver Configuration CAAM/SNVS

`CRYPTO_DEV_FSL_CAAM_SECVIO`

Enables inclusion of Security Violation driver and configuration interface as part of the build configuration. Note that the driver is not buildable as a module in its present form.

How to Reach Us:

Home Page:
freescale.com

Web Support:
freescale.com/support

Document Number: IMX6QLXRM
Rev. L3.0.35_4.1.0
09/2013