RM00293

i.MX Linux Reference Manual

Rev. LF6.12.34_2.1.0 — 25 September 2025 Reference manual

Document information

Information	Content
Keywords	i.MX, Linux, RM00293, LF6.12.34_2.1.0
Abstract	The i.MX family Linux Board Support Package (BSP) supports the Linux Operating System (OS) on the i.MX application processors.

1 Introduction

1.1 Overview

The Linux Board Support Package (BSP) for the i.MX family supports the Linux Operating System (OS) on the i.MX application processors.

This software package is designed to support Linux OS on the i.MX family of Integrated Circuits (ICs) and their corresponding platforms. It provides the necessary software to interface the standard open-source Linux kernel to the i.MX hardware. The goal is to help i.MX customers accelerate product build based on i.MX devices that use the Linux OS.

The BSP is not a platform or product reference implementation. It does not include all of the product-specific drivers, hardware-independent software stacks, Graphical User Interface (GUI) components, Java Virtual Machine (JVM), and applications required for a product. Some of these are made available in their original open-source form as part of the base kernel.

The BSP is not intended for silicon verification. While it may assist in this process, its functionality and test coverage are not sufficient to replace traditional silicon verification test suites.

1.1.1 Software base

The i.MX BSP is based on the version 6.12.34 of the Linux kernel from the official Linux kernel website (www.kernel.org). It is enhanced with the features provided by NXP.

On Linux OS, to change the configuration using the menu configuration with a Yocto Project environment, use Bitbake as follows:

```
bitbake linux-imx -c menuconfig
```

1.1.2 Features

The table below describes the features supported by the BSP for specific platforms.

Table 1. BSP supported features

Feature	Description	Chapter Source	Applicable Platform
Machine-Specific Layer			
MSL	Machine-Specific Layer (MSL) supports interrupts, Timer, Memory Map, GPIO/IOMUX, SPBA, and SDMA. • Interrupts GIC: The Linux kernel contains common Arm GIC interrupts handling code. • Timer (GPT): The General Purpose Timer (GPT) is set up to generate an interrupt as programmed to provide OS ticks. Linux OS facilitates timer use through various functions for timing delays, measurement, events, alarms, high-resolution timer features, and so on. Linux OS defines the MSL timer API required for the OS-tick timer and does not expose it beyond the kernel tick implementation. • GPIO/EDIO/IOMUX: The GPIO and EDIO components in the MSL provide an abstraction layer between the various drivers and the	Machine-Specific Layer (MSL)	All

Table 1. BSP supported features*...continued*

Feature	Description	Chapter Source	Applicable Platform
	configuration and utilization of the system, including GPIO, IOMUX, and external board I/O. The IO software module is board-specific, and resides in the MSL layer as a self-contained set of files. I/O configuration changes are centralized in the GPIO module so that changes are not required in the various drivers. • SPBA: The Shared Peripheral Bus Arbiter (SPBA) provides an arbitration mechanism among multiple masters to allow access to the shared peripherals. The SPBA implementation under MSL defines the API to allow different masters to take or release ownership of a shared peripheral.		
General Drivers			
Thermal Driver	The thermal driver monitors the SoC temperature in a certain frequency to protect the SoC. It defines three trip points: critical, hot, and active.	Thermal Driver	All
OProfile	OProfile is a system-wide profiler for Linux systems, capable of profiling all running code at low overhead.	OProfile	All
Pulse Width Modulator	The pulse-width modulator (PWM) has a 16-bit counter and is optimized to generate sound from stored sample audio images and generate tones.	Pulse-Width Modulator (PWM)	All
Sensors	Sensors cover accelerometer, ambient light, and magnetometer sensors.	Sensors	All
Watchdog	The Watchdog Timer module protects against system failures by providing an escape from unexpected hang or infinite loop situations or programming errors.	Watchdog	All
DMA Engine			
SDMA API	The Smart Direct Memory Access (SDMA) API driver controls the SDMA hardware and provides an API to other drivers for transferring data between MCU, DSP, and peripherals.	Smart Direct Memory Access (SDMA) API	All
APBH-Bridge-DMA	Both AHB-to-APBH and AHB-to-APBX DMA support configurable DMA descript chain.	AHB-to-APBH Bridge with DMA (APBH-Bridge-DMA)	All
Power Management Drivers			
Low-level Power Management	The low-level power management driver implements hardware-specific operations to meet power requirements and conserves power. Driver implementations are often different for different platforms. It is used by the DPM layer.	Low-level Power Management (PM) Driver	All
Dynamic Bus Frequency	The bus frequency driver dynamically manages the various system frequencies to improve power consumption.	Dynamic Bus Frequency Driver	i.MX 6 and i.MX 7
CPU Freq	The CPU frequency scaling allows the clock speed of the CPU to be changed.	CPUFreq	All

Table 1. BSP supported features*...continued*

Feature	Description	Chapter Source	Applicable Platform
PMIC PF Regulator	PF regulator driver provides the low-level control of the power supply regulators, selection of voltage levels, and enabling/disabling of regulators.	PF_Regulator	All
Anatop Regulator	The Anatop regulator drive provides low-level control of power supply regulators.	Anatop Regulator	i.MX 6 and i.MX 7
Connectivity Drivers			
ENET 1588 Stack	Implementation of the Precision Time Protocol (PTP) according to IEEE standard 1588.	Fast Ethernet Controller (FEC) Driver	All except for i.MX 95 and i.MX 943
Fast Ethernet Controller	The ENET Driver performs the full set of IEEE 802.3/Ethernet CSMA/CD media access control and channel interface functions.	Fast Ethernet Controller (FEC) Driver	All except for i.MX 95 and i.MX 943
FlexCAN	The FlexCAN driver provides the interfaces to send and receive CAN messages.	FlexCAN Driver	i.MX 6Quad, i.MX 6Dual, i.MX 6DualLite, i.MX 6Solo, i.MX 6UltraLite, i.MX 6SoloX, i.MX 7Dual, i.MX 8M Plus, i.MX 8QuadMax, i.MX 8DXL, i.MX 8Quad XPlus, i.MX 9
MediaLB	MediaLB is an on-PCB or inter-chip communication bus allowing applications to access the MOST Network data or communicate with other applications.	MediaLB	i.MX 6SoloX i.MX 6Quad i.MX 6Dual
PCIe	PCI Express hardware module can be either configured to act as a Root Complex or a PCIe Endpoint.	PCIe	All
Ethernet Controller	The NIC functionality in NETC is known as Ethernet Controller (ENETC). ENETC supports virtualization/isolation based on PCIe Single Root IO Virtualization (SR-IOV), advanced QoS with 8 traffic classes and 4 drop resilience levels, and a full range of TSN standards and NIC offload capabilities.	Section 4.13	i.MX 95, i.MX 943
NETC 1588 Timer	NETC 1588 Timer provides current time with nanosecond resolution, precise periodic pulse, pulse on timeout (alarm), and time capture on external pulse support. This blocks capabilities support implementing time synchronization as required for IEEE 1588 and IEEE 802.1AS-2020.	Section 4.14	i.MX 95, i.MX 943
NETC Switch	NETC provides full 802.1Q Ethernet switch functionality, advanced QoS with 8 traffic classes and 4 drop resilience levels, and a full range of TSN standards capabilities.	Section 4.15	i.MX 943
Ethernet Controller with TSN (ENET_QoS)	The EQoS driver supports RMII/RGMII in compliance with the IEEE 802.3-2015. Also supports TSN/AVB.	Section 4.16	i.MX 8DXL, i.MX 8M Plus, i.MX 91/93
Video			

Table 1. BSP supported features*...continued*

Feature	Description	Chapter Source	Applicable Platform
Capture	Camera Overview for Camera and capture interfaces.	Capture Overview	All
Display	Display Overview.	Display Overview	All
VPU	The Video Processing Unit (VPU) is a multistandard video decoder and encoder that can perform decoding and encoding of various video formats.	Video Processing Unit (VPU) Driver	i.MX 6QuadPlus/Quad/Dual/Solo, i.MX 8, and i.MX 95
JPEGENC/ JPEGDEC	The JPEG-E-X and JPEG-D-X cores are standalone and high-performance 8-bit and 12-bit JPEG encoder and respectively decoder for still image and video compression/decompression applications.	JPEG Encoder and Decoder	i.MX 8QuadXPlus, 8Quad Max, and i.MX 95
Audio Drivers			
ALSA Sound	The Advanced Linux Sound Architecture (ALSA) is a sound driver that provides ALSA and OSS compatible applications with the means to perform audio playback and recording functions.	ALSA Sound Driver	All
ASRC	The Asynchronous Sample Rate Converter (ASRC) driver provides the interfaces to access the asynchronous sample rate converter module.	Asynchronous Sample Rate Converter (ASRC)	All
S/PDIF	The S/PDIF driver is designed under the Linux ALSA subsystem. It implements one playback device for TX and one capture device for RX.	The Sony/Philips Digital Interface (S/PDIF) Driver	All
Storage MTD Drivers			
SPI NOR MTD	The SPI NOR MTD driver provides the support to the Atmel data Flash using the SPI interface.	SPI NOR Flash Memory Technology Device (MTD) Driver	All
NAND MTD	The NAND MTD driver interfaces with the integrated NAND controller supporting UBIFS, CRAMFS, JFFS2UBI, UBIFSCRAMFS, and JFFS2 file systems.	NAND GPMI Flash Driver	i.MX 6Quad, i.MX 6Dual, i.MX 6DualLite, i.MX 6Solo, i.MX 6UltraLite, i.MX 7Dual
SATA	The SATA AHCI driver is based on the LIBATA layer of the block device infrastructure of the Linux kernel.	SATA Driver	i.MX 6QuadPlus, i.MX 6Quad, i.MX 6Dual, i.MX 8QuadMax, i.MX 8Quad XPlus
Bus Drivers			
I2C	The Lower Power I2C bus driver interfaces with the I2C bus to transfer data over the I2C bus.	Inter-IC (I2C) Driver	All
eCSPI	The low-level Enhanced Configurable Serial Peripheral Interface (ECSPI) driver interfaces a custom, kernel-space API to both ECSPI modules.	Enhanced Configurable Serial Peripheral Interface (ECSPI) Driver	i.MX 6, i.MX 7, i.MX 8
LPSPI	LPSPI provides an efficient interface (either as a controller or peripheral) to an SPI bus, which is a synchronous serial communication interface used in embedded systems.	Section 4.18	i.MX 7ULP, i.MX 8ULP, i.MX 8X, i.MX 93, i.MX 95, i.MX 943

Table 1. BSP supported features*...continued*

Feature	Description	Chapter Source	Applicable Platform
MMC/SD/SDIO -uSDHC	The MMC/SD/SDIO Host driver implements the standard Linux driver interface to eSDHC.	MMC/SD/SDIO Host Driver	All
Connectivity Drivers			
UART	The Universal Asynchronous Receiver/ Transmitter (UART) driver interfaces the serial driver API to all UART ports.	Universal Asynchronous Receiver/Transmitter (UART) Driver	All
USB	The USB driver interfaces to the ARC USB-OTG controller.	CHIPIDEA USB	All
USB3	The USB driver interfaces to the ARC/Cadence/ DWC3 USB controller.	Section 4.10	All

1.2 Audience

This document is targeted to individuals who port the i.MX Linux OS Board Support Package (BSP) to customer-specific products.

The audience is expected to have a working knowledge of the Linux kernel internals, driver models, and i.MX processors.

1.2.1 Conventions

This document uses the following notational conventions:

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

1.2.2 Acronyms, abbreviations, and definitions

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

Table 2. Acronyms, abbreviations, and definitions

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

Table 2. Acronyms, abbreviations, and definitions...*continued*

Term	Definition
CPU	Central Processing Unit: generic term used to describe a processing core.
CRC	Cyclic Redundancy Check: Bit error protection method for data communication.
CSI	Camera Sensor Interface.
DCNANO	Display Controller Nano: a high-performance graphics core that can be used for reading rendered images from the frame buffer.
DFS	Dynamic Frequency Scaling.
DMA	Direct Memory Access: an independent block that can initiate memory-to-memory data transfers.
DPM	Dynamic Power Management.
DCSS	Display controller sub-system.
DP	Display Port: similar IP as HDMI.
DPU	Display Processor Unit.
DSI	Display Serial Interface.
DRM	Display Rendering Manager or Digital Rights Manager.
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.
EPDC	Electrophoretic Display Controller.
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 the 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.
GPU	Graphics Processor Unit.
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,

Table 2. Acronyms, abbreviations, and definitions*...continued*

Term	Definition
	and is generated by a formula in such a way that it is unlikely that some other text produces the same hash value.
HDMI	High-Definition Multimedia Interface.
I/O	Input/Output.
ICE	In-Circuit Emulation.
IP	Intellectual Property.
IPU	Image Processing Unit: supports video and graphics processing functions and provides an interface to video/still image sensors and displays.
IrDA	Infrared Data Association: a nonprofit organization whose goal is to develop globally adopted specifications for infrared wireless communication.
ISR	Interrupt Service Routine.
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).
LDB	LVDS Display Bridge.
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.
LVDS	Low Voltage Differential Signaling.
MIPI	Mobile Industry Process Interface.
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 multicompany 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.

Table 2. Acronyms, abbreviations, and definitions*...continued*

Term	Definition
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.
PxP	Pixel Pipeline.
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 low and exception vectors.
RPMSG	Remote Processor Messaging.
RTIC	Real-Time Integrity Checker: a security hardware module.
SC	System Controller.
SCC	SeCurity Controller: a security hardware module.
SCFW	System Controller Firmware.
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 primary/secondary 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 Mbps 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 can also connect to a host PC.
VADC	Video analog to Digital Converter.
VPU	Video Processing Unit.

Table 2. Acronyms, abbreviations, and definitions*...continued*

Term	Definition
Word	A group of bits comprising 32-bits.

1.3 References

i.MX has multiple families supported in software. The following lists the available families and their respective System-on-Chips (SoCs). Details on which SoCs are supported in the current release can be found in the *i.MX Linux Release Notes* (RN00210). Some SoCs from previous releases may still be buildable in the current release but not validated if they are at the previous validated level.

- i.MX 6 Family: 6QuadPlus, 6Quad, 6DualLite, 6SoloX, 6SLL, 6UltraLite, 6ULL, 6ULZ
- i.MX 7 Family: 7Dual, 7ULP
- i.MX 8 Family: 8QuadMax, 8QuadPlus, 8ULP
- i.MX 8M Family: 8M Plus, 8M Quad, 8M Mini, 8M Nano
- i.MX 8X Family: 8QuadXPlus, 8DXL, 8DXL OrangeBox, 8DualX
- i.MX 9 Family: i.MX 91, i.MX 93, i.MX 95, i.MX 943

This release includes the following references and additional information:

- *i.MX Linux Release Notes* (RN00210): Provides the release information.
- *i.MX Linux User's Guide* (UG10163): Provides the information on installing U-Boot and Linux OS and using i.MX-specific features.
- *i.MX Yocto Project User's Guide* (UG10164): Describes the board support package for NXP development systems using Yocto Project to set up host, install tool chain, and build source code to create images.
- *i.MX Porting Guide* (UG10165): Provides the instructions on porting the BSP to a new board.
- *i.MX Machine Learning User's Guide* (UG10166): Provides the machine learning information.
- *i.MX DSP User's Guide* (UG10167): Provides the information on the DSP for i.MX 8.
- *i.MX 8M Plus Camera and Display Guide* (UG10168): Provides the information on the ISP Independent Sensor Interface API for the i.MX 8M Plus.
- *i.MX Digital Cockpit Hardware Partitioning Enablement for i.MX 8QuadMax* (UG10169): Provides the i.MX Digital Cockpit hardware solution for i.MX 8QuadMax.
- *i.MX Graphics User's Guide* (UG10159): Describes the graphics features.
- *Harpoon User's Guide* (UG10170): Presents the Harpoon release for i.MX 8M device family.
- *i.MX Linux Reference Manual* (RM00293): Provides the information on Linux drivers for i.MX.
- *i.MX VPU Application Programming Interface Linux Reference Manual* (RM00294): Provides the reference information on the VPU API on i.MX 6 VPU.
- *EdgeLock Enclave Hardware Security Module API* (RM00284): This document is a software reference description of the API provided by the i.MX 8ULP, i.MX 93, and i.MX 95 Hardware Security Module (HSM) solutions for the EdgeLock Enclave (ELE) Platform.

The quick start guides provide basic information about the board and instructions for setting it up. They are available on the NXP website:

- SABRE Platform Quick Start Guide (IMX6QSDPQSG)
- i.MX 6UltraLite EVK Quick Start Guide (IMX6ULTRALITEQSG)
- i.MX 6ULL EVK Quick Start Guide (IMX6ULLQSG)
- i.MX 7Dual SABRE-SD Quick Start Guide (SABRESDBIMX7DUALQSG)
- i.MX 8M Quad Evaluation Kit Quick Start Guide (IMX8MQUADEVKQSG)
- i.MX 8M Mini Evaluation Kit Quick Start Guide (8MMINIEVKQSG)
- i.MX 8M Nano Evaluation Kit Quick Start Guide (8MNANOEVKQSG)
- i.MX 8QuadXPlus Multisensory Enablement Kit Quick Start Guide (IMX8QUADXPLUSQSG)

- i.MX 8QuadMax Multisensory Enablement Kit Quick Start Guide (IMX8QUADMAXQSG)
- i.MX 8M Plus Evaluation Kit Quick Start Guide (IMX8MPLUSQSG)
- i.MX 8ULP EVK Quick Start Guide (IMX8ULPQSG)
- i.MX 8ULP EVK9 Quick Start Guide (IMX8ULPEVK9QSG)
- i.MX 93 EVK Quick Start Guide (IMX93EVKQSG)
- i.MX 93 9x9 QSB Quick Start Guide (93QSBQSG)

Documentation is available at nxp.com.

- i.MX 6 information is at nxp.com/iMX6series.
- i.MX SABRE information is at nxp.com/imxSABRE.
- i.MX 6UltraLite information is at nxp.com/iMX6UL.
- i.MX 6ULL information is at nxp.com/iMX6ULL.
- i.MX 7Dual information is at nxp.com/iMX7D.
- i.MX 7ULP information is at nxp.com/imx7ulp.
- i.MX 8 information is at nxp.com/imx8.
- i.MX 6ULZ information is at nxp.com/imx6ulz.
- i.MX 91 information is at nxp.com/imx91.
- i.MX 93 information is at nxp.com/imx93.
- i.MX 95 information is at nxp.com/imx95.
- i.MX 943 information is at nxp.com/imx94.

2 System

2.1 Machine-Specific Layer (MSL)

2.1.1 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
- Clock
- Shared Peripheral Bus Arbiter (SPBA)
- Smart Direct Memory Access (SDMA)

2.1.2 Interrupts (Operation)

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

2.1.2.1 Interrupt hardware operation

The Interrupt Controller controls and prioritizes all internal and external interrupt sources. By default, all interrupts have the same priority.

Each interrupt source can be enabled or disabled by configuring the interrupt controller’s registers.

There are three types of interrupts in GIC:

- PPI is a private peripheral interrupt of each CPU. It can only be handled by each CPU.
- SGI is a software generated interrupt. It can be triggered by software operation, and can only be handled by each CPU.
- SPI is a shared peripheral interrupt, which is an external interrupt source from the SoC platform. It can be handled by all CPUs.

2.1.2.2 Interrupt software operation

For the Arm architecture-based processors with GIC-400 of i.MX 6 and i.MX 7 SoCs, normal interrupt and fast interrupt are two different exception types. The exception vector addresses can be configured to start at a low address (0x0) or high address (0xFFFF0000) for i.MX 6 and i.MX 7 platforms. The Linux OS implementation running on the Arm architecture chooses the high-vector address model.

For Arm architecture-based processors with GIC-500 of i.MX 8 SoCs, the exception vector addresses are defined as VBAR_ELn + offset. The offset depends on which exception level the interrupt exception is taken. The file `Documentation/arm/Interrupts` has a description of the Arm interrupt architecture.

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 the system startup.

Table 3. Interrupt files

File	Description
`drivers/irqchip/irq-gic.c`	i.MX 6/7 SoCs with GIC-400
`drivers/irqchip/irq-gic-v3.c`	i.MX 8 SoCs with GIC-500 i.MX 93 and i.MX 91 with GIC-600 i.MX 95 and i.MX 943 with GIC-700
`drivers/irqchip/irq-imx-irqsteer.c`	Interrupt functions with `CONFIG_IMX_IRQSTEER` configuration
`drivers/irqchip/irq-imx-intmux.c`	Interrupt functions with `CONFIG_IMX_INTMUX` configuration
`irq-imx-gpcv2.c`	Interrupt functions with `CONFIG_IMX_GPCV2` configuration

2.1.2.3 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

2.1.2.4 Interrupt source code structure

The interrupt module is located in `drivers/irqchip`.

The table below lists the source files for interrupts.

Table 4. Interrupt files

File	Description
`drivers/irqchip/irq-imx-irqsteer.c`	Interrupt functions with `CONFIG_IMX_IRQSTEER` configuration
`drivers/irqchip/irq-imx-gpcv2.c`	Interrupt functions with `CONFIG_IMX_GPCV2` configuration
`drivers/irqchip/irq-imx-intmux.c`	Interrupt functions for with `CONFIG_IMX_INTMUX` configuration

2.1.2.5 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 from the Interrupt Controller, the number of interrupts is also expanded to support GPIO interrupt and (on some platforms) EDIO interrupts. This allows drivers to use the standard interrupt interface supported by Arm device running Linux OS, such as the `request_irq()` and `free_irq()` functions.

2.1.3 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 following table describes the different timers used.

Table 5. Timers

Timer	Description
General Purpose Timer (GPT)	GPT is configured to generate a periodic interrupt at a certain interval (every 10 ms). Used by i.MX 6 to go into WFI mode. Used by i.MX 6 and i.MX 7.
Enhanced Periodic Interrupt Timer (EPIT)	Available on i.MX 6 and i.MX 7.
Arm Arch Timer	i.MX 8 and i.MX 9 usage instead of GPT.
System Counter Timer	i.MX 8M, i.MX 8X, and i.MX 9 usage instead of GPT.

2.1.3.1 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 serve the OS for the purposes mentioned in Section 2.1.3. Another function provides the time elapsed as the last timer interrupt.

2.1.3.2 Timer features

The timer implementation supports the following features:

- Functions required by the Linux OS to provide the system timer and dynamic timers.
- Generates an interrupt every 10 ms for i.MX 6 and i.MX 7, and every 4 ms for i.MX 8 and i.MX 9. This is based on `CONFIG_HZ_XXX`.

2.1.3.3 Timer source code structure

Table 6. Timer files

File	Description
`arch/arm/mach-imx/epit.c`	Enhanced Periodic Interrupt Timer
`driver/clocksource/timer-imx-sysctr.c`	System Controller Timer
`driver/clocksource/timer-imx-tpm.c`	TPM Timer
`drivers/clocksource/timer-imx-gpt.c`	General Purpose Timer
`drivers/clocksource/arch-arm-timer.c`	Arm arch Timer

2.1.3.4 Timer programming interface

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

This is done with the `clocksource_mxc` structure of the `struct clocksource` type and `clockevent_mxc` structure of the `struct clockevent_device` type. Both structures provide routines required for reading the current timer values and scheduling the next timer event. The module implements a timer interrupt routine that serves the Linux OS with timer events for the purposes mentioned in the beginning of this chapter.

2.1.4 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.

2.1.4.1 Memory Map hardware operation

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

2.1.4.2 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.

2.1.5 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 the 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 the GPIO input pin with interrupt enabled. In addition, the software does not have control over what function a pin should have. The software only configures pin usage according to the system design.

2.1.5.1 IOMUX hardware operation

This section 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. Also enables loopback mode when applicable.
- `SW_SELECT_INPUT`: Controls the pin input path. This register is only required when multiple pads drive the same internal port.
- `SW_PAD_CTL`: Controls the pad slew rate, driver strength, pull-up/down resistance, and so on.

2.1.5.2 IOMUX software operation

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

2.1.5.3 IOMUX features

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

2.1.5.4 IOMUX source code structure

Table below lists the source files for the IOMUX module. The files are in the `drivers/princtrl/freescale` folder.

Table 7. IOMUX files

File	Description
`drivers/pinctrl/freescale/pinctrl-imx.c`	i.MX `pinctrl` core driver
`drivers/pinctrl/freescale/pinctrl-imx6q.c`	i.MX 6Quad/DualLite `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx6sx.c`	i.MX 6SoloX `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx6sll.c`	i.MX 6SLL `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx6ul.c`	i.MX 6UltraLite and 6ULL `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx7d.c`	i.MX 7Dual `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx7ulp.c`	i.MX 7ULP `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx8qm.c`	i.MX 8QuadMax `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx8qxp.c`	i.MX 8QuadXPlus `pinctrl` driver

Table 7. IOMUX files*...continued*

File	Description
`drivers/pinctrl/freescale/pinctrl-imx8mq.c`	i.MX 8M Quad `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx8mm.c`	i.MX 8M Mini `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx8mn.c`	i.MX 8M Nano `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx8ulp.c`	i.MX 8ULP `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx93.c`	i.MX 93 `pinctrl` driver
`drivers/pinctrl/freescale/pinctrl-imx-scmi.c`	i.MX 943 and i.MX 95 `pinctrl` driver

2.1.5.5 IOMUX programming interface

See the `pinctrl` binding documents: `Documentation/devicetree/bindings/pinctrl/fsl,imx*.[yaml,txt]`.

2.1.5.6 IOMUX control through GPIO module

For a multipurpose 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 information 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 the purpose that cannot be changed by the 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 the 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.

2.1.5.6.1 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.

2.1.5.6.1.1 Muxing Control

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

The settings in these registers choose if a pin is used 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 the IOMUX block of the GPIO.

2.1.5.6.1.2 PULLUP control

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

2.1.5.6.2 GPIO software operation (general)

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

2.1.5.6.3 GPIO implementation

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

2.1.6 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.

2.1.6.1 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 in the MSL layer to provide better portability and maintainability. This GPIO interrupt is initialized automatically during the system startup.

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

2.1.6.1.1 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 are required for the public drivers such as Ethernet and UART drivers as no special GPIO function call is needed for registering an interrupt.

2.1.6.2 GPIO features

This GPIO implementation supports the following features:

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

2.1.6.3 GPIO module source code structure

All of the GPIO module source code is in the GPIO framework in the following files, located in the directories indicated at the beginning of this chapter.

Table 8. GPIO files

File	Description
`drivers/gpio/gpio-mxc.c`	Function implementation on i.MX 6, i.MX 7Dual, i.MX 8M Nano, i.MX 8M Mini, i.MX 8M Quad, i.MX 8QuadXPlus, i.MX 8QuadMax, and i.MX 8DXL
`drivers/gpio/gpio-vf610.c`	Function implementtation on i.MX 7ULP, i.MX 8ULP, and i.MX 9

2.1.6.4 GPIO programming interface

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

2.1.7 Clock

The Linux clock framework relies on the underlying hardware to provide support for clock tree management.

The following table describes different clock hardware used.

Table 9. Clock

File	Description
Clock controller module (CCM)	i.MX 6Quad/DualLite/SoloX/UltraLite/ULL/SLL, i.MX 7Dual, i.MX 8M Quad, i.MX 8M Mini, i.MX 8M Nano, i.MX 8M Plus, and i.MX 93
Peripheral clock control (PCC) and System clock generator (SCG)	i.MX 7ULP, i.MX 8ULP
Distributed slave system controller (DSC)	i.MX 8QuadMax/8QuadXPlus

2.1.7.1 Clock software operation

The clock software implementation provides an initialization function that initializes the clock tree according to hardware clock type and settings, and then provides clock operation callbacks to operate the hardware clock module.

2.1.7.2 Clock features

The clock implementation supports the following features according to different clock types:

- Prepare/Unprepare a clock.
- Enable/Disable a clock.
- Get/Set the clock rate.
- Get/Set the clock parent.

2.1.7.3 Clock source code structure

The source code structure is as follows.

Table 10. Clock source code structure

File	Description
`drivers/clk/imx/clk-imx6q.c`	i.MX 6Quad/6DualLite clock driver
`drivers/clk/imx/clk-imx6sx.c`	i.MX 6SoloX clock driver
`drivers/clk/imx/clk-imx6ul.c`	i.MX 6UltraLite and 6ULL clock driver
`drivers/clk/imx/clk-imx6sll.c`	i.MX 6SLL clock driver
`drivers/clk/imx/clk-imx7d.c`	i.MX 7Dual clock driver
`drivers/clk/imx/clk-imx7ulp.c`	i.MX 7ULP clock driver
`drivers/clk/imx/clk-imx8qm.c`	i.MX 8QuadMax clock driver
`drivers/clk/imx/clk-imx8qxp.c`	i.MX 8QuadXPlus clock driver
`drivers/clk/imx/clk-imx8mq.c`	i.MX 8M Quad clock driver
`drivers/clk/imx/clk-imx8mm.c`	i.MX 8M Mini clock driver
`drivers/clk/imx/clk-imx8mn.c`	i.MX 8M Nano clock driver
`drivers/clk/imx/clk-imx8ulp.c`	i.MX 8ULP clock driver
`drivers/clk/imx/clk-imx93.c`	i.MX 93 clock driver
`drivers/clk/clk-scmi.c`	i.MX 95 and i.MX 943

2.1.7.4 Clock programming interface

Different clock types provide different clock operation callbacks. Device drivers call standard clock APIs to clock framework and eventually call into the platform clock driver, and the corresponding clock node's operation callback is executed.

2.2 System Controller

2.2.1 Introduction

The System Controller is implemented on i.MX 8 and i.MX 8X families, provides an abstraction to many underlying features of the hardware, and runs on a Cortex-M processor that executes SC firmware (SCFW). This section describes the features of the SCFW and APIs exposed to other software components.

The System Controller features include:

- System Initialization and Boot: The SC firmware runs on the SCU immediately after the SCU Read-only-memory (ROM) finishes loading code/data images from the first container. It is responsible for initializing many aspects of the system. This includes additional power and clock configuration and resource isolation hardware configuration. By default, the SC firmware configures the primary boot core to own most of the resources and launches the boot core. Additional configuration can be done by the boot code.
- System Controller Communication: Other software components in the system communicate to the SC through an exposed API library. This library is implemented to make Remote Procedure Calls (RPC) through an underlying Inter-Processor Communication (IPC) mechanism. The IPC is facilitated by a hardware-based mailbox system. Software components (Linux, QNX, FreeRTOS, MCUXpresso SDK) delivered for i.MX8 already include ports of the client API. Other third parties need to port the API to their environment before the API can be used. The porting kit release includes archives of the client API for existing SW. These can be

used as reference for porting the client API. All that needs to be implemented is the IPC layer, which uses the messaging units (MU) to communicate with the SCFW.

- Power Management: All aspects of power management including power control, bias control, clock control, reset control, and wake-up event monitoring are grouped within the SC Power Management service.
  - **Power Control**: The SC firmware is responsible for centralized management of power controls and external power management devices. It manages the power state and voltage of power domains as well as bias control. It also resets peripherals as required due to power-state transitions. This is implemented with the API by communicating power state needs for individual resources.
  - **Clock Control**: The SC firmware is responsible for centralized management of clock controls. This includes clock sources such as oscillators and PLLs as well as clock dividers, muxes, and gates. This is implemented with the API by communicating clocking needs for individual resources.
  - **Reset Control**: The SC firmware is responsible for reset control. This includes booting/rebooting a partition, obtaining reset reasons, and starting/stopping of CPUs.

  Before any hardware in the SoC can be used, SW must first power up the resource and enable any clocks that it requires. Otherwise, access may lead to a bus error.
- Resource Management: The SC firmware is responsible for managing ownership and access permissions to system resources. The features of the resource management service supported by SC firmware include:
  - Management of system resources such as SoC peripherals, memory regions, and pads.
  - Allows resources to be partitioned into different ownership groupings that are associated with different execution environments including multiple operating systems executing on different cores, TrustZone, and hypervisor.
  - Associates ownership with requests from messaging units within a resource partition.
  - Allows memory to be divided into memory regions that are then managed like other resources.
  - Allows owners to configure access permissions to resources.
  - Configures hardware components to provide hardware enforced isolation.
  - Configures hardware components to directly control secure/nonsecure attribute driven on bus fabric.
  - Provides ownership and access permission information to other system controller functions (for example, pad ownership information to the pad muxing functions).
  - Protection of resources is provided in two ways:
    - The SCFW itself checks resource access rights when API calls are made that affect a specific resource. Depending on the API call, this may require that the caller be the owner, parent of the owner, or an ancestor of the owner.
    - Any hardware available to enforce access controls is configured based on the RM state. This includes the configuration of IP, such as XRDC2, XRDC, or RDC, as well as management pages of IP like CAAM.
- Pad Configuration: Pad configuration is managed by the SC firmware. The pad configuration features supported by the SC firmware include:
  - Configuring the mux, input/output connection, and low-power isolation mode.
  - Configuring the technology-specific pad setting such as drive strength and pullup/pulldown.
  - Configuring compensation for pad groups with dual-voltage capability.
- Timers: Many timer oriented services are grouped within the SC Timer service. This includes watchdogs, RTC, and system counter.
  - **Watchdog**: The SC firmware provides "virtual" watchdogs for all execution environments. Features include update of the watchdog timeout, start/stop of the watchdog, refresh of the watchdog, return of the watchdog status such as maximum watchdog timeout that can be set, watchdog timeout interval, and watchdog timeout interval remaining.
  - **Real-Time-Clock**: The SC firmware is responsible for providing access to the RTC. Features include setting the time, getting the time, and setting alarms.
  - **System Counter**: The SC firmware is responsible for providing access to the SYSCTR. Features include setting an absolute alarm or a relative, periodic alarm. Reading is done directly through the local hardware interfaces available for each CPU.

- Interrupts: The System Controller needs a method to inform users about asynchronous notification events. This is done through the Interrupt service. The service provides APIs to enable/disable interrupts to the user and to read the status of pending interrupts. Reading the status automatically clears any pending state.
- Miscellaneous: On previous i.MX 6 and 7 devices, miscellaneous features were controlled using IOMUX GPR registers with signals connected to configurable hardware. This functionality is being replaced with DSC GPR signals. The SC firmware is responsible for programming the GPR signals to configure these subsystem features. The SC firmware also responsible for monitoring various temperature, voltage, and clock sensors.
  - **Controls**: The SC firmware provides access to miscellaneous controls. Features include software request to set (write) miscellaneous controls and software request to get (read) miscellaneous controls.
  - **Security**: The SC firmware provides access to several security functions including image loading and authentication.
  - **DMA**: The SC firmware provides access to DMA channel grouping and priority functions.
  - **Temp**: The SC firmware provides access to temperature sensors.

With this abstraction, some hardware described in the SoC Reference Manual that is used by the SCFW is not directly accessible to other cores. This includes the following:

- All resources in the SCU subsystem (SCU M4, SCU LPUART, SCU LPI2C, etc.).
- All resource accessed through the MSI links from the SCU subsystem (including pads, DSC, XRDC2, eCSR)
- OCRAM controller, CAAM MP, eDMA MP, and LPCG.
- DB STC and LPCG, IMG GPR.
- GIC/IRQSTR LPCG, IRQSTR.SCU, and IRQSTR.CTI.
- Any other resources reserved by the port of the SCFW to the board.

The System Controll firmware (SCFW) provided with each release works with the corresponding i.MX reference boards. A porting kit is also provided, containing a subset of source that can be customized for new boards. This porting kit is available on nxp.com and includes a porting guide.

2.3 Boot Image

2.3.1 Introduction

For i.MX 6 and i.MX 7, the boot image uses only the U-Boot bootloader. For the SoC in the i.MX 8 and i.MX 9 series, the boot image is more complex and includes U-Boot and various firmware required for a successful boot. This section describes the additional components for an i.MX 8 series bootloader.

For i.MX 7ULP, the boot partition requires the Arm Cortex M-4 SDK flash because the Arm Cortex M-4 boots the U-Boot bootloader, but other i.MX 6 and i.MX 7 with Arm Cortex M-4 cores do not require this for successful boot.

The i.MX 8 and i.MX 9 bootloaders are created using the `imx-mkimage` tool available on imx-mkimage on github.com/nxp-imx/ and all i.MX 8 series require the Arm trusted firmware available on imx-atf on github.com/nxp-imx/.

For details on how to use the `imx-mkimage` tool to create an i.MX boot partition, see the *i.MX Linux User's Guide* (UG10163). This tool for execution requires the following components.

For i.MX 8M Quad, i.MX 8M Mini, and i.MX 8M Nano, the following firmware is needed:

- Synopsys DDR firmware.
- Signed HDMI firmware, which integrates with the DCSS driver. The HDMI firmware is for i.MX 8M Quad only.
- Arm Trusted firmware – `bl31-*soc*`.

For i.MX 8QuadMax, the following firmware is needed:

- System Controller Firmware (SCFW).

- Arm Trusted firmware – `bl31-*soc*`.
- SECO firmware container image (`ahab-container.img`) for B0.

For i.MX 8QuadXPlus, i.MX 8DualX, and i.M 8DualXLite, the following firmware is needed:

- System Controller Firmware (SCFW).
- Arm Trusted firmware – `bl31-*soc*`.
- SECO firmware container image (`ahab-container.img`).

For i.MX 93 and i.MX 91, the following firmware is needed:

- Synopsys DDR firmware.
- Arm Trusted firmware – `bl31-*soc*`.
- ELE firmware container.

For i.MX 95 and i.MX 943, the following firmware is needed:

- Synopsys DDR firmware.
- Arm Trusted firmware.
- OEI firmware.
- ELE firmware container.
- System manager firmware.

All the i.MX series require the Arm trusted firmware and U-Boot. The i.MX SoCs (all i.MX 6, 7, and 8M families) supporting OP-TEE enabled with OP-TEE boot requires that the `tee.bin` should be created from building `optee_ox`.

Type 2 hypervisors, such as Jailhouse and KVM, are not part of the bootloader. Type 1 hypervisors are part of the loader however Xen and only support i.MX 95 19x19 EVK.

2.4 Anatop Regulator Driver

2.4.1 Introduction

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

This device driver uses the regulator core driver to access the Anatop hardware control registers and is only supported on i.MX 6 and i.MX 7.

2.4.2 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 changes depending on the type of the external memory selected. Other supplies may 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 regulator can be bypassed, so that the external PMIC can supply power directly to decrease the power number, such as VDD_SOC and VDD_ARM.

2.4.3 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.

2.4.4 Driver features

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

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

2.4.5 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.

2.4.6 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 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, see opensource.wolfsonmicro.com/node/15.

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

- `regulator_get` used to look up 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 the regulator output voltage:

  -
    ```
    int regulator_set_voltage(struct regulator *regulator, int uV);
    ```
- `regulator_get_voltage` used to get the 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: `drivers/regulator/core.c`.

2.4.7 Source code structure

The Anatop regulator driver is located in the `drivers/regulator` directory:

Table 11. Anatop power management driver files

File	Description
`drivers/regulator/core.c`	Regulator interface
`drivers/regulator/anatop-regulator.c`	Anatop regulator client driver

The Anatop regulators are registered in each SoC-specific DTS file in `arch/arm/boot/dts`.

2.4.8 Menu configuration options

In menu configuration, enable the following modules:

- **Device Drivers** -> **Voltage and Current regulator support** -> **Anatop Regulator Support**.
- **System Type** -> **Freescale i.MX on-chip ANATOP LDO regulators**.

2.5 Power Management

2.5.1 Low Level Power Management (PM)

2.5.1.1 Introduction

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

The following table lists the differences between how power management is handled for each supported i.MX family.

Table 12. Power Management modes

i.MX family	Supported low power mode
i.MX 6	RUN, WAIT, STOP, and DORMANT
i.MX 7	RUN, WAIT, STOP, DORMANT, and LPSR
i.MX 8M	RUN, IDLE, SUSPEND, and SNVS
i.MX 8, i.MX 8X	None - handled by the System Controller
i.MX 8ULP	ACTIVE, SLEEP, Power Down, and Deep Power Down
i.MX 91, i.MX 93, i.MX 95, i.MX 943	RUN, IDLE, SUSPEND, and BBSM

Note: *i.MX 8ULP has additional low power modes: Partial Active and Deep Sleep modes. The two modes are not used in Linux OS, because no typical software power mode can be used to support them.*

The following table lists the detailed clock information for different low power modes.

Table 13. Low Power Modes

Mode	Core	Module	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	On	On
LPSR	Power off	Disable	Off	Off	On
DORMANT	Power off	Disable	Off	Off	On
SNVS	Power off	Disable	Off	Off	On

For detailed information about low power modes, see the Applications Processor Reference Manual associated with the SoC.

2.5.1.2 Software operation

The i.MX 6 and i.MX 7 power management driver maps the low-power modes to the kernel power management states as listed below:

- Standby-maps to STOP mode, which offers significant power saving, as all blocks in the system are put into a low-power state, except for the Arm core, which is still powered on, and memory is placed in self-refresh mode to retain its contents.
- Mem (suspend to RAM) maps to DORMANT mode, which offers most significant power saving, as all blocks in the system are put into a low-power state, except for memory, which is placed in self-refresh mode to retain its contents. If there is `fsl,enable-lpsr` defined in DTB OCRAM node, mem is mapped to LPSR mode instead of DORMANT, and all the blocks in the system are put into power off state, except the LPSR, SNVS, and DRAM power domains.
- System idle maps to WAIT mode.
- If the Arm Cortex-M4 processor is alive together with the Arm Cortex-A processor before the kernel enters standby/mem mode, and if Arm Cortex-M4 processor is not in its low-power idle mode, the Arm Cortex-A processor triggers the SoC to enter WAIT mode instead of STOP mode to make sure that Arm Cortex-M4 processor can continue running.

The i.MX 6 and i.MX 7 power management driver performs the following steps to enter and exit low-power mode:

1. Allow the Cortex-A platform to issue a deep sleep mode request.
2. In STOP or DORMANT mode:
   - Program i.MX 6 `CCM_CLPCR` or i.MX 7 `GPC_LPCR_A7_BSC` and `GPC_SLPCR` registers to set low-power control register.
   - In DORMANT mode, request switching off the CPU power when `pdn_req` is asserted.
   - Request switching off the embedded memory peripheral power when `pdn_req` is asserted.
   - Program the GPC mask register to unmask wakeup interrupts.
3. Call `cpu_do_idle` to execute WFI pending instructions for wait mode.
4. Execute `imx6_suspend` or `imx7_suspend` in IRAM.
5. In DORMANT mode, save the Arm context, and change the drive strength of DDR PADs as "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. In DORMANT mode, restore the Arm core and DDR drive strength.

In DORMANT mode, the i.MX 6 and i.MX 7 can assert the `PMIC_STBY_REQ` pin to the PMIC and request a voltage change. The U-Boot or Machine-Specific Layer (MSL) usually sets the standby voltage in STOP mode according to i.MX 6 and i.MX 7 data sheet.

On i.MX 8M family, the power management driver uses the following modes:

- RUN mode: In this mode, the Quad-A53 CPU core is active and running. Some portions can be shut off for power saving.
- IDLE mode: This mode is defined as a mode, which CPU can automatically enter when there is no thread running and all high-speed devices are not active. The CPU can be put into power gated state but with L2 data retained, DRAM and bus clock are reduced, and most of the internal logics are clock gated but still remain powered.
- SUSPEND mode: This mode is defined as the most power saving mode where all the clocks are off and all the unnecessary power supplies are off. Cortex-A53 CPU platform is fully power gated. All the internal digital logics and analog circuits that can be powered down are off.
- SNVS mode: This mode is also called RTC mode. In this mode, only the power for the SNVS domain remains on to keep RTC and SNVS logic alive.

On i.MX 8 and i.MX 8X:

- Low-power mode management is not controlled by a dedicated hardware block.
- All low-power modes are implemented in system controller firmware (SCFW) using software method.
- SCFW powers off clusters/CPUs when the system is suspended.

On i.MX 8ULP:

- uPower is responsible for controlling the power mode transition, Power switch, and memory ON/OFF.
- For a detailed power mode description, see the Applications Processor Reference Manual associated with SoC.

On i.MX 9 family:

- RUN mode: In this mode, the Cortex-A55 CPU core is active and running. Some portions can be shut off for power saving.
- IDLE mode: This mode is defined as a mode, which CPU can automatically enter when there is no thread running and all high-speed devices are not active. The CPU can be put into power gated state but with L3 memory retained. DDR can be put into auto clock gating, power switchable MIX can be power off and most of the internal logics are clock gated but still remain powered.

- SUSPEND mode: This mode is defined as the most power saving mode where all the clocks are off and all the unnecessary power supplies are off. Cortex-A55 CPU platform is power gated. All the internal digital logics and analog circuits that can be powered down are off.
- BBSM mode: This mode is also called RTC mode. In this mode, only the power for the BBSM domain remains on to keep RTC and BBSM logic alive.

2.5.1.3 Source code structure

The table below lists the Power Management driver source files.

Table 14. Power Management driver files

File	Description
• `arch/arm/mach-imx/pm-imx6.c` • `arch/arm/mach-imx/suspend-imx6.S` • `arch/arm/mach-imx/cpuidle-imx6q.c`	Supports i.MX 6QuadPlus/Quad/Dual/Solo power management operation.
• `arch/arm/mach-imx/pm-imx6.c` • `arch/arm/mach-imx/suspend-imx6.S` • `arch/arm/mach-imx/cpuidle-imx6sll.c` • `arch/arm/mach-imx/imx6sll_low_power_idle.S`	Supports i.MX 6SLL power management operation.
• `arch/arm/mach-imx/pm-imx6.c` • `arch/arm/mach-imx/suspend-imx6.S` • `arch/arm/mach-imx/cpuidle-imx6ul.c` • `arch/arm/mach-imx/imx6ul_low_power_idle.S`	Supports i.MX 6UltraLite power management operation.
• `arch/arm/mach-imx/pm-imx6.c` • `arch/arm/mach-imx/suspend-imx6.S` • `arch/arm/mach-imx/cpuidle-imx6ul.c` • `arch/arm/mach-imx/imx6ull_low_power_idle.S`	Supports i.MX 6ULL power management operation.
• `arch/arm/mach-imx/pm-imx6.c` • `arch/arm/mach-imx/suspend-imx6.S` • `arch/arm/mach-imx/cpuidle-imx6sx.c` • `arch/arm/mach-imx/imx6sx_low_power_idle.S`	Supports i.MX 6SoloX power management operation.
• `arch/arm/mach-imx/pm-imx7.c` • `arch/arm/mach-imx/suspend-imx7.S` • `arch/arm/mach-imx/cpuidle-imx7d.c` • `arch/arm/mach-imx/imx7d_low_power_idle.S`	Supports i.MX 7Dual power management operation.
• `arch/arm/mach-imx/pm-imx7ulp.c` • `arch/arm/mach-imx/suspend-imx7ulp.S` • `arch/arm/mach-imx/cpuidle-imx7.c`	Supports i.MX 7ULP power management operation.
• `drivers/pmdomain/imx/gpcv2.c` • `drivers/pmdomain/imx/imx8m*-blk-ctrl.c`	Supports i.MX 8M power domains.
• `driver/soc/imx/imx8ulp_lpm.c`	Supports i.MX 8ULP system level voltage and frequency scaling.
• `driver/soc/imx/imx93_lpm.c`	Supports i.MX 93 and i.MX 91 system level voltage and frequency scaling, and DDR auto clock gating control.
Arm Trusted firmware exists in imx-atf on github.com/nxp-imx/.	Supports i.MX 8, 8X, 8M, 8ULP, i.MX 93, i.MX 91, i.MX 95, and i.MX 943 to use Arm trusted firmware for power management operation.

2.5.1.4 Menu configuration options

In menu configuration, enable **CONFIG_PM**: **CONFIG_PM** builds support for power management. By default, this option is `Y` In `menuconfig`. This option is available under: **Power management options** -> **Power Management support**.

In menu configuration, enable **CONFIG_SUSPEND**. **CONFIG_SUSPEND** builds support for suspend. In `menuconfig`, this option is available under: **Power management options** -> **Suspend to RAM and standby**.

2.5.1.5 Programming interface

View the `cpu_idle` for each SoC as shown in the source code structure table and search for `lpm`. This is the API for lower-power mode. This implements all the steps required to put the system into WAIT and STOP modes.

2.5.2 PMIC PF regulator

2.5.2.1 Introduction

PF100/200/300 is a PMIC chip. PF200/PF3000 is based on PF100 with little change, since they share the same PF100 driver. 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 uses the PF100 regulator driver to access the PF100 hardware control registers. The PF100 regulator driver is based on the regulator core driver and it is attached to the kernel I2C bus.

PF8100/8200 PMIC is designed for i.MX 8 and i.MX 8X families and is controlled by system controller firmware (SCFW) since it is a system-level device. SCFW creates some specific power resource for the Linux touch, such as "SC_R_BOARD_R0".

2.5.2.2 Hardware operation

The PMIC PF regulator 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/or other circuitry.

Linear regulators are directly supplied from the battery or from the switchers and include supplies for I/O and peripherals, audio, camera, BT, WLAN, and so on. Naming conventions are suggestive of typical or possible use case applications, but the switchers and regulators may be 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. The PMIC_ON_REQ pin of i.MX 6, which is controlled by the SNVS block of i.MX 6, connects with the PWRON pin of PF100 to control PF100 on/off, so that the system can be powered off.

2.5.2.3 Software operation

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

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, 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.

2.5.2.4 Driver features

The PMIC PF regulator driver is based on the 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.

2.5.2.5 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, see opensource.wolfsonmicro.com/node/15.

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

- `regulator_get` is a unified API call to look up and obtain a reference to a regulator:
  ```
  struct regulator *regulator_get(struct device *dev, const char *id);
  ```
- `regulator_put` is a unified API call to free the regulator source:
  ```
  void regulator_put(struct regulator *regulator, struct device *dev);
  ```
- `regulator_enable` is a unified API call to enable regulator output:
  ```
  int regulator_enable(struct regulator *regulator);
  ```
- `regulator_disable` is a 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 a unified API call to set regulator output voltage:
  ```
  int regulator_set_voltage(struct regulator *regulator, int uV);
  ```
- `regulator_get_voltage` is a 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: `drivers/regulator/core.c`.

2.5.2.6 Driver architecture

The following figure shows the basic architecture of the PMIC PF regulator driver.

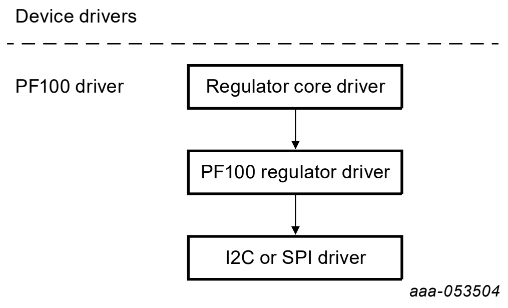

Figure 1. PMIC PF regulator driver architecture

2.5.2.7 Driver interface details

Access to the PFUZE100 regulator is provided through the API of the regulator core driver.

The PFUZE100 regulator driver provides the following regulator controls:

- 4 buck switch regulators in 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 (`PFUZE100_SWxSTANDBY`) in advance.
- 6 Linear Regulators: VGEN1, VGEN2, VGEN3, VGEN4, VGEN5, and VGEN6.
- 1 LDO/Switch supply for VSNVS support on i.MX processors.
- 1 low current, high accuracy, voltage reference for DDR Memory reference voltage.
- 1 Boost regulator with USB OTG support.
- Most power rails from PFUZE100 have been programmed properly according to the hardware design. Therefore, you cannot find the kernel using PFUZE100 regulators. The PFUZE100 regulator driver has implemented these regulators so that customers can use it freely if the default PFUZE100 value does not meet their hardware design.

2.5.2.8 Source code structure

The PFUZE regulator driver is located in the `drivers/regulator` directory:

Table 15. PFUZE driver files

File	Description
`drivers/regulator/pfuze100-regulator.c`	Implementation of the PFUZE100 regulator client driver.
`drivers/regulator/pf1550.c`	Implementation of the PFUZE1550 regulator client driver.
`drivers/regulator/pf1550-regulator-rpmsg.c`	Implementation of the PFUZE150 regulator RPMSG code.

There is no board file related to PMIC. Some PFUZE driver code has been moved to U-Boot, such as standby voltage setting. Some code is implemented by the DTS file. Search for PFUZE100 in the U-Boot source and PFUSE in device trees DTSI files for i.MX 6 and i.MX 7 in `arch/arm/boot/dts` and for i.MX 8M in `arch/arm64/boot/dts`.

2.5.2.9 Menu configuration options

In menu configuration, enable the following module:

Device Drivers -> **Voltage and Current regulator support** -> **Freescale PFUZE100/200/3000 regulator driver**.

2.5.3 CPU Frequency Scaling (CPUFREQ)

2.5.3.1 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 voltages of the necessary power supplies are changed to the voltage value defined in device tree scripts (DTS). This method can reduce power consumption (therefore saving battery power), because the CPU uses less power as the clock speed is reduced.

2.5.3.2 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 DTS, 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. Interactive CPU frequency governor is used, which cannot be changed manually. To change the CPU frequency manually, the userspace CPU frequency governor can be used. By default, the conservative CPU frequency governor is used.

See 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 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 the 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
```

2.5.3.3 Source code structure

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

Table 16. CPUFREQ driver files

File	Description
`drivers/cpufreq/imx6q-cpufreq.c`	i.MX 6 CPUFreq functions
`drivers/cpufreq/imx-cpufreq-dt.c`	i.MX 7 and 8 CPUFreq functions

For CPU frequency working point settings, see the SoC corresponding DTSI file in `arch/arm/boot/dts` for i.MX 6 and i.MX7 and `arch/arm64/boot/dts` for i.MX 8, i.MX 8X, and i.MX 8M. For i.MX 95 and i.MX 943, the CPUFreq OPP information is provided through the SCMI performance protocol dynamically. No static OPP is defined in DTS.

2.5.3.4 Menu configuration options

The following Linux kernel configuration is provided for this module:

- **CONFIG_CPU_FREQ**
  In `menuconfig`, this option is located under:
  – **CPU Power Management** -> **CPU Frequency scaling**

The following options can be selected:

- CPU Frequency scaling
- CPU frequency translation statistics
- Default CPU frequency governor (conservative) (interactive)
- Performance governor
- Powersave governor
- Userspace governor for userspace frequency scaling
- Interactive CPU frequency policy governor
- Conservative CPU frequency governor
- Schedutil CPU frequency governor
- CPU frequency driver for i.MX CPUs

2.5.4 Dynamic Bus Frequency

2.5.4.1 Introduction

To improve the power consumption, the Bus Frequency driver dynamically manages the various system frequencies for i.MX 6, i.MX 7, and i.MX 8M families.

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

2.5.4.2 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 frequencies of the system clocks (like AHB, DDR, and AXI) based on peripheral activity and CPU loading.

2.5.4.3 Software operation

The Bus Frequency depends on the request and release of device drivers for its operation. Drivers call Bus Frequency APIs to request or release the bus setpoint as needed. The Bus Frequency sets the system frequency to the highest frequency setpoint based on the peripherals that are currently requesting.

To enable the Bus Frequency driver, use the following command:

```
echo 1 > /sys/bus/platform/drivers/imx_busfreq/soc\:busfreq/enable
```

To disable the Bus Frequency driver, use the following command:

```
echo 0 > /sys/bus/platform/drivers/imx_busfreq/soc\:busfreq/enable
```

If the Arm Cortex-M4 processor is alive with Arm Cortex-A processor together, the Arm Cortex-M4 processor also requests or releases Bus Frequency high setpoint for its operation. This means that Arm Cortex-A processor treats Arm Cortex-M4 processor as one of its high-speed devices.

The setpoint modes perform the following:

- High Frequency Setpoint mode is used when most peripherals that need higher frequency for good performance are active. For example, video playback and graphics processing.
- Audio Playback setpoints mode is used in audio playback mode.
- Low Frequency setpoint mode is used when the system is idle waiting for user input (display is off). For i.MX 8M, this mode is used when no peripheral device requests high mode or audio mode.

The following table lists the software setpoints for each Family.

Table 17. BusFrequency setpoints

SoC	Setpoints
i.MX 6	• High Frequency Setpoint: AHB is at 132 MHz, and AXI is at 264 MHz. • Audio Playback setpoints: On i.MX 6, AHB is at 25 MHz, AXI is at 50 MHz, and DDR is at 50 MHz for DDR3 and 100 MHz for LPDDR2. • Low Frequency setpoint: AHB is at 24 MHz, AXI is at 24 MHz, and DDR is at 24 MHz.
i.MX 7Dual	• High Frequency Setpoint: AHB is at 135 MHz, AXI is at 332 MHz, and DDR is at the maximum frequency. • Audio Playback setpoints: AHB is at 24 MHz, AXI is at 24 MHz, and DDR is at 100 MHz. • Low Frequency setpoint: AHB is at 24 MHz, AXI is at 24 MHz, and DDR is at 24 MHz.
i.MX 8M	• High Bus Frequency mode: The DDRC core clock is set to 800 MHz. The DDRC APB clock is set to 200 MHz. The NOC clock is set to 800 MHz. The main AXI cock is set to 333 MHz, and the AHB clock is set to 133 MHz. • Audio Bus Frequency mode: The DDRC core clock is set to 25 MHz. The DDRC APB clock is set to 20 MHz, the NOC clock is set to 100 MHz. The main AXI clock is set to 25 MHz, and the AHB clock is set to 20 MHz. The DDR PLL is powered down for power saving. • Low Bus Frequency mode: The DDRC core clock is set to 25 MHz. The DDRC APB clock is set to 20 MHz. The NOC clock is set to 100 MHz. The main AXI clock is set to 25 MHz. The AHB clock is set to 20 MHz. The DDR PLL is powered down for power saving.

2.5.4.4 Source code structure

The following table lists the source files and headers.

Table 18. Bus Frequency driver files

File	Description
`arch/arm/mach-imx/busfreq-imx.c`	i.MX 6 and i.MX 7 Bus Frequency functions
`include/linux/busfreq-imx.h`	i.MX Bus Frequency API Definitions
`arch/arm/mach-imx/busfreq_ddr3.c`	i.MX 6 and i.MX 7 DDR3 Bus Frequency functions
`arch/arm/mach-imx/busfreq_lpddr2.c`	i.MX 6 and i.MX 7 LPDDR2 Bus Frequency functions
`arch/arm/mach-imx/lpddr2_freq_imx6.S`	i.MX 6 LPDDR2 Bus Frequency functions
`arch/arm/mach-imx/lpddr2_freq_imx6q.S`	i.MX 6QuadPlus/Quad/Dual/Solo LPDDR2 Bus Frequency functions
`arch/arm/mach-imx/lpddr2_freq_imx6sll.S`	i.MX 6SLL LPDDR2 Bus Frequency functions
`arch/arm/mach-imx/lpddr2_freq_imx6sx.S`	i.MX 6SoloX LPDDR2 Bus Frequency functions
`arch/arm/mach-imx/lpddr3_freq_imx.S`	i.MX 6 and i.MX 7 LPDDR3 Bus Frequency functions
`arch/arm/mach-imx/ddr3_freq_imx6.S`	i.MX 6 Bus Frequency functions
`arch/arm/mach-imx/ddr3_freq_imx6sx.S`	i.MX 6SoloX Bus Frequency functions
`arch/arm/mach-imx/ddr3_freq_imx7d.S`	i.MX 7Dual DDR3 Bus Frequency functions
`drivers/soc/imx/busfreq-imx8mq.c`	i.MX 8M Bus Frequency functions
`driver/soc/imx/imx8ulp_lpm.c`	i.MX 8ULP system level voltage and frequency scaling
`driver/soc/imx/imx93_lpm.c`	i.MX 93 and i.MX 91 system level voltage and frequency scaling

Bus Frequency modes are defined in the SoC DTSI files in `arch/arm/boot/dts` for i.MX 6 and i.MX 7 and `arch/arm64/boot/dts` for i.MX 8M and i.MX 9.

On i.MX 8ULP, there is a simple interface to change the APD-side voltage and frequency scaling.

To enable the system level voltage and frequency scaling, use the following command:

```
echo 1 > /sys/device/platform/imx8ulp-lpm/enable
```

Note: *Before enabling this mode, the FEC and the display must be off, and the system is idle.*

To disable the system level voltage and frequency scaling, use the following command:

```
echo 0 > /sys/device/platform/imx8ulp-lpm/enable
```

For i.MX93 and i.MX 91 platforms, several dynamic power saving features can be supported with the flexible HW architecture. All these features can be used case by case to maximum dynamic power saving.

- OD/ND/LD mode
  Unlike the previous i.MX 8M family, there is no separate VDD_ARM power rail for the Cortex-A platform. A single VDD_SOC power rail is used for the whole digital logic in the SoC. The VDD_SOC can be nominal (ND), overdrive (OD), or a “low drive” (LD) voltage. Fine-grained DVFS is theoretically supported by the hardware (but not customer-exposed). In general, it is expected that many customers choose to statically operate at either nominal or low drive voltage (for lowest power) or overdrive voltage (for highest performance). However, dynamic voltage scaling is supported on a system-level “mode change” (for example,

if the entire chip goes into a low power standby mode while the display is updating a clock, but the system is otherwise idle and waiting for input).

- DDR frequency scaling
  On i.MX 93 and i.MX 91, the DDR subsystem has multiple frequency setpoints support. DDR can change the frequency dynamically to reduce the power consumption. There are two methods to change the DDR frequency dynamically: The HWFFC and SWFFC. The HWFFC switching flow is faster than the SWFFC. It has a very short latency. Normally, it can be used for DDR frequency scaling on the fly with the display on, but it can only be used when switching the frequency to half of the highest frequency. SWFFC may need to be used to switch the target frequency to a setpoint that is lower than half of the highest frequency.
- DDRC auto clock gating
  When the bus is idle after the number of cycles configured in the `ssi_idle_strap` field in the DDR BLK_CTRL module, the DDRC performs auto clock gating to save power. This feature can be used to balance the DDR subsystem performance and power significantly. The number of the idle cycles before clock gating can be adjusted dynamically based on the actual use case to fine tune the power saving.
- Software implementation
  On i.MX 93 and i.MX 91, the whole digital logic is supplied by a single VDD_SOC power rail. It is not that good to support conventional cortex-A platform DVFS through the Linux CPUFreq framework. Too many clock constraints need to be considered when changing the VDD_SOC voltage. To simplify the case, the voltage scaling is supported by a system wide mode switching (OD/ND/LD) to force all the clocks and voltage constrains. This is supported by using an NXP specific Linux kernel side module: `drivers/soc/imx/imx93_lpm.c`. This driver exports two user space control nodes through the `/sys` interface. The following figure shows the design details.

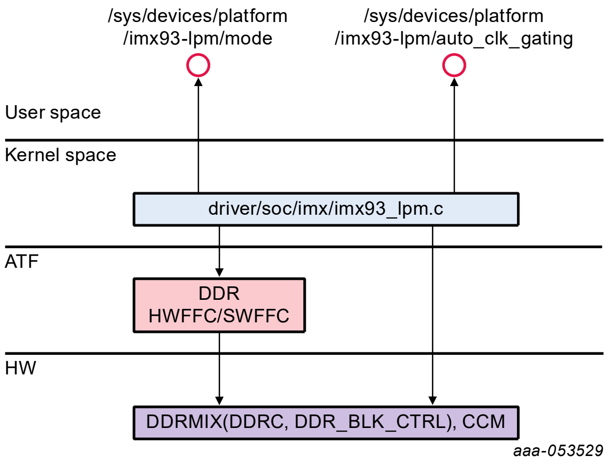
/sys/devices/platform /imx93-lpm/mode
/sys/devices/platform /imx93-lpm/auto_clk_gating
User space
Kernel space
driver/soc/imx/imx93_lpm.c
ATF
DDR HWFFC/SWFFC
HW
DDRMIX(DDRC, DDR_BLK_CTRL), CCM
aaa-053529

Figure 2. Software implementation

  – `mode`
    The mode control node is used for systemwide OD/ND/LD mode switching. For systemwide mode switching, the DDR frequency also needs to change dynamically to meet the voltage requirement, so when doing OD/ND mode switching, the DDR frequency also needs to be changed. For i.MX 93, the maximum DDR frequency that can be supported is 3733 MT/s. With HWFFC, the half speed is 1866 MT/s. This half-speed frequency can meet the OD, ND, and even LD mode clock/voltage requirement. That means we can use the HWFFC flow in OD/ND/LD mode switching to scaling the DDR frequency. DDR frequency can also be changed with SWFFC to other frequency lower than the half speed of the maximum frequency. If the DDR frequency has been changed to half speed through HWFFC, there is no way to directly change to other frequency through SWFFC, and vice versa. DDR frequency must be changed back to the highest frequency, and then be changed to the target frequency with HWFFC or SWFFC. See the following figure.

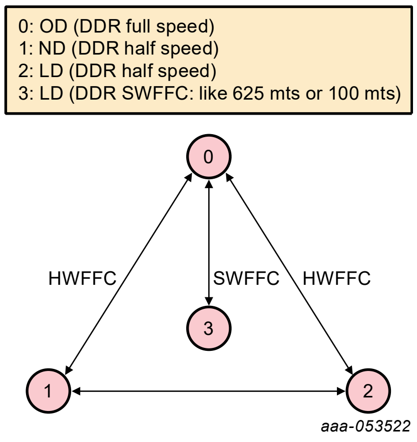

Figure 3. Mode control node

LD mode is only available when the i.MX 93 `ld.dtb` is used. For OD and ND modes, we can switch between the two modes freely even with the default DTB.

Valid parameters: 0 - 3

- 0: OD
- 1: ND
- 2: LD
- 3: LD (DDR swffc)

Example:

```
echo 1 > /sys/devices/platform/imx93-lpm/mode   // change to nd mode, ddr to
 half speed
echo 0 > /sys/devices/platform/imx93-lpm/mode   // back to od mode, ddr to
 full speed
echo 2 > /sys/devices/platform/imx93-lpm/mode   // change to ld mode, ddr to
 half speed
echo 3 > /sys/devices/platform/imx93-lpm/mode // change to ld mode ddr to
 lowest speed with SWFFC
```

Limitation:

High resolution display like 1080P 60fsp cannot be supported when doing systemwide mode switching. The display may occur flicker when changing the DDR frequency. Users should turn off the display or use a lower resolution.

High resolution display like 1080P 60fsp cannot be supported in LD mode. Therefore, to change to LD mode, the display should be off or with a lower resolution.

- `auto_clk_gating`

This control node is used to enable the DDRC auto clock gating to save power when there is no access to the DDR after the programed idle count expires. Writing ‘0’ will disable the auto clock gating, and writing a non-zero value will set `ssi_idle_strip` to this non-zero value and enable the auto clock gating. A value < 256 has some significant side effort for DDR performance, so it is recommended to set a value >= 256 to enable it.

Valid parameters: 0 or 256 to 2^16 -1

Example:

```
echo 0 > /sys/devices/platform/imx93-lpm/auto_clk_gating   // disable ddrc
 auto clock gating
echo 512 > /sys/devices/platform/imx93-lpm/auto_clk_gating   // enable ddrc
 auto clock gating
```

Limitation:

When the auto clock gating is enabled, a high resolution display like 1080P 60fps may flicker at a lower DDR frequency.

2.5.4.5 Menu configuration options

There are no menu configuration options for this driver. The Bus Frequency drivers are included and enabled by default for the SoC that support Bus Frequency drivers.

2.5.5 Battery Charging

2.5.5.1 Introduction

Battery Charging is supported by the max8903-charger for the i.MX 6 SABRE SD boards.

2.5.5.2 Software operation

None.

2.5.5.3 Source code structure

The battery charging source is based in `drivers/power/supply/sabresd_battery.c`.

2.5.5.4 Menu configuration options

In menu configuration, enable the following module:

Device Drivers -> **Power supply class support** -> **Sabresd Board Battery DC-DC Charger for USB and Adapter Power**.

2.6 OProfile

2.6.1 Introduction

OProfile is a systemwide profiler capable of profiling all running code at low overhead.

OProfile consists of a kernel driver, a daemon for collecting sample data, and several post-profiling tools for turning data into information.

2.6.1.1 Overview

OProfile uses 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.

2.6.1.2 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.
- Systemwide 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: 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 use the most CPU time across the whole system can be produced.
- System support: Works with any i.MX supported kernels.

2.6.1.3 Hardware operation

OProfile is a statistical continuous profiler.

Profiles are generated by regularly sampling the current registers on each CPU (from an interrupt handler, the saved PC value at the time of the 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 therefore 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).

By regularly sampling the PC value in this way, we can closely approximate what was actually executed and how frequently. In most cases, this statistical estimate is sufficiently accurate to represent 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 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.

2.6.1.4 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 `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()`, and `stop()`, which manage the hardware-specific details that 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.

2.6.1.5 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.

2.6.1.6 Generic kernel driver

The generic kernel driver resides in `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.

2.6.1.7 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 some 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 so on.

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).

2.6.1.8 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.

2.6.1.9 Interrupt requirements

The number of interrupts generated regarding the OProfile driver are numerous. The latency requirements are not needed.

The rate at which interrupts are generated depends on the event.

2.6.2 Software operation

For i.MX with Cortex-A7, OProfile requires adding the Cortex-A7 Event Monitor.

2.6.2.1 Source code structure

Oprofile platform-specific source files are available in `arch/arm/oprofile`.

Table 19. OProfile source files

File	Description
`common.c`	Source file with the implementation required for all platforms

The generic kernel driver for Oprofile is located under `drivers/oprofile`.

2.6.2.2 Menu configuration options

The following Linux kernel configurations are provided for this module.

In menu configuration, enable the following module:

- **CONFIG_OPROFILE**: configuration option for the OProfile driver. In `menuconfig`, this option is available under:
  General Setup -> **Profiling support (EXPERIMENTAL)** -> **OProfile system profiling (EXPERIMENTAL)**

2.6.2.3 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`.

2.6.2.4 Example software configuration

The following steps show an example of how to configure the OProfile:

1. Use the command `bitbake linux-imx -c menuconfig`. On the screen, go to Package list and select Oprofile.
2. Return to the first screen and select **Configure Kernel**. Follow the instruction from Section 2.6.2.2 to enable Oprofile in the kernel space.
3. Save the configuration and start to build.
4. Copy Oprofile binaries to the target rootfs. Copy `vmlinux` to the `/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.
```

2.7 Pulse-Width Modulator (PWM)

2.7.1 Introduction

The pulse-width modulator (PWM) has a 16-bit counter and is optimized to generate sound from the stored sample audio images and generate tones. The PWM also provides control for the backlight.

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.

2.7.2 Hardware operation

The figure below shows the PWM block diagram.

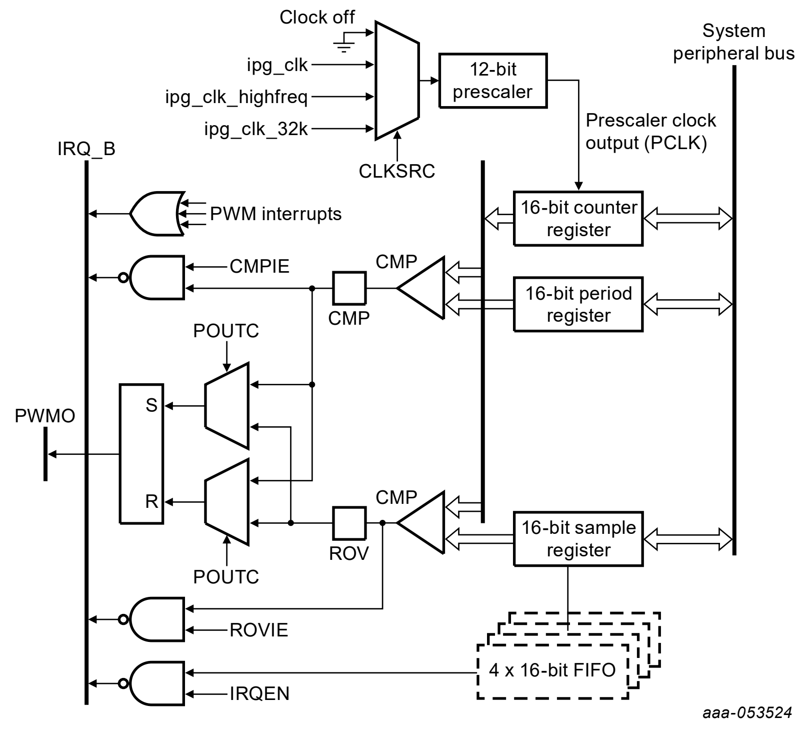

Figure 4. PWM block diagram

The PWM follows the 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 secondary bus signals

2.7.3 Clocks

The clock that feeds the prescaler can be selected from:

- High frequency clock: provided by the CCM. The PWM can be run on this clock in low-power mode.
- Low reference clock: 32 kHz low reference clock provided by the CCM. The PWM can be run on this clock in the low power mode.
- Global functional clock: for normal operations. In low-power mode, 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.

2.7.4 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.

The table below provides a summary of the interface functions in the source code.

Table 20. PWM driver summary

Function	Description
`struct pwm_device *pwm_get(struct device *dev, const char *con_id)`	Look up and request a PWM device.
`void pwm_put(struct pwm_device *pwm)`	Release 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, 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.

2.7.5 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 off

2.7.6 Source code structure

Table 21. PWM driver files

File	Description
`drivers/pwm/pwm.h`	Functions declaration
`drivers/pwm/pwm-imx.c`	i.MX Pulse Width modulation Functions

2.7.7 Menu configuration options

In menu configuration, enable the following module:

- **Device Drivers** -> **Pulse-Width Modulation (PWM) Support** -> **i.MX PWM support**
- Select the following option to enable the Backlight driver:
  Device Drivers -> **Graphics support** -> **Backlight & LCD device support** -> **Generic PWM based Backlight Driver**

2.8 Remote Processor Messaging

2.8.1 Introduction

With the newest multicore architecture designed by using the Arm Cortex-A series processors and the ArmCortex-M series processors, industrial applications can achieve greater power efficiency for a reduced carbon footprint. This reduces power consumption without performance deterioration.

A homogeneous SoC would traditionally run a single operating system (OS) that controls all the memory. The OS or a hypervisor would handle task management among available cores to maximize system utilization. Such a system is called Symmetric MultiProcessing (SMP).

A heterogeneous multicore chip is where different processing cores run different instruction sets and different OSs. Each processing core handles a specific task as required. Such a system is called Asymmetric Multiprocessing (AMP). To understand the distinction between the SMP and AMP systems, it is possible for a homogeneous multicore SoC to be an AMP system, but a heterogeneous multicore SoC cannot be an SMP system.

A multicore architecture brings new challenges to the system design, because the software must be rewritten to distribute tasks across the available cores. In addition, all the peripheral resources need to be properly allocated to avoid resource contention and achieve efficient sharing of the data spaces between the cores. A multicore SoC also needs mechanisms for reliable communication and synchronization among tasks running on different processing cores.

RPMsg is a virtio-based messaging bus, which allows kernel drivers to communicate with remote processors available on the system. In turn, drivers could then expose appropriate user space interfaces if needed. Every RPMsg device is a communication channel with a remote processor (so the RPMsg devices are called channels). Channels are identified by a textual name and have a local ("source") RPMsg address, and remote ("destination") RPMsg address. For more information, see www.kernel.org/doc/Documentation/rpmsg.txt.

As shown in the following figure, the messages pass between endpoints through bidirectional connection-less communication channels.

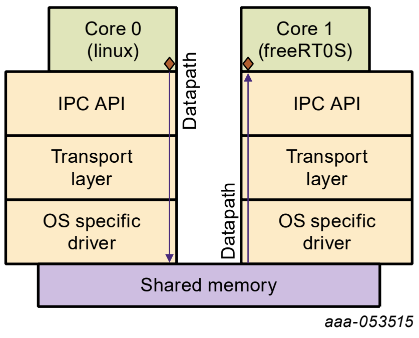

Figure 5. New multicore, multi-OS architecture

2.8.2 Features

- Designed for low-latency and low-overhead operation, and compliant with the Linux RPMsg framework.
- Optimized for embedded environments with constrained CPU and memory resources.
- Implementation by using shared memory without data translation or message headers.
- Application communication by using a client-server methodology.
- Dynamic allocation of the RPMsg channels.

2.8.3 Source code

The RPMSG driver software is in `drivers/rpmsg`.

Table 22. RPMSG source

File	Description
`drivers/rpmsg/virtio_rpmsg_bus.c`	Common code
`drivers/rpmsg/imx_rpmsg.c`	i.MX platform-related code
`drivers/rpmsg/imx_rpmsg_pingpong.c`	i.MX RPMsg ping-pong tests
`drivers/rpmsg/imx_rpmsg_tty.c`	i.MX RPMsg TTY driver

2.8.4 Menu configuration options

In menu configuration, enable the following module:

- **Device Drivers** -> **IMX RPMSG pingpong driver** -> **loadable modules only**
- **Device Drivers** -> **IMX RPMSG tty driver** -> **loadable modules only**

2.8.5 Running i.MX RPMsg test programs

To run the i.MX RPMsg test program, perform the following operations:

1. Make sure that the proper Cortex-M4 processor RTOS and Linux images are used.
   For example, on the i.MX 7Dual platforms:
   - `rpmsg_pingpong_sdk_7dsdb.bin` -> `ping-pong` test used on the i.MX 7Dual SDB board
   - `rpmsg_str_echo_sdk_7dsdb.bin` -> `tty string echo` test used on the i.MX 7Dual SDB board
   - `rpmsg_pingpong_sdk_7dval.bin` -> `ping-pong` test used on the i.MX 7Dual 12x12 LPDDR3 Arm2 board
   - `rpmsg_str_echo_sdk_7dval.bin` -> `tty string echo` test used on the i.MX 7Dual 12x12 LPDDR3 Arm2 board
2. Load the Cortex-M4 processor RTOS image, and kick it off in U-Boot.
   Load the Cortex-M4 processor RTOS image by the TFTP server or by the bootable SD card in U-Boot.
   - Load the Cortex-M4 processor RTOS image by the TFTP server. For example,
     a. Boot into U-Boot and stop.
     b. Use the following command to TFTP the responding Cortex-M4 processor RTOS image and boot it.

```
dhcp 0x7e0000 10.192.242.53:rpmsg_pingpong_sdk_7dval.bin;  bootaux
 0x7e0000
```

   - Load the Cortex-M4 processor RTOS image by the SD card. For example,
     a. Created A bootable SD card by the MFGtools. Then, copy the Cortex-M4 processor RTOS files to the first partition formatted by the VFAT file system.
     b. Change the default Cortex-M4 processor RTOS name of the U-Boot.

```
setenv m4image '<The name of the M4/RTOS image>';save
```

     c. Set up a bootargs used by the Cortex-M4 processor.

```
setenv run_m4_tcm 'if run loadm4image; then cp.b ${loadaddr} 0x7e0000
 0x8000; bootaux 0x7e0000; fi'; save
```

     d. Modify the original `bootcmd` by adding `run run_m4_tcm`.

```
setenv bootcmd "run run_m4_tcm; <original contents of the bootcmd>"; save
```

Note:
`uart_from_osc` is mandatory required by i.MX 6SoloX when the Cortex-M4 processor RTOS image is running. Therefore, the `mmcargs` of U-Boot should be modified on i.MX 6SoloX.

```
setenv mmcargs 'setenv bootargs console=${console},${baudrate} root=
${mmcroot}, uart_from_osc';save
```

3. Run the RPMsg test program.
   a. Make sure that `imx_rpmsg_pingpong.ko` and `imx_rpmsg_tty.ko` are built out.
   b. Use `insmod imx_rpmsg_pingpong.ko` or `insmod imx_rpmsg_tty.ko` to run the test program.
   Note: *Do not run different test programs at the same time.*
   c. Run the following command and ensure that the RPMsg TTY receiving program is running at the backend when starting RPMsg TTY tests.

   ```
   /unit_tests/mxc_mcc_tty_test.out /dev/ttyRPMSG30 115200 R 100 1000 &
   ```

   Logs at the Linux OS side:

   ```
   insmod imx_rpmsg_tty.ko
   imx_rpmsg_tty rpmsg0: new channel: 0x400 -> 0x1!
   Install rpmsg tty driver!
   echo deadbeaf > /dev/ttyRPMSG30
   imx_rpmsg_tty rpmsg0: msg(<- src 0x1) deadbeaf len 8
   ```

2.9 Thermal

2.9.1 Introduction

Thermal driver is a necessary driver for monitoring and protecting the SoC. The thermal driver monitors the SoC temperature in a certain frequency from an internal thermal sensor.

It defines two trip points: critical and passive. The cooling device takes actions to protect the SoC according to the different trip points that the SoC has reached:

- When reaching the critical point, the cooling device shuts down the system.
- When reaching the passive point, the cooling device lowers the CPU frequency and notifies the GPU/VPU to run at a lower frequency.
- When the temperature drops to 10°C below the passive point, the cooling device releases all the cooling actions.

The thermal driver has two parts:

- The thermal zone defines trip points and monitors the SoC's temperature.
- The cooling device takes the actions according to different trip points.

The critical and passive points thresholds are configured in the following files.

- i.MX 6 and i.MX 7 SoCs configure this in `drivers/thermal/imx_thermal.c`.
- i.MX 8M SoCs configure this in their DTSI file and specify **CONFIG_IMX8M_THERMAL** in `defconfig`.
- i.MX 8 and i.MX 8X SoCsconfigure this in their DTSI file and specify **CONFIG_IMX_SC_THERMAL** in `defconfig`.

2.9.2 Software operation

The thermal driver registers a thermal zone and a cooling device. A structure, `thermal_zone_device_ops`, describes the necessary interface that the thermal framework needs. The framework calls the related thermal zone interface to monitor the SoC temperature and performs the cooling protection.

The thermal driver can be accessed through the following interface:

- `/sys/bus/platform/drivers/imx_thermal` for i.MX 6 and i.MX 7.
- `/sys/class/thermal/thermal_zoneX` for i.MX 8 and i.MX 8X.
- `/sys/bus/platform/drivers/qoriq_thermal` for i.MX 8M Quad.
- `/sys/class/thermal/thermal_zone0/temp` for i.MX 8M Mini.

2.9.3 Source code structure

The table below shows the driver source files available in `drivers/thermal`.

Table 23. Thermal driver files

File	Description
`imx_thermal.c`, `device_cooling.c`	Thermal zone driver source file for i.MX 6 or i.MX 7.
`qoriq_thermal.c`, `device_cooling.c`	Thermal zone driver source files for i.MX 8M and i.MX 93.
`imx_sc_thermal.c`, `device_cooling.c`	Thermal zone driver source files for i.MX 8 and i.MX 8X.
`imx91_thermal.c`, `device_cooling.c`	Thermal zone driver source files for i.MX 91.
`scmi-hwmon.c`, `device_cooling.c`	Thermal zone driver source files for i.MX 8ULP, i.MX 95, and i.MX 943.

2.9.4 Menu configuration options

In menu configuration, enable the following module:

- For i.MX6 and i.MX7: **Device Drivers** -> **Generic Thermal sysfs driver** -> **Temperature sensor driver for i.MX SoCs**.
- For i.MX 8QuadMax and i.MX 8QuadXPlus: **Device Drivers** -> **Generic Thermal sysfs driver** -> **Thermal sensor driver for NXP i.MX8 SoCs**

2.10 Sensors

2.10.1 Introduction

Sensors include a group of drivers for Accelerometer, Pressure, Gyroscope, Ambient Light, and Magnetometer.

Sensors are configured in the device trees for each board.

i.MX supports accelerometers for the following SoCs:

- i.MX 6SABRE-SD, 6SABRE-AI, and 6SoloX use the MMA8451 sensor.
- i.MX 6UltraLite and 6ULL EVK use the FXLS8571Q sensor.
- i.MX 7Dual SABRE-SD, i.MX 8QuadMax, and i.MX 8QuadXPlus use the FX0S8700 sensor.

i.MX Supports pressure sensor MPL3115 for the following SoCs:

- i.MX 7Dual SABRE-SD
- i.MX 8QuadMax
- i.MX 8QuadXPlus

i.MX Supports gyroscope sensor FXAS2100 for the following SoC:

- i.MX 7Dual SABRE-SD

i.MX Supports ambient light sensor ISL29023 for the following SoCs:

- i.MX 6 SABRE-SD and 6 SABRE-AI

- i.MX 6SoloX
- i.MX 8QuadMax
- i.MX 8QuadXPlus

i.MX supports magnetometer sensors MAG3110 for the following SoCs:

- i.MX 6 SABRE-SD
- i.MX 6UL EVK
- i.MX 6ULL EVK
- i.MX 6SoloX

i.MX supports accelerometer and gyroscope sensor LSM6DSO for the following SoC:

- i.MX 93

i.MX supports temperature sensor P3T1085 for the following SoCs:

- i.MX 93
- i.MX 95
- i.MX 943

2.10.2 Sensor driver software operation

None.

2.10.3 Source code structure

The table below lists the driver source files available in the directory.

Table 24. Sensor driver files

File	Description
`drivers/iio/accel/mma8452.c`	Acceleromater sensor
`drivers/iio/imu/fxos8700_i2c.c` `drivers/iio/imu/fxos8700_core.c`	Accelaromater and Magnetometer sensor
`drivers/iio/light/isl29018.c`	Ambient Light sensor
`drivers/iio/gyro/fxas21002c_i2c.c` `drivers/iio/gyro/fxas21002c_core.c`	Gyroscope sensor
`drivers/iio/magnetometer/mag3110.c`	Magnetometer sensor
`drivers/iio/imu/st_lsm6dsx/st_lsm6dsx_i2c.c` `drivers/iio/imu/st_lsm6dsx/st_lsm6dsx_core.c`	Accelerometer and gyroscope sensor

`drivers/iio/temperature/p3t/p3t1085_i2c.c`
`drivers/iio/temperature/p3t/p3t1085_core.c`
`drivers/iio/temperature/p3t/p3t1085_i3c.c`
	Temperature sensor

2.10.4 Menu configuration options

In menu configuration, enable the following modules:

- **Drivers** -> **Industrial I/O** -> **Accelerometers** -> **Freescale/NXP MMA8452Q**
- **Drivers** -> **Industrial I/O** -> **Inertial measurement units** -> **NXP FXOS8700 I2C driver**
- **Drivers** -> **Industrial I/O** -> **Light sensors** -> **Intersil 29018 light and proximity sensor**
- **Drivers** -> **Industrial I/O** -> **Digital gyroscope sensors** -> **NXP FXAS21002C Gyro Sensor**
- **Drivers** -> **Industrial I/O** -> **Freescale MAG3110 3-Axis Magnetometer**
- **Drivers** -> **Industrial I/O** -> **Inertial measurement units** -> **ST_LSM6DSx driver for STM 6-axis IMU MEMS sensors**
- **Drivers** -> **Industrial I/O support** -> **Temperature sensors** -> **NXP P3T1085 temprature sensor**

2.11 Watchdog (WDOG)

2.11.1 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. i.MX 6 and 7Dual share the same watch dog driver with i.MX 8M. i.MX 7ULP has a separate watchdog driver. i.MX 8 and i.MX 8X share a virtual watchdog driver interface through the system controller firmware.

2.11.2 Hardware operation

After the WDOG timer is activated, it must be serviced by the 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 after it is activated.

2.11.3 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 WDOG registers are only one-time programmable after booting, ensure these registers are written correctly.

2.11.4 Generic WDOG

The generic WGOD driver is implemented in the `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.

2.11.5 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.

2.11.6 Source code structure

The table below shows the source files for watchdog WDOG drivers that are in `drivers/watchdog`.

Table 25. Watchdog driver files

File	Description
`driveers/watchdog/imx2_wdt.c`	i.MX 6, i.MX 7Dual, and i.MX 8M watchdog function implementations. For i.MX 6 and i.MX 7, the watchdog system reset function is located under `arch/arm/mach-imx/system.c`.
`drivers/watchdog/imx7ulp_wdt.c`	i.MX 7ULP, i.MX 8ULP, and i.MX 9 watchdog function implementations
`drivers/watchdog/imx8_wdt.c`	On i.MX 8 and i.MX 8X, the software watchdog used in system controller firmware (SCFW) and kernel call those interfaces by virtual watchdog driver `imx8_wdt.c`. This is not used for i.MX 8M.

2.11.7 Menu configuration options

In menu configuration, enable the following module:

Device Drivers -> **Watchdog Timer Support** -> **IMX2+ Watchdog**

Device Drivers -> **Watchdog Timer Support** -> **IMX7ULP Watchdog**

Device Drivers -> **Watchdog Timer Support** -> **IMX8 Watchdog**

2.11.8 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 `Documentation/watchdog`.

3 Storage

3.1 AHB-to-APBH Bridge with DMA (APBH-Bridge-DMA)

3.1.1 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 denotes 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, as well as 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 their own chapters elsewhere in this document.

There is no separate DMA bus for these devices. Contention between the DMA's use of the APBH bus and the AHB-to-APB bridge functions' use of the APBH is mediated by an internal arbitration logic. For contention between these two units, the DMA is favored and the secondary AHB reports "not ready" through its HREADY output until the bridge transfer can complete. The arbiter tracks repeated lockouts and inverts the priority, guaranteeing the Arm platform every fourth transfer on the APB.

3.1.1.1 Hardware operation

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

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

3.1.1.2 Software operation

The DMA supports sixteen 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 using the general structure.

Table 26. APBH DMA channel sssignments

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

3.1.1.3 Source code structure

The table below lists the source files available in `drivers/dma/`.

Table 27. APBH DMA source files

File	Description
`mxs-dma.c`	APBH DMA implement driver

3.1.1.4 Menu configuration options

In menu configuration, enable the following module:

- **Device Drivers** -> D**MA Engine support** -> **MXS DMA support**.

3.1.1.5 Programming interface

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

3.2 EIM NOR

3.2.1 Introduction

The External Interface Module (EIM) NOR driver supports the Parallel NOR flash.

3.2.2 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.

3.2.3 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.

3.2.4 Source code

Table 28. WEIM-NOR driver files

File	Description
`drivers/bus/imx-weim.c`	Timing only changes for Parallel NOR WEIM-NOR source

3.2.5 Enabling the EIM NOR

Refer to the DTS file to enable the EIM NOR: `imx6q-sabreauto-gpmi-weim.dts` or `imx6dl-sabreauto-gpmi-weim.dts`.

3.3 MMC/SD/SDIO Host

3.3.1 Introduction

The MultiMediaCard (MMC), Secure Digital (SD), or 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, and AR6003 is verified).
- Supports card insertion and removal detections.
- Supports the standard MMC commands.
- PIO and DMA data transfers.
- Supports power management.
- Supports 1/4 8-bit operations for MMC cards.
- For i.MX 6, USDHC supports eMMC4.4 SDR and DDR modes.
- For i.MX 7Dual, USDHC supports eMMC5.0, which includes HS400 and HS200.
- Supports SD3.0, SDR50, and SDR104 modes.

3.3.2 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, the 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 can generate interrupts and set the corresponding Status Register bits. The uSDHC interrupt status enable and signal-enable registers allow the user to control if these interrupts occur.

3.3.3 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:

For uSDHC:

- The `init` function `esdhc_pltfm_init()` initializes the platform hardware and set platform dependent 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.
- `esdhc_execute_tuning()` handles the preparation for tuning. It is only used for SD3.0 UHS-I mode.
- `esdhc_set_clock()` handles the clock change request.

The figure below shows how the MMC-related drivers are layered.

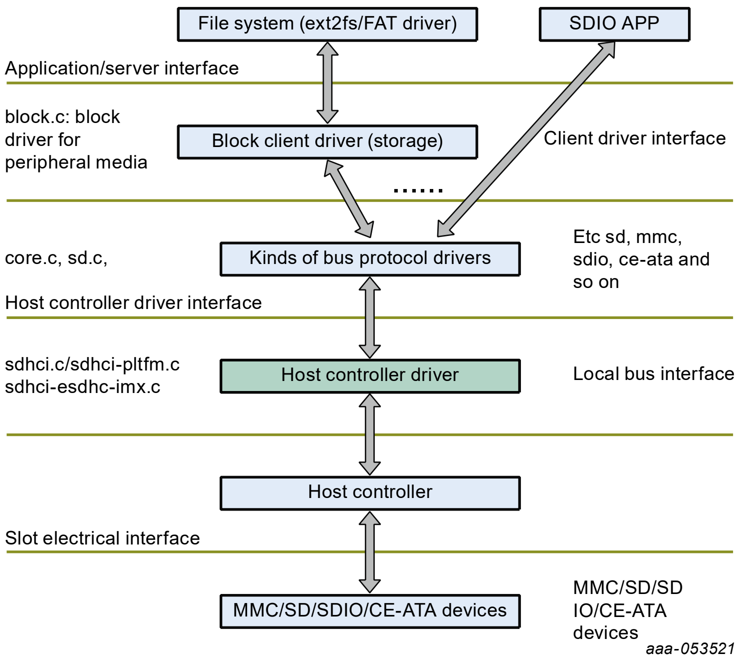
File system (ext2fs/FAT driver)
SDIO APP
Application/server interface
block.c: block driver for peripheral media
Block client driver (storage)
Client driver interface
......
core.c, sd.c,
Kinds of bus protocol drivers
Etc sd, mmc, sdio, ce-ata and so on
Host controller driver interface
sdhci.c/sdhci-pltfm.c sdhci-esdhc-imx.c
Host controller driver
Local bus interface
Host controller
Slot electrical interface
MMC/SD/SDIO/CE-ATA devices
MMC/SD/SD IO/CE-ATA devices
aaa-053521

Figure 6. MMC drivers layering

3.3.4 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 a loadable or built-in module.

3.3.5 Source code structure

The table below shows the uSDHC source files available in `drivers/mmc/host`.

Table 29. uSDHC Driver Files MMC/SD driver files

File	Description
`drivers/mmc/host/sdhci.c`	SDHCI standard stack code
`driers/mmc/host/sdhci-pltfm.c`	SDHCI platform layer
`drivers/mmc/host/sdhci-esdhc-imx.c`	uSDHC driver
`drivers/mmc/host/sdhci-esdhc.h`	uSDHC driver header file

3.3.6 Menu configuration options

The following Linux kernel configuration options are provided for 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 the 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 eSDHC i.MX controller**

  To compile the SDHCI driver as a loadable module, several options should be selected as indicated below:
  - `CONFIG_MMC_SDHCI=m`, which can be found at **Device Drivers** -> **MMC/SD Card Support** -> **Secure Digital Host Controller Interface support**.
  - `CONFIG_MMC_SDHCI_PLTFM=m`, which 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`, which 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 built-in module, several options should be selected as indicated below:
  - `CONFIG_MMC_SDHCI=y`, which can be found at **Device Drivers** -> **MMC/SD Card Support** -> **Secure Digital Host Controller Interface support**.
  - `CONFIG_MMC_SDHCI_PLTFM=y`, which 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`, which 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.

3.3.7 Device tree binding

Required properties:

- `compatible`: Should be `fsl,<chip>-esdhc`.

- `reg`: Should contain an eSDHC registers location.
- `interrupts`: Should contain eSDHC interrupt.

Optional properties:

- `non-removable`: Indicate that the card is wired to the host permanently.
- `fsl`, `wp-internal`: Indicate to use controller internal write protection.
- `cd-gpios`: Specify GPIOs for card detection.
- `wp-gpios`: Specify GPIOs for write protection.
- `fsl`, `delay-line`: Specify delay line value for eMMC DDR mode.

```
Example:usdhc@02194000 { /* uSDHC2 */
compatible = "fsl,imx6q-usdhc";
reg = <0x02194000 0x4000>;
interrupts = <0 23 0x04>;
clocks = <&clks 164>, <&clks 164>, <&clks 164>;
clock-names = "ipg", "ahb", "per";
pinctrl-names = "default";
pinctrl-0 = <&pinctrl_usdhc2_1>;
cd-gpios = <&gpio2 2 0>;
wp-gpios = <&gpio2 3 0>;
bus-width = <8>;
no-1-8-v;
keep-power-in-suspend;
enable-sdio-wakeup;
status = "okay";
};
```

Reference:

- `Documentation/devicetree/bindings/mmc/fsl-imx-esdhc.txt`
- `arch/arm/boot/dts/imx6*.dtsi`

3.3.8 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`.

3.3.9 Loadable module operations

The SDHCI driver can be built as a loadable or built-in module.

1. How to build the SDHCI driver as a loadable module.
   - `CONFIG_MMC_SDHCI=m`, which can be found at **Device Drivers** -> **MMC/SD Card Support** -> **Secure Digital Host Controller Interface support**.
   - `CONFIG_MMC_SDHCI_PLTFM=m`, which 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`, which 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 i.MX eSDHC i.MX controller**.
2. How to load and unload the SDHCI module.
   Due to dependency, load or unload the module following the module sequence as follows.

Run the following commands to load the module:

- Load the modules by the `insmod` command, assuming the files of `sdhci.ko` and `sdhci-platform.ko` exist in the current directory.

```
$> insmod sdhci.ko
$> insmod sdhci-platform.ko
```

- Load the modules by the `modprobe` command. Make sure that the files of `sdhci.ko` and `sdhci-platform.ko` exist in the library directory of the corresponding kernel module.

```
$> modprobe sdhci.ko
$> modprobe sdhci-platform.ko
```

Run the following commands to unload the module:

- Unload the modules by the `insmod` command:

```
$> rmsmod sdhci-platform
$> rmsmod sdhci
```

- Unload the modules by the `modprobe` command:

```
$> modprobe -r sdhci-platform
$> modprobe -r sdhci
```

3.4 NAND GPMI Flash

3.4.1 Introduction

The NAND Flash Memory Technology Devices (MTD) driver is used in the Generic-Purpose Media Interface (GPMI) controller on the i.MX 6 series and i.MX 7Dual. 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.

The NAND MTD driver interfaces with the integrated NAND controller supporting file systems, such as UBIFS, CRAMFS, JFFS2UBI, UBIFSCRAMFS, and JFFS2. The driver implementation supports the lowest level operations on the external NAND Flash chip, such as block read, block write, and block erase as the NAND Flash technology only supports block access. Because blocks in a NAND Flash are not guaranteed to be good, the NAND MTD driver is also able to detect bad blocks and feed that information to the upper layer to handle bad block management.

3.4.2 Hardware operation

NAND Flash is a nonvolatile storage device used for embedded systems.

The driver 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 cannot be executed from there. Code must be loaded into RAM memory and executed from there. The i.MX 6 contains a hardware error-correcting block.

3.4.3 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 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 the i.MX 6. 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 www.linux-mtd.infradead.org.

3.4.4 Basic operations: Read/Write

The NAND driver exports the following callbacks:

```
gpmi_ecc_read_page (with ECC)
gpmi_ecc_write_page (with ECC)
gpmi_read_byte (without ECC)
gpmi_read_buf (without ECC)
gpmi_write_buf (without ECC)
gpmi_ecc_read_oob (with ECC)
gpmi_ecc_write_oob (with ECC)
```

Since Kernel 4.1, the GPMI driver provides raw read/write modes, which exports these callbacks:

- `gpmi_ecc_read_page_raw` (without ECC)
- `gpmi_ecc_write_page_raw` (without ECC)
- `gpmi_ecc_read_oob_raw` (without ECC)
- `gpmi_ecc_write_oob_raw` (without ECC)

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

3.4.5 Error correction

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

For BCH, the page layout of 2K page is (2K + 64), the page layout of 4K page is (4K + 218), and the page layout of 8K page is (8K + 448).

3.4.6 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`.

3.4.7 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.

3.4.8 Source code structure

The NAND driver is located in `drivers/mtd/nand/gpmi-nand`.

The table below lists the source files for the NAND Driver.

Table 30. NAND driver files

File	Description
• `drivers/mtd/nand/gpmi-nand/bch-regs.h` • `drivers/mtd/nand/gpmi-nand/gpmi-nand.h` • `drivers/mtd/nand/gpmi-nand/gpmi-regs.h`	Functions declaration
• `drivers/mtd/nand/gpmi-nand/gpmi-lib.c` • `drivers/mtd/nand/gpmi/nand/gpmi-nand.c`	GPMI NAND Functions

3.4.9 Menu configuration options

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

- **Device Drivers** -> **Memory Technology Device (MTD) support** -> **GPMI NAND Flash Controller driver**

These MTD options must be enabled:

- `CONFIG_MTD_NAND = [y | m]`
- `CONFIG_MTD = y`
- `CONFIG_MTD_BLOCK = y`

These UBI options must be enabled:

- `CONFIG_MTD_UBI=y`
- `CONFIG_UBIFS_FS=y`

3.5 Quad/Flexible/External Serial Peripheral Interface (QuardSPI/FlexSPI/XSPI)

3.5.1 Introduction

The Quad Serial Peripheral Interface (QuadSPI) block acts as an interface to one single or two external serial flash devices, each with up to four bidirectional data lines. The Flexible Serial Peripheral Interface (FlexSPI) host controller supports two SPI channels and up to 4 external devices. Each channel supports Single/Dual/Quad/Octal mode data transfer (1/2/4/8 bidirectional data lines). XSPI is a memory controller IP, designed to support serial flash or RAM memories. The controller includes a programmable sequence engine providing flexibility to support existing and future memory devices. XSPI supports single, dual, quad, and octal modes of operation.

It supports the following features:

- Flexible sequence engine to support various flash vendor devices.
- Single, dual, quad, and octal modes of operation.
- DDR/DTR mode wherein the data is generated on every edge of the serial flash clock.
- Support for flash data strobe signal for data sampling in DDR and SDR mode.
- DMA support to read RX Buffer data via AMBA AHB bus (64-bit width interface) or IP registers space (32-bit access).

3.5.2 Hardware operation

On some boards, the Quad SPI NOR - N25Q256A is equipped, while on some other boards, S25FL128S is equipped. Check the Quad SPI NOR type on the boards, and then configure it properly.

The N25Q256A is a high-performance multiple input/output serial Flash memory device. The innovative, high-performance, dual and quad input/output instructions enable double or quadruple the transfer bandwidth for READ and PROGRAM operations. The memory is organized as 512 (64 kB) main sectors and can be erased 64 kB sectors at a time. The device features 3-byte or 4-byte address modes to access memory beyond 128 MB. When 4-byte address mode is enabled, all commands requiring an address must be entered and exited with a 4-byte address mode command: `ENTER 4-BYTE ADDRESS MODE` command and `EXIT 4-BYTE ADDRESS MODE` command. The 4-byte address mode can also be enabled through the nonvolatile configuration register.

The memory can be operated with three different protocols: Extended SPI (standard SPI protocol upgraded with dual and quad operations), Dual I/O SPI, and Quad I/O SPI. Each protocol contains unique commands to perform READ operations in DTR mode. This enables high data throughput while running at lower clock frequencies.

The S25FL128S device is a flash non-volatile memory product. It connects to a host system through a Serial Peripheral Interface (SPI). Traditional SPI single bit serial input and output (Single I/O or SIO) is supported as well as optional two bit (Dual I/O or DIO) and four bit (Quad I/O or QIO) serial commands. It also adds support for Double Data Rate (DDR) read commands for SIO, DIO, and QIO that transfer address and read data on both edges of the clock.

3.5.3 Software operation

In a Flash-based embedded Linux system, a number of Linux technologies work together to implement a file system. The following figure shows the relationships between some of the standard components.

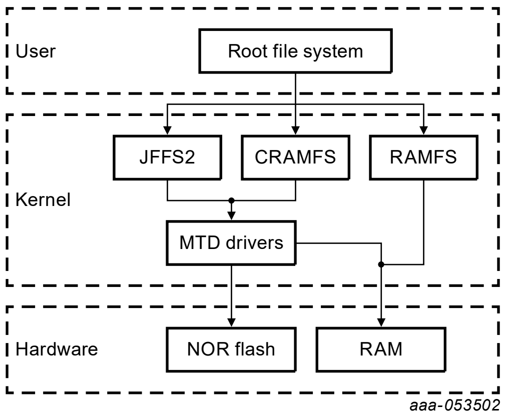

Figure 7. Components of a Flash-based file system

The MTD subsystem for Linux OS 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 the NOR Flash.

3.5.4 Driver features

This NOR driver implementation supports the following feature:

- Provides necessary information for the upper-layer MTD driver.

3.5.5 Source code structure

Table 31. Driver files

File	Description
`drivers/mtd/spi-nor/core.c`	SPI-NOR framework
`drivers/spi/spi-fsl-qspi.c`	Quad SPI Driver
`drivers/spi/spi-nxp-fspi.c`	FlexSPI Driver
`drivers/spi/spi-nxp-xspi.c`	XSPI Driver

3.5.6 Menu configuration options

To enable the SPI-NOR driver, set the follwoing options:

Device Drivers -> **Memory Technology Device (MTD) support** -> **SPI NOR device support**

- Select QuadSPI:
  Device Drivers -> **SPI support** -> **Freescale QSPI controller**
- Select FlexSPI:
  Device Drivers -> **SPI support** -> **NXP Flex SPI controller**
- Select XSPI:
  Device Drivers -> **SPI support** -> **NXP xSPI controller**

3.6 SATA

3.6.1 Introduction

The SATA AHCI driver is based on the LIBATA layer of the block device infrastructure of the Linux kernel. For the detailed hardware operation of SATA, see the Synopsys DesignWare Cores SATA AHCI documentation `SATA_Data_Book.pdf`.

3.6.2 Board configuration options

With the power off, install the SATA cable and hard drive.

3.6.3 Software operation

For the details about the libATA APIs, see the libATA Developer's Guide.

The SATA AHCI driver is based on the libATA layer of the block device infrastructure of the Linux kernel. i.MX integrated AHCI Linux driver combined the standard AHCI drivers handle the details of the integrated i.MX SATA AHCI controller, while the libATA layer understands and executes the SATA protocols. The SATA device, 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 the integrated DMA engine, which assists the SATA controller hardware in the DMA transfer modes.

3.6.4 Source code structure

The source code of the i.MX AHCI SATA driver is located in `drivers/ata`. The standard AHCI and AHCI platform drivers are used to perform the actual SATA operations.

Table 32. SATA driver files

File	Description
`drivers/ata/ahci_imx.c`	i.MX AHCI SATA Driver
`drivers/ata/ahci.c`	Standard AHCI drivers
`drivers/ata/ahci-platform.c`	Standard AHCI platform drivers

3.6.5 Menu configuration options

The following Linux kernel configurations are provided for the SATA driver:

```
Symbol: AHCI_IMX [=y]
Type  : tristate
Prompt: Freescale i.MX AHCI SATA support
  Location:
    -> Device Drivers
      -> Serial ATA and Parallel ATA drivers (ATA [=y])
        -> Platform AHCI SATA support (SATA_AHCI_PLATFORM [=y])
```

In busybox, enable `fdisk` under **Linux System Utilities**.

3.6.6 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*`.

3.6.7 Usage example

Note: *There may be a known error message when a few kinds of SATA disks are initialized, such as:*

```
ata1.00: serial number mismatch '090311PB0300QKG3TB1A' != ''
ata1.00: revalidation failed (errno=-19)
```

This should be ignored.

1. After building the kernel and the SATA AHCI driver, and deploying, boot the target, and then log in as `root`.
2. Make sure that the AHCI and AHCI platform drivers are built in the kernel or loaded into the kernel.

You should see messages similar to the following:

```
ahci: SSS flag set, parallel bus scan disabled
ahci ahci: AHCI 0001.0300 32 slots 1 ports 3 Gbps 0x1 impl platform mode
ahci ahci: flags: ncq sntf stag pm led clo only pmp pio slum part ccc apst
scsi0 : ahci_platform
ata1: SATA max UDMA/133 mmio [mem 0x02200000-0x02203fff] port 0x100 irq 71
ata1: SATA link up 3.0 Gbps (SStatus 123 SControl 300)
ata1.00: ATA-8: SAMSUNG HM100UI, 2AM10001, max UDMA/133
ata1.00: 1953525168 sectors, multi 0: LBA48 NCQ (depth 31/32)
ata1.00: configured for UDMA/133
scsi 0:0:0:0: Direct-Access     ATA      SAMSUNG HM100UI  2AM1 PQ: 0 ANSI: 5
sd 0:0:0:0: [sda] 1953525168 512-byte logical blocks: (1.00 TB/931 GiB)
sd 0:0:0:0: [sda] 4096-byte physical blocks
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
```

```
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 appear under `/dev/sda*`. For example, to check the basic kernel settings for the drive, execute `hdparm /dev/sda`.

3.6.8 Usage example

Creating partitions:

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 reading/writing blocks. This command can corrupt the partitions and filesystem on the 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`.

```
$dd if=file_name of=/dev/sda1
```

To read 1 kB of data from the card, enter the following text, substituting the name of the file to be written for `output_file`.

```
$dd if=/dev/sda1 of=output_file bs=1024 count=1
```

Files system tests:

Format the hard disk partitions 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
```

3.7 Smart Direct Memory Access (SDMA) API

3.7.1 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 the 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.

3.7.2 Hardware operation

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

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

3.7.3 Software operation

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

The table below lists the drivers that use SDMA and the number of SDMA physical channels used by each driver. A driver can specify the SDMA channel number to use, static channel allocation, or can have the SDMA driver provide a free SDMA channel for the driver to use, dynamic channel allocation. For dynamic channel allocation, the list of SDMA channels is scanned from channel 32 to channel 1. When a free channel is found, that channel is allocated for the requested DMA transfers.

Table 33. 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

Table 33. SDMA channel usage*...continued*

Driver name	Number of SDMA channels	SDMA channel used
SPDIF	2 per device	Dynamic channel allocation
ESAI	2 per device	Dynamic channel allocation
ASRC	6 per device	Dynamic channel allocation
AUD2HTX	1 per device	Dynamic channel allocation
EASRC	8 per device	Dynamic channel allocation
ECSPI	2 per device	Dynamic channel allocation
MICFIL	1 per device	Dynamic channel allocation
XCVR	2 per device	Dynamic channel allocation

Note:

This table contains the functions currently supported by SDMA scripts, but the specific implementation may be different in each platform. The peripherals supported by SDMA are subject to the DTS configuration of each platform.

3.7.4 Source code structure

The `dmaengine.h` (header file for SDMA API) is available in the directory `linux/include/linux`.

The following table lists the source files available in the directory `drivers/dma`.

Table 34. SDMA API source files

File	Description
`drivers/dma/dmaengine.c`	SDMA management routine
`drivers/dma/imx-sdma.c`	SDMA implement driver
`drivers/dma/imx-dma.c`	i.MX DMA driver

The following table lists the image files available in the directory `firmware/imx/sdma` for 4.1 and 4.9 kernels. For 4.14 kernel, the SDMA firmware is provided with the `firmware-imx` package and not in the kernel source tree.

Table 35. SDMA script files

File	Description
`sdma-imx6q.bin`	SDMA RAM scripts for i.MX 6
`sdma-imx7d.bin`	SDMA RAM scripts for i.MX 7 and i.MX 8M

3.7.5 Special peripheral with SDMA cases

3.7.5.1 I2C in i.MX 6, i.MX 7Dual, and i.MX 8M

In the current release, the I2C controller and SDMA script in i.MX 6, i.MX 7Dual, and i.MX 8M do not support SDMA.

There are two limitations:

- I2C uses DMA mode when the frame length is greater than 16 bytes, because I2C itself still needs to use the CPU to process the first few and last few bytes when sending and receiving a frame. Therefore, when the data being sent is not long, using DMA to send data does not improve efficiency.
- The SDMA script is loaded in the rootfs stage, so any use of I2C DMA transfer in kernel boot stage will fail. It is strongly recommended not to use I2C SDMA mode when sending small amounts of data. If there is a special case that needs to send a large amount of I2C data, contact NXP Pro-support to get the patches.

3.8 SPI NOR Flash Memory Technology Device (MTD)

3.8.1 Introduction

The SPI NOR Flash Memory Technology Device (MTD) driver provides the support to the data Flash through the SPI interface. By default, the SPI NOR Flash MTD driver creates static MTD partitions to support data Flash.

3.8.2 Hardware operation

The SPI NOR - AT45DB321D is equipped on some boards, while the M25P32 is equipped on some others. 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 75 MHz. The memory is organized as 64 sectors, each containing 256 pages. Each page is 256 bytes wide. Therefore, the whole memory can be viewed as consisting of 16,384 pages, or 4,194,304 bytes. The memory can be programmed 1 byte 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.

3.8.3 Software operation

In a Flash-based embedded Linux system, a number of Linux technologies work together to implement a file system. The figure below shows the relationships between some of the standard components.

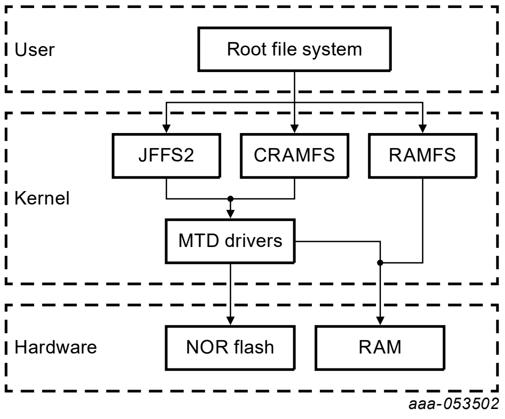

Figure 8. Components of a Flash-based file system

The MTD subsystem for Linux OS 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 the NOR Flash.

3.8.4 Source code structure

The following table lists the driver files.

Table 36. SPI NOR MTD driver files

File	Description
`drivers/mtd/devices/m25p80.c`	Source file
`drivers/mtd/spi-nor/core.c`	Source file

3.8.5 Menu configuration options

In menu configuration, enable the following module:

- **CONFIG_MTD_M25P80**: This configuration enables access to the 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, and so on)**

4 Connectivity

4.1 ADC

4.1.1 ADC introduction

The features of the ADC-Digital are as follows:

- Two 12-bit ADCs
- Linear successive approximation algorithm with up to 12-bit resolution with 10/11 bit accuracy
- Up to 1 MS/s sampling rate
- Up to 8 single-ended external analog inputs

- Single or continuous conversion (automatic return to idle after single conversion)
- Output Modes: (in right-justified unsigned format)
  - 12-bit
  - 10-bit
  - 8-bit
- Configurable sample time and conversion speed/power
- Conversion complete and hardware average complete flag and interrupt
- Input clock selectable from up to four sources
- Asynchronous clock source for lower noise operation with option to output the clock
- Selectable asynchronous hardware conversion trigger with hardware channel select
- Selectable voltage reference, Internal, External, or Alternate
- Operation in low power modes for lower noise operation
- Hardware average function
- Self-calibration mode

4.1.2 ADC external signals

- `ADC_VREFH`: Voltage reference high
- `ADC_VREHL`: Voltage reference low
- `ADC1_IN0`: Analog channel 1 input 0
- `ADC1_IN1`: Analog channel 1 input 1
- `ADC1_IN2`: Analog channel 1 input 2
- `ADC1_IN3`: Analog channel 1 input 3
- `ADC2_IN0`: Analog channel 2 input 0
- `ADC2_IN1`: Analog channel 2 input 1
- `ADC2_IN2`: Analog channel 2 input 2
- `ADC2_IN3`: Analog channel 2 input 3

The ADC pin settings should be done in the `ADCx_PCTL` register. No other extra IOMUX settings are required.

4.1.3 ADC driver overview

The ADC driver is developed under the Linux IIO (Industrial I/O) driver frame. The ADC driver only provides the basic functions. The following features are supported:

- Four external inputs for each ADC controller channel
- 12-bit ADC
- Single conversion
- Hardware average
- Low power mode of ADC
- Sample rate changes in the available sample rate group

4.1.4 Source code structure

Table 37. ADC driver files

File	Description
`drivers/iio/adc/vf610_adc.c`	i.MX 6UltraLite and i.MX 6SoloX ADC functions.
`drivers/iio/adc/imx7d_adc.c`	i.MX 7Dual ADC functions.
`drivers/iio/adc/imx8qxp_adc.c`	i.MX 8QuadXPlus ADC functions.

Table 37. ADC driver files*...continued*

File	Description
`drivers/iio/adc/imx93_adc.c`	i.MX 93 ADC functions.

4.1.5 Menu configuration options

Configure the kernel option to enable the module by `menuconfig`:

- **Device Drivers** -> **Industrial I/O support** -> **Analog to digital converters** -> **Freescale vf610 ADC driver**
- **Device Drivers** -> **Industrial I/O support** -> **Analog to digital converters** -> **i.MX 7Dual ADC driver**
- **Device Drivers** -> **Industrial I/O support** -> **Analog to digital converters** -> **i.MX 8QXP ADC driver**
- **Device Drivers** -> **Industrial I/O support** -> **Analog to digital converters** -> **i.MX 93 ADC driver**

4.1.6 Programming interface

Linux IIO provides some system interface to obtain the raw ADC data from the related input. Users can also set the sample rate in the available sample rate group. The ADC controllers system interface is located:

- `/sys/devices/soc0/soc.1/2200000.aips-bus/2280000.adc/iio:device0:`
- `/sys/devices/soc0/soc.1/2200000.aips-bus/2284000.adc/iio:device1:`

The following table lists the software interfaces.

Table 38. Software interfaces

Software interface	Description
`in_voltage0_raw` - `in_voltage3_raw`	`cat in_voltage0_raw` to get the raw ADC data
`sampling_frequency_available`	`cat sampling_frequency_available` to get the available sample rate group
`in_voltage_sampling_frequency`	`cat in_voltage_sampling_frequency` to show the current sample rate `echo value` > `in_voltage_sampling_frequency` to set the sample rate

4.2 ENET IEEE-1588

4.2.1 Introduction

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

To allow 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.

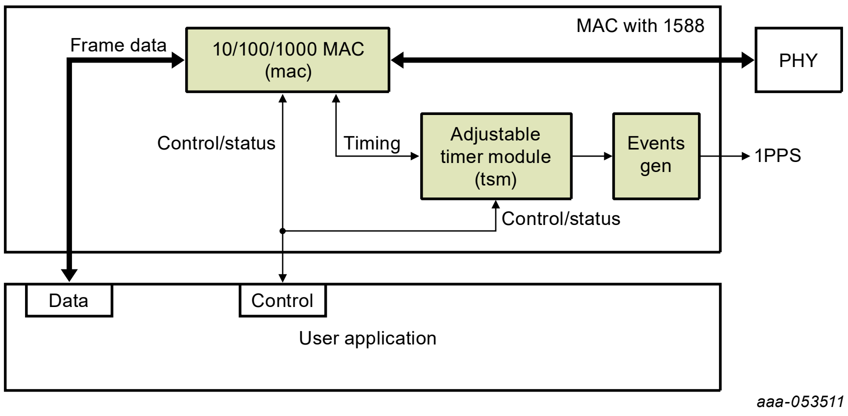

Figure 9. IEEE 1588 cunctions overview

4.2.1.1 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 each 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 also 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.

The software implements a handshaking procedure by setting the `ff_tx_ts_frm` signal. When it transmits the frame, it needs a timestamp 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 therefore do not interfere with the timestamp capture.

4.2.1.2 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 the 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.

4.2.2 Software operation

The 1588 driver has the following functions:

- Module initialization: Initializes the module with the device-specific structure, and registers a character driver.
- Interrupt servicing routine: Supports events, such as `TS_AVAIL` and `TS_TIMER`. The driver shares the interrupt servicing routine with the FEC driver.

4.2.2.1 Source code structure

The table below lists the source files in the `drivers/net/ethernet/freescale` directory.

Table 39. ENET 1588 file list

File	Description
`drivers/net/ethernet/frescale/fec.h`	Header file defining registers
`drivers/net/ethernet/freescale/fec_ptp.c`	ENET 1588 timer

4.2.2.2 Menu configuration options

By default, ENET 1588 is enabled.

4.2.2.3 Programming interface

The 1588 driver complies with the Linux PTP protocol stack interface. Stack-specific defines are added to the header file (`fec.h`).

4.2.3 1588 Stack introduction

This release supports the following type of the 1588 Stack:

- Linux PTP stack
  This software is an implementation of the Precision Time Protocol (PTP) according to the IEEE standard 1588 for Linux OS. The dual design goals are to provide a robust implementation of the standard and to use the most relevant and modern Application Programming Interfaces (API) offered by the Linux OS kernel. Supporting legacy APIs and other platforms is not a goal. The software is copyrighted by the authors and is licensed under the GNU General Public License.

The software development is hosted at Source Forge: sourceforge.net/projects/linuxptp/.

4.2.3.1 Linux PTP stack features

Linux PTP supports the following features:

- Ordinary/Boundary Clock
- Best master clock algorithm
- Transport over UDP/IPv4, UDP/IPv6, and IEEE 802.3
- Transparent clock (E2E/P2P)
- Slave only
- Supporting IEEE 802.1AS-2011 in the role of the end station

4.2.3.2 Using Linux PTP

Run the PTP4 1588 stack binary with the following commands.

Linux PTP:

```
Transport on UDP IPV4 with E2E delay mechanism: ptp4l -A -4 -H -m -i eth0
Transport on UDP IPV4 with P2P delay mechanism: ptp4l -P -A -4 -H -m -i eth0
Transport on UDP IPV6 with E2E delay mechanism: ptp4l -A -6 -H -m -i eth0
Transport on UDP IPV6 with P2P delay mechanism: ptp4l -P -A -6 -H -m -i eth0
Transport on IEEE 802.3 with E2E delay mechanism: ptp4l -A -2 -H -m -i eth0
Transport on IEEE 802.3 with P2P delay mechanism: ptp4l -P -A -2 -H -m -i eth0
```

4.3 Enhanced Configurable Serial Peripheral Interface (ECSPI)

4.3.1 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-slave select
- Multiclient requests

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 are as follows:

- 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)

The ECSPI module supports the following features:

- Implements each of the functions required by an ECSPI module to interface to the Linux OS
- Multiple SPI master controllers
- Multiclient synchronous requests

4.3.2 Software operation

The following sections describe the ECSPI software operation.

4.3.3 SPI subsystem in the Linux OS

The ECSPI driver layer is located between the client layer (SPI-NOR Flash are examples of clients) and the hardware access layer. The following figure shows the block diagram for the SPI subsystem in the Linux OS.

The SPI requests to go into I/O queues. Requests for a given SPI device are executed in FIFO order and they are completed 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.

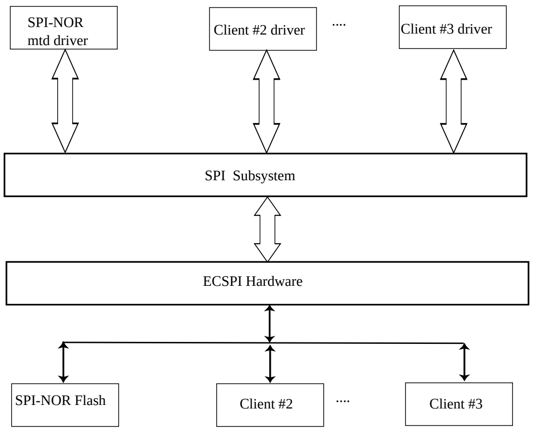

Figure 10. SPI subsystem

All SPI clients must have a protocol driver associated with them and must share the same controller driver. Only the controller driver can interact with the underlying SPI hardware module. The following figure shows how the different SPI drivers are layered in the SPI subsystem.

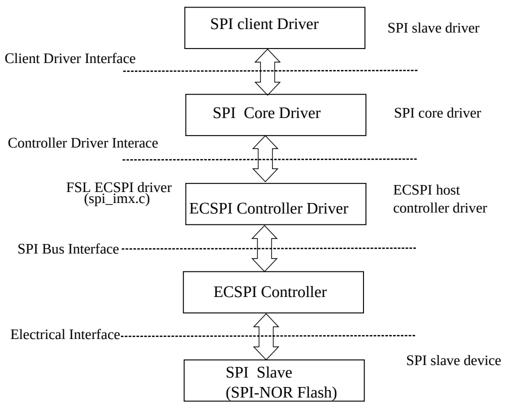

Figure 11. Layering of SPI drivers in SPI subsystem

4.3.4 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 the `sysfs` interface.

4.3.5 Standard operations

The ECSPI driver is responsible for implementing standard entry points for initialization, exit, chip selection, and transfer. The driver implements the following functions:

- The initialization function `spi_imx_init()` registers the `device_driver` structure.
- The probe function `spi_imx_probe()` performs initialization and registration of the SPI device-specific structure with the 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.
- The chip selection function `spi_imx_chipselect()` configures the hardware ECSPI for the current SPI device, and sets the word size, transfer mode, and data rate for this device.

- The SPI transfer function `spi_imx_transfer()` handles data transfers operations.
- The SPI setup function `spi_imx_setup()` initializes the current SPI device.
- The SPI driver ISR `spi_imx_isr()` is called when the data transfer operation is completed and an interrupt is generated.

4.3.6 ECSPI synchronous operation

The figure below shows how the ECSPI provides synchronous read/write operations.

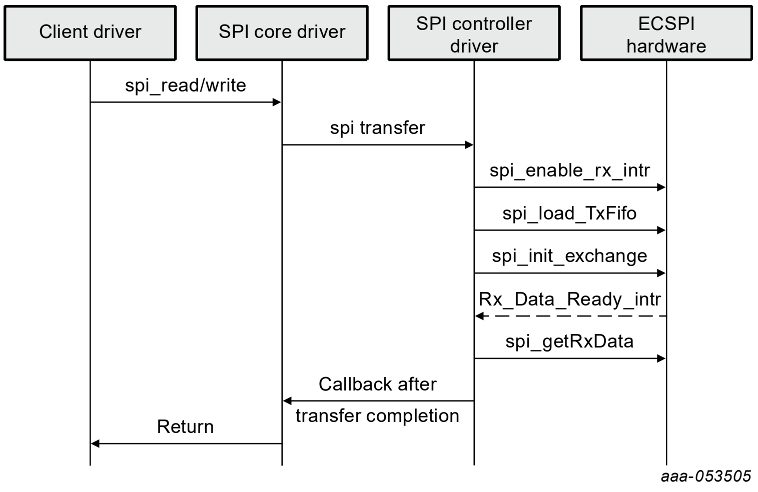

Figure 12. ECSPI synchronous operation

4.3.7 Source code structure

The table below shows the source files available in the `drivers/spi` directory.

Table 40. ECSPI driver files

File	Description
`driveers/spi/spi-imx.c`	SPI Master Controller driver

4.3.8 Menu configuration options

In menu configuration, enable the following 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**.

4.3.9 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`.

4.3.10 Interrupt requirements

The SPI interface generates interrupts.

The following table lists the ECSPI interrupt requirements.

Table 41. 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 (maximum supported by the SPI interface) with an 8-bit transfer length.

4.4 Fast Ethernet Controller (FEC)

4.4.1 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, and 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 the interface name `ethx`. The driver auto-probes the external adaptor (PHY device).

4.4.2 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, 10/100 Mbps RMII, and 10/100/1000 Mbps RGMII. In addition, the FEC supports 1000 Mbps RGMII, which uses 4-bit reduced GMII operating at 125 MHz.

This section provides a brief overview of the device functionality. For details, see the FEC chapter of the Applications Processor Reference Manual.

In MII mode, there are 18 signals defined by the IEEE 802.3 standard and supported by the EMAC. The MII, RMII, and RGMII modes use a subset of the 18 signals. These signals are listed in the table below.

Table 42. Pin usage in MII, RMII, and RGMII modes

Direction	EMAC pin name	RMII usage	RGMII usage (not supported by i.MX 6UltraLite)
In/Out	FEC_MDIO	Management Data Input/ output	Management Data Input/Output
Out	FEC_MDC	General output	Management Data Clock

Table 42. Pin usage in MII, RMII, and RGMII modes*...continued*

Direction	EMAC pin name	RMII usage	RGMII usage (not supported by i.MX 6UltraLite)
Out	FEC_TXD[0]	Data out, bit 0	Data out, bit 0
Out	FEC_TXD[1]	Data out, bit 1	Data out, bit 1
Out	FEC_TXD[2]	Not Used	Data out, bit 2
Out	FEC_TXD[3]	Not Used	Data out, bit 3
Out	FEC_TX_EN	Transmit Enable	Transmit Enable
Out	FEC_TX_ER	Not Used	Not Used
In	FEC_CRS	Not Used	Not Used
In	FEC_COL	Not Used	Not Used
In	FEC_TX_CLK	Not Used	Synchronous clock reference (REF_CLK, can connect from PHY)
In	FEC_RX_ER	Receive Error	Not Used
In	FEC_RX_CLK	Not Used	Synchronous clock reference (REF_CLK, can connect from PHY)
In	FEC_RX_DV	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
In	FEC_RXD[1]	Data in, bit 1	Data in, bit 1
In	FEC_RXD[2]	Not Used	Data in, bit 2
In	FEC_RXD[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 into the following parts. For details, see the Applications Processor Reference Manuals.

- 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 the carrier sense to become inactive, and then determines if the carrier sense stays inactive for 60-bit times. If the transmission begins after waiting an additional 36-bit times (96-bit times after the 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. 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 received 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 that 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 operations, such as PHY discovery, link status, and state machine. The MDIO bus will be created in FEC driver and registered into the system. See `Documentation/networking/phy.txt` under the Linux OS source directory for more information.

4.4.3 Software operation

The FEC driver supports the following functions:

- Module initialization: Initializes the module with the device-specific structure.
- RX/TX transmission.
- Interrupt servicing routine.
- PHY management.
- FEC management, such as initialization, start, and stop.
- The i.MX 6 FEC module uses the little-endian format.

4.4.4 Source code structure

The table below lists the source files. They are available at the `drivers/net/ethernet/freescale` directory.

Table 43. FEC driver files

File	Description
`drivers/net/ethernet/freescale/fec.h`	Header file defining registers
`drivers/net/ethernet/freescale/fec_main.c`	Linux driver for Ethernet LAN controller

4.4.5 Menu configuration options

Configure the kernel to provide for 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 configurations in the `menuconfig` if needed.

4.4.6 Programming interface

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

`fec.h` defines the structure for the register access and the structure 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 */
};
struct bufdesc_ex {
       struct bufdesc desc;
       unsigned long cbd_esc;
       unsigned long cbd_prot;
       unsigned long cbd_bdu;
       unsigned long ts;
       unsigned short res0[4];
};
/*
*      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 */
```

4.4.6.1 Getting a MAC Address

The MAC address can be set through the kernel command line, kernel device tree DTS file, OCOTP, or MAC registers set by bootloader, such as U-Boot. The FEC driver uses it to configure the MAC address for the network device. In general, use kernel command line in a form of fec.macaddr=0x00,0x04,0x9f,0x01,0x30,0xe0 to set the MAC address. Due to certain pin conflicts (FEC RMII mode needs 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.

4.5 FlexCAN

4.5.1 Introduction

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 to meet 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 the PF_CAN protocol family.

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

The FlexCAN on the i.MX 8QuadMax/8QuadXPlus supports the CAN FD protocol.

4.5.2 Software operation

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

The CAN network device driver interface provides a generic interface to set up, configure, and monitor the CAN network devices. The user can then configure the CAN device, like setting the bit-timing parameters, through 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 the CAN bus, such as bit-timing setting. For details, see the kernel doc: `Documentation/networking/can.txt`.

4.5.3 Source code structure

The table below lists the driver source file available in `drivers/net/can`.

Table 44. FlexCAN driver files

File	Description
`drivers/net/can/flexcan/`	FlexCAN driver

4.5.4 Menu configuration options

The following kernel configuration options are provided for this module.

- **CONFIG_CAN**: Build support for the PF_CAN protocol family. In `menuconfig`, this option is available under: **Networking** -> **CAN bus subsystem support**.
- **CONFIG_CAN_RAW**: Build support for the 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 the 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 the 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 the FlexCAN device driver. In `menuconfig`, this option is available under:
Networking -> **CAN bus subsystem support** -> **CAN Device Driver** -> **Freescale FlexCAN**.

4.6 Inter-IC (I2C)

4.6.1 Introduction

LPI2C is a bidirectional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices.

The LPI2C driver for Linux OS has two parts:

- Bus driver-low level interface that is used to communicate with the LPI2C bus
- Chip driver-interface between other device drivers and the LPI2C bus driver

The I2C bus driver is a low-level interface that is used to interface with the I2C bus. This driver is invoked by the I2C chip driver and it 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 bus driver to transfer data over the I2C bus. This bus driver supports:

- Compatibility with the I2C bus standard
- Bit rates up to 1 Mbps
- Standard I2C master mode and slave mode
- Power management features by suspending and resuming I2C

4.6.2 LPI2C bus driver overview

The LPI2C bus driver is invoked only by the 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 LPI2C bus driver to transfer data over the LPI2C 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
- Interrupt-driven, byte-by-byte data transfer
- Standard I2C master mode and slave mode

4.6.3 I2C device driver overview

The I2C device driver implements all the Linux I2C data structures that are required to communicate with the LPI2C 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 LPI2C 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 LPI2C bus driver and calls the appropriate function in the LPI2C 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

4.6.4 Software operation

The I2C driver for the Linux OS has two parts:

- An I2C bus driver
- An I2C chip driver

4.6.5 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.

During normal communication, it times out and returns an error when the transfer has some error condition, for example, when NACK is detected. When error condition occurs, the I2C driver should stop the current transfer.

4.6.6 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. 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.

4.6.7 Driver features

The LPI2C driver supports the following features:

- I2C communication protocol
- I2C master mode and slave mode of operation

4.6.8 Source code structure

The table below shows the driver source files available in `drivers/i2c/busses`.

Table 45. I2C driver file

File	Description
`drivers/i2c/busses/i2c-imx-lpi2c.c`	LPI2C bus driver for i.MX 7ULP, i.MX 8ULP, i.MX 8X, i.MX 93, i.MX 94, and i.MX 95
`drivers/i2c/busses/i2c-imx.c`	I2C bus driver for i.MX 6, i.MX 7, and i.MX 8M

4.6.9 Menu configuration options

Configure the kernel option to enable the module by `menuconfig`:

For i.MX 6, i.MX 7 and i.MX 8M, select **Device Drivers** -> **I2C support** -> **I2C Hardware Bus support** -> **IMX I2C interface**.

For i.MX 8, i.MX 8X, i.MX 93, and i.MX 95, select **Device Drivers** -> **I2C support** -> **I2C Hardware Bus support** -> **IMX Low Power I2C interface**.

4.6.10 Programming interface

The LPI2C device driver can use the standard SMBus interface to read and write the registers of the device connected to the LPI2C bus. For more information, see `include/linux/i2c.h`.

4.7 Media Local Bus

4.7.1 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.

The MediaLB module implements the Physical Layer and Link Layer of the MediaLB specification, interfacing the i.MX to the MediaLB controller.

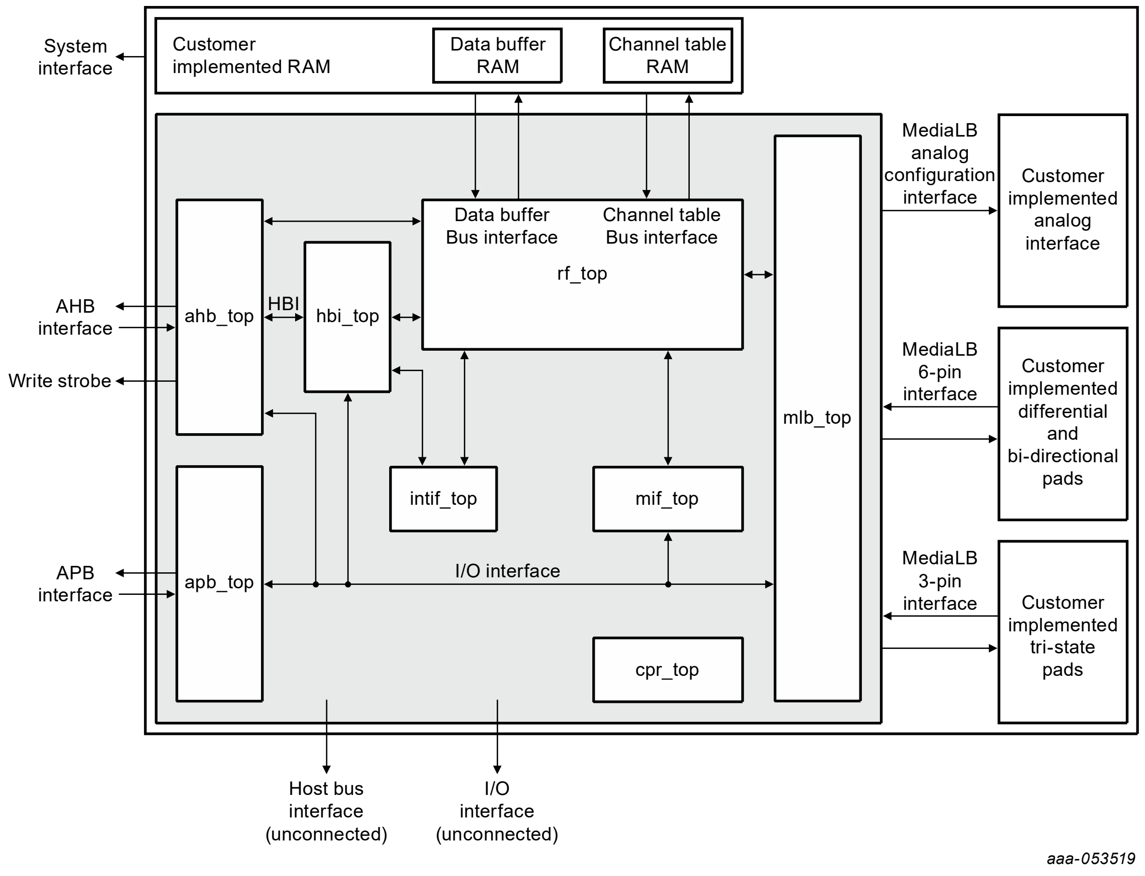
System interface
Customer implemented RAM
Data buffer RAM
Channel table RAM
MediaLB analog configuration interface
Customer implemented analog interface
Data buffer Bus interface
Channel table Bus interface
rf_top
AHB interface
Write strobe
ahb_top
HBI
hbi_top
MediaLB 6-pin interface
Customer implemented differential and bi-directional pads
mlb_top
intif_top
mif_top
APB interface
apb_top
I/O interface
MediaLB 3-pin interface
Customer implemented tri-state pads
cpr_top
Host bus interface (unconnected)
I/O interface (unconnected)
aaa-053519

Figure 13. 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 64 logical channels and up to 64 physical channels. Each logical channel is referenced using a unique channel address and represents a unidirectional data path between a MediaLB device transmitting the data and the MediaLB device(s) receiving the data.

The supported features are as follows:

- Synchronous, asynchronous, control, and isochronous channels.
- Up to 64 logical channels and 64 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.
- 256Fs, 512Fs, and 1024Fs frame rates.
- Asynchronous, control, synchronous, and isochronous channel types.
- The following configurations to MLB device module:
  - Frame rate

  – Device address
  – Channel address
- MLB channel exception gets interface. All the channel exceptions are sent and handled by the application.

4.7.2 MLB driver overview

The MLB driver is designed as a common Linux OS 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.

The following figure shows the MLB driver architecture.

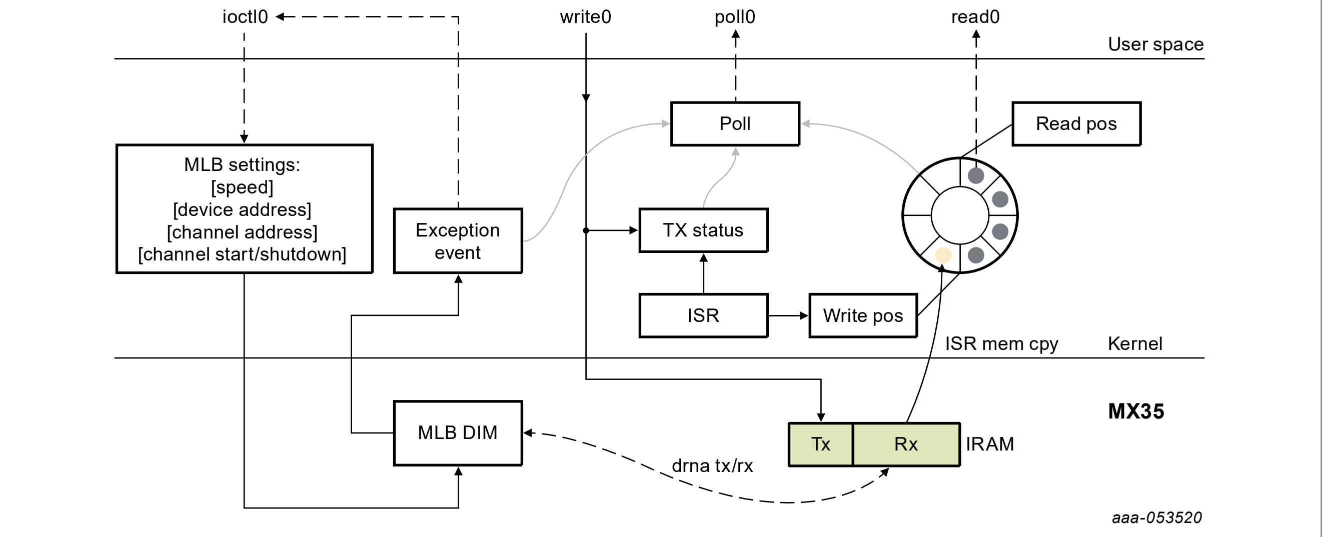

Figure 14. MLB driver architecture diagram

The MLB driver creates four minor devices. These four devices support control TX/RX channel, asynchronous TX/RX channel, synchronous TX/RX channel, and isochronous TX/RX channel. Their device files are `/dev/ctrl`, `/dev/async`, `/dev/sync`, and `/dev/isoc`. Each minor device has the same interfaces, and handles both TX and RX operations. The following description is for both control and asynchronous devices.

The driver uses IRAM as the MLB device module TX/RX buffer. All the data transmission and reception between module and IRAM are 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 copies 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 the ring buffer. After the read is 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 positions are 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 the 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.

4.7.3 Software operation

The MLB driver provides a common interface to the application.

- Packet read/write: BLOCK and NON-BLOCK 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 1024 Bytes.
- Polling: The MLB driver provides a polling interface, which polls for three status. The application can use it to get the 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`: The MLB driver provides the following IOCTL:

```
MLB_SET_FPS
```

Argument type: unsigned int.

Set the frame rate. The argument must be 256, 512, or 1024.

```
MLB_GET_VER
```

Argument type: unsigned long.

Get the MLB device module version, which is 0x02000202 by default on the i.MX 35.

```
MLB_SET_DEVADDR
```

Argument type: unsigned char.

Set the 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
```

Start up the corresponding type of the channel for transmit and reception.

```
MLB_CHAN_SHUTDOWN
```

Shut down the corresponding type of the channel.

```
MLB_CHAN_GETEVENT
```

Argument type: unsigned long.

Get the exception event from the 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
```

4.7.4 Source code structure

The table below lists the MLB driver source files.

Table 46. MLB driver source files

File	Description
`drivers/mxc/mlb/mxc_mlb.c`	Source file for MLB driver
`include/linux/mxc_mlb.h`	Include file for MLB driver

4.7.5 Menu configuration options

In menu configuration, enable the following module:

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

4.8 PCI Express Root Complex

4.8.1 Introduction

The PCI Express hardware module, contained in the i.MX SoC, can either be configured to act as a Root Complex or a PCIe Endpoint. This section describes the PCI Express Root Complex implementation on i.MX SoC families. It also describes the drivers needed to be configured and operated on i.MX PCI Express device as Root Complex.

PCI Express (PCIe) is a Third-Generation I/O Interconnect, targeting low cost, high volume, multiplatform interconnection usages. It has the concepts with earlier PCI and PCI-X and offers backward compatibility for existing PCI software with the following differences:

- PCIe is a point-to-point interconnect.
- Serial link between devices.
- Packet-based communication.
- Scalable performance through aggregated Lanes from X1 to X16.
- PCIe switch is needed to have connection between more than two PCIe devices.

4.8.2 Terminologies and conventions

The following terminologies and conventions are used in this section:

- 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
  - 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.

– 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
  – Logically, an interface between a component and a PCI Express Link.
  – 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
  – 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.
  – 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. An Upstream Flow is the communication moving towards RC.

4.8.3 PCIe topology on i.MX

There is one PCIe port on the i.MX. Currently, only the RC mode is enabled in the Linux BSP.

The following figure shows the diagram of the PCIe RC port on the i.MX.

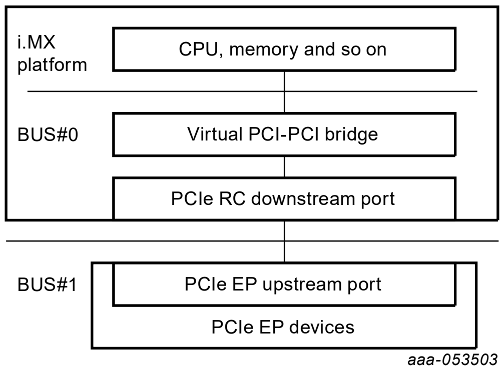

Figure 15. Diagram of the PCIe RC port on the i.MX

PCI Enumeration Mapping

As PCI Express is a point-to-point topology, to maintain compatibility with legacy PCI Bus-Device notion used for software enumeration, we introduce the following concepts, which allow various nodes and their internals to be identified (such as PCIe Switches) in terms of PCI devices/functions:

- Host Bridge: A bridge, integrated into RC to have PCI compatible connection to the Host. The PCI side of this bridge is Bus #0 always. This means that the device on this bus is 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 that each port has a primary and secondary PCI bus and the downstream is mapped into the remote configuration space.
- The root port associated virtual bridge has Bus #0 on the primary side with the secondary bus on the downstream.
- Each PCIe Switch is viewed as a collection of as many virtual PCI-PCI bridges as the number of downstream ports, connected to a virtual PCI bus, which is actually the 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 the downstream port of a switch or of the root port.

4.8.4 Features

The followings are the various features supported by the i.MX as a PCI Express Root Complex driver.

- Express Base Specification Revision 2.0 or 3.0 compliant.
- Gen2 operation with x1 link supporting 5 GT/s raw transfer rate in a single direction on i.MX 6, i.MX 7, i.MX 8M Quad, and i.MX 8M Mini. Gen3 speed on other i.MX platforms.
- Support Legacy Interrupts (INTx) and MSI.
- It fits into the Linux PCI Bus framework to provide PCI compatible software enumeration support.
- In addition, it provides an 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 OS to handle AER, ASPM, and so on.
- Interrupt handling facility for EP drivers either as Legacy Interrupts (INTx).
- Access to EP I/O BARs through generic I/O accessories in the Linux PCI subsystem.
- Seamless handling of PCIe errors.
- Supports the L0, L0s, L1, and L1 Substate ASPM power management.
- i.MX 95 and i.MX 943 PCIe support the PCIe link recovery when the link is down abnormally.

4.8.5 Linux OS PCI subsystem and RC driver

In the Linux OS, the PCI implementation can be divided into the following main components: PCI BIOS architecture-specific Linux OS implementation, Host Controller (RC) Module, and Core.

- PCI BIOS Architecture-specific Linux OS 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 are assigned before the Linux OS is 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 the PCI Core, which is called during PCI system initialization and accessing the 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 the 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).

4.8.6 PCIe driver source files

Table 47. Source files

File	Description
`drivers/pci/controller/dwc/pci-imx6.c`	i.MX 6 PCIe source

4.8.7 System resource: memory layout

The PCIe memory layout is defined in the PCIe DT-binding node according to different DTS files.

- `dbi` region: The register region of the PCIe RC, used to access the registers of the PCIe RC controller.
- `config`: Configuration outbound region, used to access the configuration registers of the remote EP device.
- `ranges`: Three address-cells, and two size-cells, used to define the IO space and memory space of PCIe.

For exmaple, the `dbi`, `config`, IO space, and memory space of the i.MX 8M Mini PCIe:

```
pcie0: pcie@33800000 {
 compatible = "fsl,imx8mm-pcie";
 ...
 reg = <0x33800000 0x400000>, <0x1ff00000 0x80000>;
 reg-names = "dbi", "config";
 #address-cells = <3>;
 #size-cells = <2>;
 device_type = "pci";
 bus-range = <0x00 0xff>;
 ranges = <0x81000000 0 0x00000000 0x1ff80000 0 0x00010000>, /* downstream I/O
 64KB */
   <0x82000000 0 0x18000000 0x18000000 0 0x07f00000>; /* non-prefetchable memory
 */
 ...
};
```

4.8.8 Interrupters

Both INTx and MSI can be used as PCIe interrupters. Add `pci=nomsi` into the kernel command line to disable MSI mode, and roll back to INTx interrupter mode.

The INTx or MSI map is defined in the corresponding `dt-binding` nodes in the DTS files.

For example, the interrupters of i.MX 8M Mini PCIe:

```
pcie0: pcie@33800000 {
 compatible = "fsl,imx8mm-pcie";
 ...
 interrupts = <GIC_SPI 122 IRQ_TYPE_LEVEL_HIGH>;
 interrupt-names = "msi";
 #interrupt-cells = <1>;
 interrupt-map-mask = <0 0 0 0x7>;
 interrupt-map = <0 0 0 1 &gic GIC_SPI 125 IRQ_TYPE_LEVEL_HIGH>,
   <0 0 0 2 &gic GIC_SPI 124 IRQ_TYPE_LEVEL_HIGH>,
   <0 0 0 3 &gic GIC_SPI 123 IRQ_TYPE_LEVEL_HIGH>,
   <0 0 0 4 &gic GIC_SPI 122 IRQ_TYPE_LEVEL_HIGH>;
 ...
};
```

4.8.9 PCIe link recovery

i.MX PCIe root complex supports link recovery if the flag `IMX_PCIE_FLAG_LINK_NOTIFY` is present.

After the system boots up, and a PCIe device is detected and enumerated, a link can be re-established to up after the link is down.

The following items are supported in this new feature:

- The link-down interrupt would be triggered when a link-down event occurs.
- The link can be re-established to the up state.
- The link-up interrupter can be triggered after the link is up from link-down state.

Out-of-scope feature:

The EP device functions after the exceptional link-down and link-up events.

It is the capability of the EP to recover its own functions when the link-down and link-up events occur.

The test procedure on the i.MX 95 and i.MX 943 platforms is as follows:

1. After the system boots up and the EP devices are detected and enumerated successfully, toggle the `ltssm_en` bit to trigger the link-down event.

```
PCIE0:
./memtool 4c341058=0;./memtool 4c341058=1;
PCIE1:
./memtool 4c3c1058=0;./memtool 4c3c1058=1;
```

2. Then, the link event interrupter is triggered and link recovery is done automatically.

Reference logs:

```
root@imx943evk:~# ./memtool 4c3c1058=0;./memtool 4c3c105
[ 229.023382] pci0003:00: Recovering root ports due to Link Down
[ 229.335757] imx6q-pcie 4c380000.pcie: PCIe Gen.1 x1 link up
[ 229.467747] pcieport 0003:00:00.0: Root port has been reset
```

4.9 USB

4.9.1 Introduction

The universal serial bus (USB) driver implements a standard Linux driver interface to the CHIPIDEA 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 CHIPIDEA 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 and Host3).
- High-speed Host Only core (OTG2), high-speed, full speed, and low devices are supported. A USB2Pci bridge is connected to OTG2 by default. Therefore, users may not be able to connect other USB devices on this port.
- High-speed Inter-Chip core (Host2).
- 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.

4.9.2 Architectural overview

The USB host system is composed of a number of hardware and software layers.

The figure below shows a conceptual block diagram of the building block layers in a host system that supports USB 2.0.

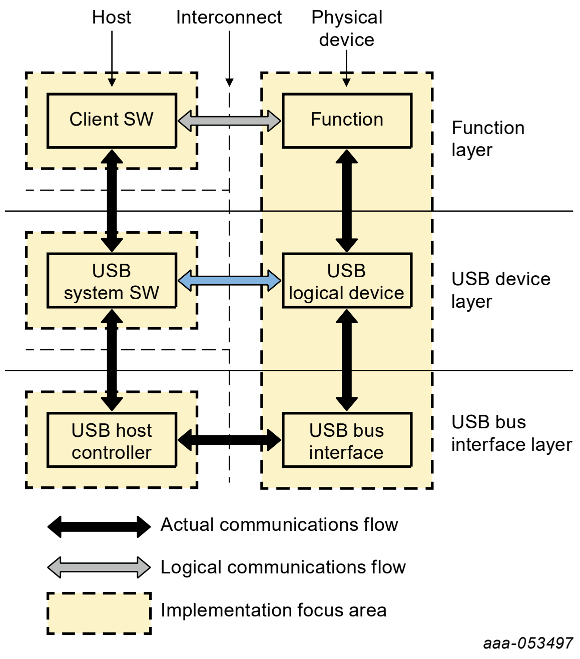

Figure 16. USB block diagram

4.9.3 Hardware operation

For the information on hardware operations, see the EHCI `spec.ehci-r10.pdf`. The specification is available at [Enhanced Host Controller Interface for USB 2.0: Specification](#).

4.9.4 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. For OTG, ID dynamic switch host/device modes are supported. Currently, the runtime suspend for USB is supported, which means that when the USB is not in use (both for host and peripheral mode), the USB enters low-power mode.

4.9.5 Source code structure

The table below lists the USB source in `drivers/usb`.

Table 48. Chipidea USB driver files

File	Description
`drivers/usb/chipidea/core.c`	Chipidea IP core driver
`drivers/usb/chipidea/udc.c`	Chipidea peripheral driver
`drivers/usb/chipidea/host.c`	Chipidea host driver
`drivers/usb/chipidea/otg.c`	Chipidea OTG driver
`drivers/usb/chipidea/otg_fsm.c`	Chipidea OTG HNP and SRP driver
`drivers/usb/chipidea/ci_hdrc_imx.c`	i.MX glue layer
`drivers/usb/chipidea/usbmisc_imx.c`	i.MX SoC abstract layer
`drivers/usb/phy/phy-mxs-usb.c`	i.MX 6, i.MX 7ULP, or i.MX 8ULP USB physical driver

4.9.6 Menu configuration options

In menu configuration, enable the following modules:

```
> Device Drivers > USB support
    <*>   Chipidea Highspeed Dual Role Controller
    [*]     Chipidea device controller
    [*]     Chipidea host controller
```

```
> Device Drivers > USB support > USB Physical Layer drivers
    <*> NOP USB Transceiver Driver
    <*> Freescale MXS USB PHY support
```

```
> Device Drivers > USB support > USB Gadget Support
```

- `CONFIG_USB`: Build support for Host-side USB.
- `CONFIG_USB_EHCI_HCD EHCI`: HCD (USB 2.0) support. Default `y`.
- `CONFIG_USB_CHIPIDEA`: Chipidea high-speed Dual-Role Controller. Default `y`.
- `CONFIG_USB_CHIPIDEA_UDC`: Chipidea device controller. Default `y`.
- `CONFIG_USB_CHIPIDEA_HOST`: Chipidea host controller. Default `y`.
- `CONFIG_USB_GADGET`:bUSB Gadget Support. Default `y`.
- `CONFIG_USB_MXS_PHY`: Freescale MXS USB PHY support. Default `y`.
- `CONFIG_NOP_USB_XCEIV`: NOP USB Transceiver Driver. Default `y`.

Note: *Some platforms, such as i.MX 6UL, i.MX 7ULP, and i.MX 8ULP, need the Freescale MXS USB PHY driver. Other platforms, such as i.MX 7Dual, i.MX 8M Mini, i.MX 93, and i.MX 95, need the NOP USB Transceiver Driver.*

4.9.7 USB wakeup usage

The following example is for the OTG port and the first EHCI device.

Controller wakeup settings, after the following settings, the VBUS and ID are the wakeup source.

```
echo enabled > /sys/bus/platform/devices/20c9000.usbphy/power/wakeup
echo enabled > /sys/bus/platform/devices/2184000.usb/power/wakeup
echo enabled > /sys/bus/platform/devices/ci_hdrc.0/power/wakeup
```

EHCI wakeup settings, after the following settings, the host has the wakeup ability, such as remote wakeup and connect/disconnect wakeup.

```
echo enabled > /sys/bus/usb/devices/usb1/power/wakeup
echo enabled > /sys/bus/usb/devices/1-1/power/wakeup
```

Note: *When the OTG mode switches from the host to the device, it deletes the EHCI wakeup, and the user needs to set it again before the system suspending.*

4.9.8 How to close the USB child device power

The following commands are used 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)
```

4.9.9 Changing the controller operation mode

To change the default settings, modify the DTS file as follows:

```
dr_mode =  "host"  /* Set dr_mode = "host" /* Set controller as host only mode
 */
dr_mode = "peripheral" /* Set controller as peripheral only mode*/
dr_mode = "otg" /* Set controller as dual role mode */ as gadget-only mode */
dr_mode =  "peripheral" /* Set controller as host-only mode */
dr_mode =  "otg" /* Set controller as otg mode */
```

4.9.10 Loadable module support

The modprobe utility automatically loads the modules, which have dependency among all modules.

The loading command is as follows:

```
modprobe phy-generic
modprobe phy_mxs_usb
modprobe ci_hdrc_imx
```

The unloading command is as follows:

```
modprobe -r ci_hdrc_imx
modprobe -r phy_mxs_usb
modprobe -r phy-generic
```

4.9.11 USB charger detection

The i.MX SoC has the USB charger detection ability, but it has no charging ability. The user can use the `/sys` entry to know the USB charger type, charging current, and whether the charger exists, as shown below:

```
cat /sys/class/power_supply/imx6_usb_charger/type
cat /sys/class/power_supply/imx6_usb_charger/current_max
cat /sys/class/power_supply/imx6_usb_charger/present
```

Currently, the i.MX 6 SABRE-SD board does not support the USB charger detection function. i.MX 6 SABRE-Auto supports the function.

4.9.12 Embeded host certification

4.9.12.1 Adding the TPL-support property

To pass the embeded host USB certification, add `tpl-support` in DTS to enable Targeted Peripheral List (TPL). For example, to enable TPL on the Host port of i.MX 6UltraLite EVK board (`imx6ul-14x14-evk.dts`):

```
&usbotg2 {
dr_mode = "host";
disable-over-current;
tpl-support;
status = "okay";
};
```

4.9.12.2 VBUS control

The VBUS should be kept off until the Linux USB host function is ready. For example, on the i.MX 6UltraLite EVK board, because the pin is multiplexed with the touch function, you need to rework the board to make the GPIO (GPIO1_IO02) selected for VBUScontrol.

Disable the touch function in its DTS file (`imx6ul-14x14-evk.dts`) as follows:

```
&tsc {
pinctrl-names = "default";
pinctrl-0 = <&pinctrl_tsc>;
xnur-gpio = <&gpio1 3 0>;
measure_delay_time = <0xffff>;
pre_charge_time = <0xfff>;
status = "disabled";
};
```

Add VBUS GPIO pinctrl and its regulator node:

```
pinctrl_usb_otg2: usbotg2grp {
          fsl,pins = <
              MX6UL_PAD_GPIO1_IO02__GPIO1_IO02        0xb0
          >;
};
reg_usb_otg2_vbus: regulator@2 {
          compatible = "regulator-fixed";
          reg = <2>;
          pinctrl-names = "default";
          pinctrl-0 = <&pinctrl_usb_otg2>;
          regulator-name = "usb_otg2_vbus";
          regulator-min-microvolt = <5000000>;
          regulator-max-microvolt = <5000000>;
          gpio = <&gpio1 2 GPIO_ACTIVE_HIGH>;
          enable-active-high;
};
&usbotg2 {
  vbus-supply = <&reg_usb_otg2_vbus>;
  dr_mode = "host";
```

```
    disable-over-current;
    tpl-support;
    status = "okay";
  };
```

4.10 USB3

4.10.1 Introduction

For i.MX 8 and i.MX 8X families, a SuperSpeed USB IP from Cadence, and for i.MX 8M Plus and i.MX 95, a SuperSpeed USB IP (DWC3) from Synopsys are provided supporting USB 3.0, which includes a new transfer rate referred to as Super Speed (SS) USB with higher transfer rates and significantly faster than the USB 2.0 standard.

The supported features are as follows:

- Host mode is implemented with a Linux OS standard XHCI driver with super-speed supported and tested.
- For Device Mode, only a single queue is supported. Mass storage, ether, and serial are supported.

4.10.2 Source code structure

Table 49. CDNS3 USB3 driver source files

File	Description
`drivers/usb3/cdns3/cdns3-nxp-reg-def.h`	Register definitions
`drivers/usb3/cdns3/core.c`	USB3 core driver
`drivers/usb3/cdns3/core.h`	USB3 Core header
`drivers/usb3/cdns3/dev-regs-macro.h`	USB3 Macros
`drivers/usb3/cdns3/dev-regs-map.h`	USB3 Register mapping
`drivers/usb3/cdns3/gadget.c`	USB3 Gadget
`drivers/usb3/cdns3/gadget.h`	USB3 Gadget header
`drivers/usb3/cdns3/gadget-export.h`	USB3 Gadget Export header
`drivers/usb3/cdns3/host.c`	USB3 Host
`drivers/usb3/cdns3/host-export.h`	USB3 Host Export header
`drivers/usb3/cdns3/io.h`	USB3 IO

Table 50. DWC3 USB3 driver source files

File	Description
`drivers/usb/dwc3/core.c`	USB3 Core driver
`drivers/usb/dwc3/core.h`	USB3 Core header
`drivers/usb/dwc3/drd.c`	USB3 Dual-role switch driver
`drivers/usb/dwc3/host.c`	USB3 Host
`drivers/usb/dwc3/gadget.c`	USB3 gadget
`drivers/usb/dwc3/gadget.h`	USB3 gadget header
`drivers/usb/dwc3/io.h`	USB3 IO

Table 50. DWC3 USB3 driver source files*...continued*

File	Description
`drivers/usb/dwc3/dwc3-imx8mp.c`	NXP IMX specific Glue driver for i.MX 8M Plus and i.MX 95
`drivers/usb/dwc3/debugfs.c`	For Debug purposes
`drivers/usb/dwc3/trace.c`	For Trace purposes
`drivers/usb/dwc3/trace.h`	Trace header

4.11 Low Power Universal Asynchronous Receiver/Transmitter (LPUART)

4.11.1 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 eDMA-driven transmit/receive of characters.
- Standard Linux baud rates up to 4 Mbps.
- Transmit and receive characters with 7-bit, 8-bit, 9-bit, or 10-bit character length.
- 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 is set properly and to get the general information on the device. The UART type should be set to 52 as defined in the `serial_core.h` header file.
- Power management feature by suspending and resuming the UART ports.
- Standard TTY layer IOCTL calls.

All the UART ports can be accessed from the device files `/dev/ttyLP0` to `/dev/ttyLP1`.

4.11.2 Hardware operation

To determine the number of the UART modules available on the device, see the Applications Processor Reference Manual associated with the SoC.

Each UART hardware port is capable of the standard RS-232 serial communication.

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.

4.11.3 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 by enabling the DMA channel in the DTS file.

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. For more information, see the Linux documentation on the serial driver in the kernel source tree.

4.11.4 Driver features

The UART driver supports the following features:

- Baud rates up to 4 Mbps.
- Recognizes frame and parity errors.
- Recognizes the modem control signals.
- Ignores characters with frame, parity, and break errors if requested.
- Implements support for hardware flow control.
- Gets and sets the UART port information; certain flow control count information is not available in hardware-driven hardware flow control mode.
- Power management.
- Interrupt-driven and DMA-driven data transfer.

4.11.5 Source code structure

The table below shows the UART driver source files.

Table 51. UART driver files

File	Description
`drivers/tty/serial/fsl_lpuart.c`	LP UART driver

For the i.MX 8, i.MX 8X, and i.MX 8M, configuration options are specified in the device trees located in the `arch/arm64/boot/dts` directory.

4.11.6 Menu configuration options

The UART driver is enabled by default.

The menu configuration option is located at:

Device Drivers -> **Character devices** -> **Serial drivers** -> **Freescale LPUART serial port support [*] Console on Freescale LPUART serial port**

4.11.7 Programming interface

The UART driver implements all the methods required by the Linux serial API to interface with the UART port 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.

4.11.8 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.

4.12 Bluetooth

4.12.1 Bluetooth wireless technology introduction

The Bluetooth technology is a low-cost, low-power, and short-range wireless technology. It was designed as a replacement for cables and other short-range technologies like IrDA. The Bluetooth wireless technology operates in personal area range that typically extends up to 10 meters. For more information about Bluetooth wireless technology, see www.bluetooth.com/.

For i.MX, Bluetooth is supported with multiple vendors. For details, see the Section "Connectivity for Bluetooth wireless technology and Wi-Fi" in the *i.MX Linux User's Guide* (UG10163).

4.12.2 Bluetooth driver overview

i.MX uses the open source NXP Bluetooth driver. The Bluetooth software is divided into four parts as follows:

- 4-wire UART and TTY driver: It is the communication interface with the Bluetooth module.
- Bluetooth HCI device driver: NXP Bluetooth driver based on the Serdev driver for the NXP BT serial protocol based on running H:4. It is used for communication between Bluetooth device and host.
- Bluetooth kernel stack: Bluetooth framework and protocols implementation.
- Bluetooth user stack: Supplies several user-space utilities and integrates many profiles for use cases.

4.12.3 Bluetooth driver files

The Bluetooth driver source files are available in the kernel source directory.

- Bluetooth HCI device driver: `drivers/bluetooth/btnxpuart.c`
- Bluetooth kernel stack: `net/bluetooth/*`

4.12.4 Bluetooth stack

BlueZ is the official Linux standard Bluetooth protocol stack. It is the latest version of 5.x and is a Bluetooth stack for the Linux kernel-based family of operating systems. Its goal is to program an implementation of the Bluetooth wireless standards specifications for Linux. To use Linux Bluetooth subsystem, you need several user-space utilities like `hciconfig` and `bluetoothd`. These utilities and updates to Bluetooth kernel modules are provided in the BlueZ packages. For more information, see www.bluez.org/.

The BlueZ source code is available in the Git: `git://git.kernel.org/pub/scm/bluetooth/bluez.git`. The current BSP package tests pass with BlueZ 5.79.

4.12.5 Menu configuration options

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

- UART interface:
  - CONFIG_SERIAL_FSL_LPUART
  - CONFIG_SERIAL_IMX
  - CONFIG_TTY
- HCI interface:
  - CONFIG_BT_HCIUART
  - CONFIG_BT_NXPUART
- Bluetooth Stack:
  - CONFIG_BT
  - CONFIG_BT_RFCOMM
  - CONFIG_BT_RFCOMM_TTY
  - CONFIG_BT_BNEP
  - CONFIG_BT_BNEP_MC_FILTER
  - CONFIG_BT_BNEP_PROTO_FILTER
  - CONFIG_BT_HIDP

4.13 ENETC

4.13.1 Introduction

The NIC functionality in NETC is known as EtherNET Controller (ENETC). ENETC supports virtualization/ isolation based on PCIe Single Root IO Virtualization (SR-IOV), advanced QoS with 8 traffic classes and 4 drop resilience levels, and a full range of TSN standards and NIC offload capabilities. For more details, see Section "NET Controller (NETC) Domain" in the *i.MX Applications Processor Reference Manual* for i.MX 95 and i.MX 943.

4.13.2 Software operation

The ENETC PF driver supports the following features:

- 10 Mbps, 100 Mbps, 1 Gbps, 2.5 Gbps, and 10 Gbps port speeds: Full range of standard 802.3 Ethernet speeds.
- Half-duplex support at 10 Mbps and 100 Mbps speeds.
- EEE support: Energy Efficient Ethernet.
- Pause support: Recognizes and generates PAUSE frames with timing support for both receive and transmit.
- One-step and two-step timestamping support for PTP/IEEE1588/IEEE802.1AS-2020.
- Comprehensive statistics support enables system management and debugging.
- MAC and VLAN filtering: Filtering for multiple MAC addresses and VLANs.
- Virtualization: Hardware-supported isolation/virtualization through SR-IOV.
- Transmits and receives buffer descriptor rings to transfer packets to and from the host.
- XDP support (Both XDP copy mode and zero copy mode).
- Receives Side Scaling (RSS): Load balancing across multiple receive descriptor rings (multiple cores) implemented within an SI.
- Large Send Offload (LSO): Segmenting large TCP/UDP transmit units into multiple Ethernet frames.
- Receive Segment Coalesce (RSC): Coalescing multiple receive TCP segments into a single frame.
- VLAN tag extraction/insertion: Inserts predetermined tag on TX and removes expected tag on RX as seen by the SI.
- SI-Based VLAN: Removal and addition of SI-based VLAN.
- Checksum offload: IP and TCP/UDP checksum offload for transmit.
- Interrupt coalescing control: Interrupt coalescing can be set for each interrupt source.

- Time Specific Departure (TSD): Enables to specify the time when a frame is to be transmitted.
- TSN capabilities: Full range of TSN standards.
  - 802.1Qav: Credit-Based Shaper (CBS) support
  - 802.1Qci: Per Stream Filtering and Policing (PSFP) support
  - 802.1Qbv: Enhancements for Scheduled Traffic (EST) support
  - 802.1Qbu: Preemption support
- Wake-on-LAN (WOL) support.
- System suspend/resume support.

The ENETC VF driver supports the following features:

- Two-step timestamping support for PTP/IEEE1588/IEEE802.1AS-2020.
- MAC and VLAN filtering: Filtering for multiple MAC addresses and VLANs.
- Transmits and receives buffer descriptor rings to transfer packets to and from the host.
- XDP support (Both XDP copy mode and zero copy mode).
- Receive Side Scaling (RSS): Load balancing across multiple receive descriptor rings (multiple cores) implemented within an SI.
- Large Send Offload (LSO): Segmenting large TCP/UDP transmit units into multiple Ethernet frames.
- Receive Segment Coalesce (RSC): Coalescing multiple receive TCP segments into a single frame.
- VLAN tag extraction/insertion: Inserts predetermined tag on TX and removes expected tag on RX as seen by the SI.
- SI-Based VLAN: Removal and addition of SI-based VLAN.
- Checksum offload: IP and TCP/UDP checksum offload for transmit.
- Interrupt coalescing control: Interrupt coalescing can be set for each interrupt source.

4.13.3 Source code structure

The table below shows the source files.

Table 52. ENETC source

File	Description

`include/linux/fsl/enetc_mdio.h`
`include/linux/fsl/netc_*.h`
`include/linux/fsl/ntmp.h`
	Generic header files for multiple drivers to use

`drivers/net/ethernet/freescale/enetc/*`	ENETC PF and VF driver, EMDIO driver

4.13.4 Menu configuration options

The following kernel configuration options are provided for this module:

- `CONFIG_NXP_NETC_BLK_CTRL`: Build support for NETCMIX, IERB, and PRB, which provides pre-configuration for ENETC. In `menuconfig`, this option is available under **Device Drivers** -> **Network device support** -> **Ethernet driver support** -> **Freescale devices** -> **NETC blocks control driver**.
- `CONFIG_FSL_ENETC_MDIO`: Build support for EMDIO driver, which provides MDIO bus to manage the external PHYs. In `menuconfig`, this option is available under **Device Drivers** -> **Network device support** -> **Ethernet driver support** -> **Freescale devices** -> **ENETC MDIO driver**.
- `CONFIG_PTP_1588_CLOCK_NETC`: Build support for NETC 1588 Timer driver. This module is needed if the ENETC driver wants to support PTP synchronization. In `menuconfig`, this option is available under **Device Drivers** -> **PTP clock support** -> **NXP NETC Timer as PTP clock**.

- `CONFIG_FSL_ENETC4`: Build support for ENETC PF driver. In `menuconfig`, this option is available under **Device Drivers** -> **Network device support** -> **Ethernet driver support** -> **Freescale devices** -> **ENETC4 PF driver**.
- `CONFIG_FSL_ENETC_VF`: Build support for ENETC VF driver. In `menuconfig`, this option is available under **Device Drivers** -> **Network device support** -> **Ethernet driver support** -> **Freescale devices** -> **ENETC VF driver**.

4.14 ENETC 1588 Timer

4.14.1 Introduction

NETC 1588 Timer provides the current time with nanosecond resolution, precise periodic pulse, pulse on timeout (alarm), and time capture on external pulse support. This block capabilities support implementing time synchronization as required for IEEE 1588 and IEEE 802.1AS-2020. For more details, see Section "NET Controller (NETC) Domain" in the *i.MX Applications Processor Reference Manual* for i.MX 95 and i.MX 943.

4.14.2 Software operation

The NETC 1588 Timer driver supports the following features:

- PTP synchronization supports for ENETC and NETC Switch.
- PPS support.
- Generates pulses at a specific future time.
- External Trigger Timestamping support (EXTTS) support.

4.14.3 Source code structure

The table below shows the source files.

Table 53. ENETC 1588 Timer source

File	Description
`include/linux/fsl/netc_global.h`	Generic header file for multiple drivers to use
`drivers/ptp/ptp_netc.c`	NETC 1588 Timer driver

4.14.4 Menu configuration options

`CONFIG_PTP_1588_CLOCK_NETC` is used to build the support for this driver. In `menuconfig`, this option is available under **Device Drivers** -> **PTP clock support** -> **NXP NETC Timer as PTP clock**.

4.15 ENETC Switch

4.15.1 Introduction

NETC provides full 802.1Q Ethernet switch functionality, advanced QoS with 8 traffic classes and 4 drop resilience levels, and a full range of TSN standards capabilities. Switch CPU/host ENETC is fully integrated with the switch and does not require a back-to-back MAC. Instead, a light weight "pseudo MAC" provides the delineation between switch and Ethernet Controller. This translates to lower power (less logic and memory) and lower delay (as there is no serialization delay across this link). For more details, see Section "NET Controller (NETC) Domain" in the *i.MX Applications Processor Reference Manual* for i.MX 95 and i.MX 943.

4.15.2 Software operation

The NETC Switch driver follows the DSA framework. Currently, it supports the following features:

- Single port mode and bridge mode.
- FDB and VLAN filtering support.
- NXP switch tag support.
- 10 Mbps, 100 Mbps, 1 Gbps, and 2.5 Gbps port speeds: Full range of standard 802.3 Ethernet speeds.
- Half-duplex support at 10 Mbps and 100 Mbps speeds: Enables use of various low power PHYs.
- EEE support: Energy Efficient Ethernet.
- Pause support: Recognizes and generates PAUSE frames with timing support for both receive and transmit.
- One-step and two-step timestamping support for PTP/IEEE1588/IEEE802.1AS-2020
  - One-step: Updates PTP Correction field based on the current time and passed in timestamp value.
  - Two-step: Captures and reports the time of SFD.
- Comprehensive statistics support enables system management and debugging.
- TSN capabilities: Full range of TSN standards.
  - 802.1Qav: Credit-Based Shaper (CBS) support.
  - 802.1Qci: Per Stream Filtering and Policing (PSFP) support.
  - 802.1Qbv: Enhancements for Scheduled Traffic (EST) support.
  - 802.1Qbu: Preemption support.
- Flow actions support, such as TRAP, REDIRECT, and POLICE.

4.15.3 Source code structure

The table below shows the source files.

Table 54. ENETC switch source files

File	Description

`include/linux/fsl/enetc_mdio.h`
`include/linux/fsl/netc_*.h`
`include/linux/fsl/ntmp.h`
	Generic header file for multiple drivers to use

`include/linux/dsa/tag_netc.h`	Header file for NETC Switch tag driver
`net/dsa/tag_netc.c`	NETC Switch tag driver
`drivers/net/dsa/netc/*`	NETC Switch driver

4.15.4 Menu configuration options

The following kernel configuration options are provided for this module:

- `CONFIG_ NXP_NETC_BLK_CTRL`: Build support for NETCMIX, IERB, and PRB, which provides pre-configuration for switch. In `menuconfig`, this option is available under **Device Drivers** -> **Network device support** -> **Ethernet driver support** -> **Freescale devices** -> **NETC blocks control driver**.
- `CONFIG_ FSL_ENETC_MDIO`: Build support for EMDIO driver, which provides MDIO bus to manage the external PHYs. In `menuconfig`, this option is available under **Device Drivers** -> **Network device support** -> **Ethernet driver support** -> **Freescale devices** -> **ENETC MDIO driver**.
- `CONFIG_PTP_1588_CLOCK_NETC`: Build support for NETC 1588 Timer driver. This module is needed if the ENETC driver wants to support PTP synchronization. In `menuconfig`, this option is available under **Device Drivers** -> **PTP clock support** -> **NXP NETC Timer as PTP clock**.

- `CONFIG_NET_DSA_TAG_NETC`: Build support for NETC Switch tag driver. In `menuconfig`, this option is available under **Networking support** -> **Networking options** -> **Distributed Switch Architecture** -> **Tag driver for NXP NETC switches**.
- `CONFIG_ NET_DSA_NETC_SWITCH`: Build support for NETC Switch driver. In `menuconfig`, this option is available under **Device Drivers** -> **Network device support** -> **Distributed Switch Architecture drivers** -> **NXP NETC Ethernet switch support**.

4.16 Ethernet Controller with TSN (ENET_QoS, EQoS)

4.16.1 Introduction

The EQoS-TSN module is designed to support 10/100/1000 Mbps applications in full compliance with the IEEE 802.3-2015 specifications. It can support advanced networking capabilities, including Time-Sensitive Networking (TSN) and Audio Video Bridging (AVB). Additionally, the MAC core incorporates several features, such as a flexible receive parser, media clock recovery and generation, and safety.

The following figure shows the block diagram of this module. The module has four main blocks to perform all functions. The AHB interface is connected to all DMA channels. The DMA arbiter helps in arbitration of all paths (transmit and receive) in channels. Each channel has a separate set of control and status registers (CSR) for managing the transmit and receive functions, descriptor handling, and interrupt handling.

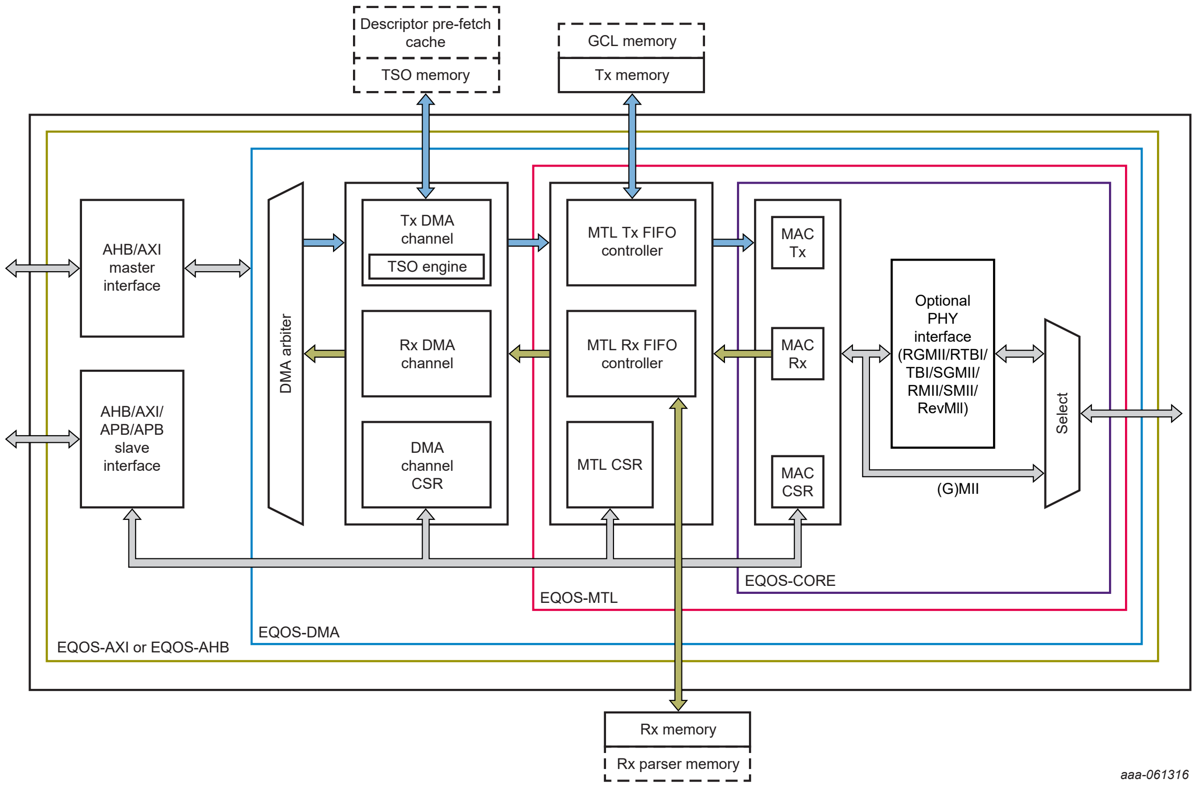

Figure 17. EQoS-TSN module

4.16.2 Software operation

The EQoS driver has the following functions:

- RMII (10/100 Mbps), RGMII (10/100/1000 Mbps)

- Time aware shaper (IEEE802.1Qbv), Time synchronization (IEEE1588-2008), and Frame pre-emption (IEEE802.1Qbu) for Time-Sensitive Networking (TSN)
- Media clock recovery and generation for AVB
- Full-duplex flow control operations (IEEE 802.3x pause packet and priority flow control)
- Flexibility to control the Pulse per second (PPS) output signal
- MDIO (clause 22 and Clause 45) interface for configuration and management of PHY device
- VLAN insertion, replacement, and deletion in transmitted packets

4.16.3 Source code structure

The table below shows the source files.

Table 55. EQoS source files

File	Description
`drivers/net/ethernet/stmicro/stmmac/stmmac_*.c`	The common logic part driven by different versions of EQoS.
`drivers/net/ethernet/stmicro/stmmac/dwmac4_*.c` `drivers/net/ethernet/stmicro/stmmac/dwmac5_*.c`	The implementation of register read and write operations based on various versions of hardware for a shared API.
`drivers/net/ethernet/stmicro/stmmac/dwmac-imx.c`	Special handling related to i.MX integration/specific functions, as well as controller probing.

4.16.4 Menu configuration options

For the EQoS controller on the i.MX platform, it is controlled by `CONFIG_STMMAC_ETH`.

Note: *In NXP's release,* `CONFIG_STMMAC_ETH` *has been set to be built-in by default. However, in the Linux upstream, this CONFIG is built as a module by default. This may affect the NFS boot.*

4.17 Wi-Fi

4.17.1 Introduction

Bluetooth and Wi-Fi are supported on i.MX through on-board chip solutions and external hardware. For various on-board chips and external solutions, see the Section "Connectivity for Bluetooth wireless technology and Wi-Fi" in the *i.MX Linux User's Guide* (UG10163).

4.17.2 Software operation

The NXP Wi-Fi driver module is supported on all i.MX chipsets available in the Linux BSP, starting from release 5.4.47-2.2.0. For a list of the supported Wi-Fi chipsets, see the Release Notes for each i.MX Linux BSP release.

4.17.3 Driver features

The NXP Wi-Fi driver supports the CFG80211, and NL80211 kernel interfaces. The driver supports AP mode, STA mode, and Wi-Fi direct mode.

The NXP Wi-Fi SoCs require a firmware image to be loaded on power-up/reset. The firmware images for the supported Wi-Fi SoCs are located in the following rootfs directory: `/lib/firmware/nxp`.

4.17.4 Source code structure

The NXP Onedriver source code files are available at github.com/nxp-imx.

4.17.5 Menu configuration options

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

- `CONFIG_MAC80211=y`
- `COCONFIG_NL80211_TESTMODE=y`
- `CONFIG_CFG80211_WEXT=y`
- `CONFIG_HOSTAP=y`
- `CONFIG_CFG80211_INTERNAL_REGDB=y`

4.17.6 Configuring WLAN from User Space

4.17.6.1 Connecting AP in Station Mode

The following command group is used to connect WLAN to a given SSID.

```
modprobe moal mod_para=nxp/wifi_mod_para.conf
head -n 4 /etc/wpa_supplicant.conf  > /etc/wpa_supplicant.conf.tmp
wpa_passphrase ssid password >> /etc/wpa_supplicant.conf.tmp
mv /etc/wpa_supplicant.conf  /etc/wpa_supplicant.conf.bak
mv /etc/wpa_supplicant.conf.tmp  /etc/wpa_supplicant.conf
wpa_supplicant -B -i mlan0 -c /etc/wpa_supplicant.conf -D nl80211
```

Here is an example of `wpa_supplicant.conf`:

```
ctrl_interface=/var/run/wpa_supplicant
ctrl_interface_group=0
update_config=1
network={
  ssid="NETGEAR73"
  #psk="freshbutter"
  psk=eb0376fc14ee5d1e6ce129ad54da038adab……
}
```

4.17.6.2 Obtaining an IP address

The following command is used to get an IP address for mlan0:

```
udhcpc -i mlan0
```

4.18 Low Power Serial Peripheral Interface (LPSPI) Driver

4.18.1 Introduction

LPSPI provides an efficient interface (either as a controller or peripheral) to an SPI bus, which is a synchronous serial communication interface used in embedded systems. It is typically used to perform short distance communications between microcontrollers and peripheral devices, on printed circuit boards. Typical applications include interfacing with secure digital cards and LCD displays.

4.18.2 Driver deatures

The LPSPI driver supports the following features:

- SPI communication protocol
- SPI master mode and slave mode of operation

4.18.3 Source code structure

The table below shows the source files.

Table 56. LPSPI source files

File	Description
`drivers/spi/spi-fsl-lpspi.c`	LPSPI Bus Driver for i.MX 7ULP, i.MX 8ULP, i.MX 8X, i.MX 93, i.MX 95, i.MX 943

4.18.4 Menu configuration options

Configure the kernel option to enable the module by `menuconfig`:

Device Drivers -> **SPI support** -> **Freescale i.MX LPSPI controller**

5 Graphics

5.1 Graphics Processing Unit (GPU)

5.1.1 Introduction

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

The 3D graphics processing unit (GPU3D) is an embedded engine that accelerates user-level graphics Application Programming Interface (APIs), such as OpenGL ES 1.1, OpenGL ES 2.0, and OpenGL ES 3.0 and OpenCL 1.1EP. The 2D graphics processing unit (GPU2D) is an embedded 2D graphics accelerator targeting graphical user interfaces (GUI) rendering boost. The VG graphics processing unit (GPUVG) 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 the kernel source tree, but the libraries are delivered as binary only.

Graphics Processing Unit	Hardware	Applicable Platform
3D	Mali G310	i.MX 95
2D	PXP Blitter	i.MX 93
3D	GC7000 NanoUltra31	8ULP
2D	GC520L	8ULP
3D	Vivante dual-GC7000XSVX	8QuadMax
3D	Vivante GC7000Lite	8QuadXPlus/8M Quad
3D	Vivante GC7000 Nano Ultra	7ULP and 8M Mini
3D	Vivante GC7000 UltraLite	8M Plus
3D	Vivante GC7000 Ultra Lite	8M Nano
3D	Vivante GC2000	6Quad/6Dual

Graphics Processing Unit	Hardware	Applicable Platform
3D	Vivante GC2000+	6QuadPlus/6DualPlus
3D	Vivante GC880	6DualLite/6Solo
3D/2D	Vivante GC400T	6SoloX
2D	Vivante GC320	6Quad/6Dual/6DualLite/6Solo
Vector	Vivante GC355	6Quad/6Dual
2D	Vivante GC328	7ULP

Note:

- *GC400T does not support OpenGL ES 3.0.*
- *GC880/GC400T does not support OpenCL 1.1EP. GC2000 and GC2000+ support OpenCL 1.1 EP.*
- *GC7000XSVX supports OpenCL 1.2 FP, OpenVX 1.0.1, and Vulkan 1.0.*

5.1.2 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) 1.5 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.
- OpenGL ES 3.0/3.1/3.2 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 the thermal driver is active, the 3D core runs at 1/64 clock.
- OpenCL1.1/1.2FP API defined by Khronos Group.
- OpenVX 1.0.1 API defined by Khronos Group.
- Vulkan 1.0 API defined by Khronos Group.

5.1.3 Hardware operation

For detailed hardware operations, see the GPU chapters in the Applications Processor Reference Manual specific to SoC.

5.1.4 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, 2.0, and 3.0 API
- EGL 1.5 API

- OpenGL ES11/20/30/31/32
- OpenCL 1.1/1.2 FP
- OpenVX 1.0.1
- Vulkan 1.0
- OpenGL 4.0
- WebGL 1.0.2
- OpenVG 1.1 API
- OpenCL 1.1 EP API

5.1.5 Source code structure

The table below lists the GPU driver kernel module source structure: `drivers/mxc/gpu-viv`.

Table 57. GPU driver files

File	Description
`Kconfig Kbuild config`	Kernel configuration file and makefile
`hal/kernel/arch`	Hardware-specific driver code for GC2000, GC880, GC400T, and GC320
`hal/kernel/archvg`	Hardware-specific driver code for GC355
`hal/kernel`	Kernel mode HAL driver
`hal/os/linux/kernel`	OS layer HAL driver

Note:

If you replace the whole content in this directory, the GPU kernel driver can be upgraded.

5.1.6 Library structure

The table below lists the GPU driver user mode library structure: `<ROOTFS>/usr/lib`.

Table 58. GPU library files

File	Description
`libCLC.so`	OpenCL front-end 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`	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 2.0/3.0/3.1/3.2 library
`libGLSLC.so`	OpenGL ES shader language compiler library

Table 58. GPU library files *...continued*

File	Description
`libVSC.so`	OpenGL front-end compiler library
`libVivanteOpenCL.so`	Vivante
`libOpenCL.so`	OpenCL ICD wrapper library
`libOpenVG.so`	OpenVG 1.1 library
`libVDK.so`	VDK wrapper library
`libVIVANTE.so`	Vivante user mode driver
`xorg/modules/drivers/vivante_drv.so`	EXA library for X11 acceleration
`libwayland-viv.so`	Wayland server-side library for Vivante's EGL driver
`libgc_wayland_protocol.so`	Vivante Wayland Protocol Extension Library
`libOpenVX.so`	OpenVX 1.0 library
`libvulkan.so`	Vulkan 1.0 library

SONAME is used for `libEGL.so`, `libGLESv2.so`, `libGLESv1_CM.so`, `libGLESv1_CL.so`, and `libGL.so`.

For `libOpenVG.so`, there are two libraries for the OpenVG feature. `libOpenVG.3d.so` is the GC7000XSVX/GC2000+/GC2000/GC880/GC400T-based OpenVG library. `libOpenVG.2d.so` is the GC355-based OpenVG library.

- For i.MX 6DualPlus/QuadPlus and i.MX 6Dual/Quad, both `libOpenVG.3d.so` and `libOpenVG.2d.so` can be used.
- For i.MX 6DualLite and i.MX 6SoloX, only `libOpenVG.3d.so` can be used.
- If there is no SOC limitation, for the x11 backend, `libOpenVG.3d.so` is linked by default.
- If there is no SOC limitation, for frame buffer, directFB, and Wayland backends, the default openVG library is linked to `libOpenVG.2d.so`.

This can be done by using the following sequence of commands:

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

5.1.7 API references

See the following websites for detailed specifications:

- OpenGL ES 1.1, 2.0, and 3.0 API: www.khronos.org/opengles/
- OpenCL 1.1 EP www.khronos.org/opencl/
- EGL 1.4 API: www.khronos.org/egl/
- OpenVG 1.1 API: www.khronos.org/openvg/
- OpenGL ES API: www.khronos.org/opengles/
- OpenCL API: www.khronos.org/opencl/
- OpenVX API: www.khronos.org/openvx/
- Vulkan API: www.khronos.org/vulkan/
- OpenGL API: www.khronos.org/opengl/
- WebGL API: www.khronos.org/webgl/

5.1.8 Menu configuration options

In menu configuration, enable the following modules for the GPU driver:

CONFIG_MXC_GPU_VIV is a configuration option for the GPU driver. In `menuconfig`, this option is available under **Device Drivers** -> **MXC support drivers** -> **MXC Vivante GPU support** -> **MXC Vivante GPU support**.

On the screen displayed, select **Configure the kernel**, and 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** -> **imx-gpu-viv**.

This package provides proprietary binary libraries and test code built from the GPU for the frame buffer.

5.2 Wayland

5.2.1 Introduction

Wayland is a protocol for a compositor to talk to its clients and a C library implementation of that protocol. The compositor can be a standalone display server running on Linux kernel mode setting and evdev input devices, an X application, or a Wayland client itself. The clients can be traditional applications, X servers, or other display servers.

Part of the Wayland project is also the Weston reference implementation of a Wayland compositor. The Weston compositor is a minimal and fast compositor and is suitable for many embedded and mobile use cases.

This section describes how to enable Wayland/Weston support on an i.MX series device.

5.2.2 Software operation

This release is based on the Wayland 1.23.1 and Weston 14.0.2 for i.MX 8 and i.MX 9, Weston 10.0.5 for i.MX 6 and i.MX 7.

5.2.3 Yocto build instructions

The instructions for Yocto Project build are as follows:

1. Prepare a Yocto build directory and follow the setup instructions in the *i.MX Yocto Project User's Guide* (UG10164) for DISTRO Wayland.
2. Set up Yocto for Wayland in the build directory:
   ```
   $ MACHINE = <your-machine> DISTRO=fsl-imx-xwayland source imx-setup-
   release.sh -b build-wayland
   ```
3. Build an image.
   ```
   $ bitbake imx-image-full
   ```

5.2.4 Customizing Weston

The i.MX Weston includes two compositors. One is the EGL3D compositor, which is accelerated by the 3D core. The other is the G2D compositor accelerated by the 2D BLT engines.

Weston options can be updated in the file `/etc/init.d/weston`.

Table 59. Common options for Weston

Weston option	Description
`tty`	Default to current `tty`.

Table 59. Common options for Weston*...continued*

Weston option	Description
`device`	`/dev/fb0`, default frame buffer, Multi display supported in G2D compositor.
`use-gl`	EGL accelerated, defaults to be **1**.
`use-g2d`	G2D accelerated, defaults to be **0**.
`idle-time`	Idle time in seconds.

5.2.4.1 Multi-display supported in Weston

Multi-display is supported in the G2D compositor only. Add these options to start Weston:

```
weston --tty=1 --device=/dev/fb0,/dev/fb2 --use-g2d=1 &
```

5.2.4.2 Multi-buffer supported in Weston

The Weston server supports both single buffering and multi-buffering. In single buffering, the damage area is rendered to the offscreen surface and blits to the front buffer. The offscreen surface is used to avoid flickering. By default, the Weston server starts with single buffering.

In multi-buffering, instead of rendering to offscreen, the damage area is rendered to the back buffer and does the flip, but the frame rate is restricted to the display rate. A maximum of three buffers are supported.

Before starting the Weston server, export `FB_MULTI_BUFFER` to control the number of buffers to be used.

Environment variables for single buffering:

```
export FB_MULTI_BUFFER=1
```

Environment variables for double buffering:

```
export FB_MULTI_BUFFER=2
```

5.2.5 Running Weston

Perform the following operations to run Weston:

1. Boot the i.MX device.
2. To run clients, the second button in the top bar runs weston-terminal, from which you can run clients. There are a few demo clients available in the Weston build directory, but they are simple and mostly for testing specific features in the Wayland protocol:
   - `weston-terminal` is a simple terminal emulator, not very compliant, but works well enough for bash.
   - `weston-flower` draws a flower on the screen, testing the frame protocol.
   - `weston-smoke` tests SHM buffer sharing.
   - `weston-image` loads the image files passed on the command line and shows them.

6 Video

6.1 Capture overview

6.1.1 Introduction

The i.MX capture driver is supported through the V4L2 interface with camera sensor controllers and 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 the camera, and starting the camera preview.

6.1.2 Capture controllers and interfaces

The list of capture controllers is as follows:

- Camera Serial Interface (CSI)
- IPU-CSI
- Video Interface Unit (VIU)
- Image Sensor Interface (ISI)
- Image Sensor Processing (ISP)

The list of capture interfaces for transferring image data is as follows:

- Parallel-CSI
- MIPI-CSI2
- HDMI RX
- TV Decoder

This section describes the differences between the various controllers and interfaces.

Note: *The i.MX 6 with IPU uses* `internaldev` *for V4L2 interface while all others use* `subdev` *for V4L2 interface.*

The following table lists the different controllers and interfaces combinations.

Table 60. Camera controllers and interfaces

SoC	Controller	Interface
6SLL	CSI	Parallel CSI.
6SoloX	VIU	Parallel CSI and TV Decoder.
6UltraLite/6ULL	CSI	Parallel CSI.
6DualLite/Solo	IPU-CSI	Parallel CSI internaldev IPU.
6QuadPlus/Quad/Dual	IPU-CSI	Parallel CSI internaldev IPU.
7Dual/Solo	CSI	MIPI-CSI2 using Samsung and Parallel CSI.
8M Plus/8M Nano	ISI	MIPI-CSI2 using Samsung.
8M Plus	ISP	MIPI-CSI2 using Samsung.
8QuadMax	ISI	MIPI-CSI2 using Mixel and HDMI Receiver using Cadence.
8QuadXPlus	ISI	MIPI-CSI2 using Mixel and Parallel CSI using i.MX 8.
8M Quad	CSI	MIPI-CSI2 using Mixel.
8M Mini	CSI	MIPI-CSI2 using Samsung.

Table 60. Camera controllers and interfaces*...continued*

SoC	Controller	Interface
8ULP	ISI	MIPI-CSI2 using Mixel.
i.MX 93 and i.MX 91	ISI	i.MX 93 supports MIPI CSI and Parallel camera interface, but i.MX 91 only supports Parallel interface.
i.MX 95	ISI and NEO ISP	MIPI-CSI2 using Synopsys.

Some additional details are listed below:

- The NEO ISP is a new controller used for the i.MX 95.
- The ISP is a new controller used for the i.MX 8M Plus.
- The ISI controller is a new controller used for some of the i.MX 8 series.
- The i.MX 6 SoC without IPU, i.MX 7Dual and i.MX 8M use the same CSI controller driver.
- The i.MX 8 and i.MX 8X families use a newer i.MX 8 CSI driver.
- The i.MX 6 with IPU use a customized CSI that interfaces with IPU hardware.
- Each SoC can support one or more interfaces as described in the previous table. The interfaces align with Video for Linux V4L2 APIs.
- In some cases, the capture controller is not interfacing to a camera but a video input unit. Some also connect to HDMI Receivers or TV Decoders.

6.1.2.1 IPUv3

The Image Processing Unit version 3 (IPUv3) controller integrates the capture DMA Controller and the capture interface as well as the display controller and interface. The following features are supported on SoCs that use the IPUv3 controller.

- Parallel CSI with 2 Ports, 20 bits+ 8 bits
- Playback 1080i/p + D1 @ 30fps @ 30fps
- Record 1080p @ 30fps
- 2-way 720 @ 30fps
- De-interlacing high-quality motion adaptive algorithm
- Resizing: fully flexible
- Rotation/inversion support
- Color conversion: fully flexible
- Memory interface: AXI split transaction 64-bit 266 MHz
- Memory bus: selective read for combining
- Control capabilities : display and DMA controller, internal synchronization
- Synchronization: double/triple buffering, frame-by-frame, or tight sub-frame with internal memory

For i.MX 6 with IPU kernel configuration, use the following configurations:

- **Device Drivers** -> **Multimedia support** (`MEDIA_SUPPORT=y`]) -> **Media drivers** -> **Media platform devices** (`MEDIA_PLATFORM_DRIVERS [=y]`) -> **MXC Video For Linux Video Capture** (`VIDEO_MXC_CAPTURE [=m]`) -> **MXC Camera/V4L2 PRP Features support** -> **OmniVision OV5640 camera support** (`MXC_CAMERA_OV5640_V2 [=m]`)
- **Device Drivers** -> **Multimedia support** (`MEDIA_SUPPORT [=y]`) -> **Media drivers** -> **Media platform devices** (`MEDIA_PLATFORM_DRIVERS [=y]`) -> **MXC Video For Linux Video Capture** (`VIDEO_MXC_CAPTURE [=m]`) -> **MXC Camera/V4L2 PRP Features support** -> **OmniVision OV5640 camera support using MIPI** (`MXC_CAMERA_OV5640_MIPI_V2 [=m]`)

These are driver files for the IPUv3:

- `drivers/media/platform/mxc/capture/ipu_bg_overlay_sdc.c`

- `drivers/media/platform/mxc/capture/ipu_csi_enc.c`
- `drivers/media/platform/mxc/capture/ipu_fg_overlay_sdc.c`
- `drivers/media/platform/mxc/capture/ipu_prp_enc.c`
- `drivers/media/platform/mxc/capture/ipu_prp_vf_sdc_bg.c`
- `drivers/media/platform/mxc/capture/ipu_prp_vf_sdc.c`
- `drivers/media/platform/mxc/capture/ipu_still.c`
- `drivers/media/platform/mxc/capture/v4l2-int-device`

6.1.2.2 CSI controllers

- MIPI-CSI-2 with 2 ports, 4 lanes x 1.5 Gbps (2.5 Gbps on i.MX 95)
- Playback: two 1080p30
- Record: 1080p30x2
- 2-way: 1080p30x2
- Deinterlacing: simple bob and weave
- Memory interface throughput: 64-bit
- Controller capabilities: DMA
- Synchronization: double buffer

6.1.2.3 ISI

The Image Sensor Interface (ISI) handles the DMA and image formatting operations for most of the i.MX 8 and all of the i.MX 9 SoCs. The ISI uses a pixel link to attach to the capture interfaces. The number and type of capture interfaces depend on the SoC. The ISI has the following common features:

- Deinterlacing: simple bob and weave
- Resizing
- Controller capabilities: DMA
- Synchronization: double buffer

SoC specific features are as follows:

i.MX 8QuadXPlus:

- Parallel CSI: one port 24 bits
- MIPI-CSI-2 with 1 port, 4 lanes x 1.5 Gbps
- Playback: 1080p30x2
- Record: 1080p30x2
- 2-way: 1080p30x2
- Memory interface throughput: 128-bit, 400 MHz

i.MX 8QuadMax:

- MIPI-CSI-2 with 2 ports, 4 lanes x 1.5 Gbps
- HDMI receiver: 1 port HDMI 1.4 4K30
- Playback: 4K60x1, 4K30x1, or 1080px30x4
- Record: 4K30x1 or 1080px30x4
- 2-way: 4K30x1 or 1080px30x4
- Memory interface throughput: 128-bit, 400 MHz

i.MX 8M Plus:

- MIPI-CSI-2 with 2 ports, 4 Lanes x 1.5 Gbps
- Supports one source of 4K resolution at 30 fps (24 bpp)

- Supports two sources up to 2K resolution at 60 fps (24 bpp) on each channel

i.MX 8M Nano

- MIPI-CSI-2 with 1 port, 4 Lanes x 1.5 Gbps
- Supports one source up to 2K resolution at 60 fps (24 bpp)

i.MX 93/91

- MIPI-CSI-2 with 1 port, 2 lanes
- Supports one source up to 2K resolution at 60 fps (24 bpp)

i.MX MX 95

- MIPI-CSI-2 with 2 ports, 4 Lanes x 2.5 Gbps
- Supports one source of 4K resolution at 60 fps (24 bpp)
- Supports two sources up to 2K resolution at 60 fps (24 bpp) on each channel
- Supports eight sources up to 2K resolution at 30 fps (24 bpp) on each channel

For i.MX 8M Nano, 8ULP, 8QuadXPlus, 8QuadMax, 8DualX, and i.MX 93/91/95, use the following kernel configurations and driver files:

- **Device Drivers** -> **Multimedia support** (`MEDIA_SUPPORT [=y]`) -> **Media drivers** -> **Media platform devices** (`MEDIA_PLATFORM_DRIVERS [=y]`) -> **i.MX 8 Image Sensor Interface (ISI) driver** (`VIDEO_IMX8_ISI [=y]`)
  Select: `MEDIA_CONTROLLER [=y] && V4L2_FWNODE [=y] && V4L2_MEM2MEM_DEV [=y] && VIDEO_V4L2_SUBDEV_API [=y] && VIDEOBUF2_DMA_CONTIG [=y]`
  – `imx8-isi-core.c`
  – `imx8-isi-crossbar.c`
  – `imx8-isi-gasket.c`
  – `imx8-isi-hw.c`
  – `imx8-isi-pipe.c`
  – `imx8-isi-video.c`
  – `imx8-isi-m2m.c`
- **Device Drivers** -> **Multimedia support** (`MEDIA_SUPPORT [=y]`) -> **Media drivers** -> **Media platform devices** (`MEDIA_PLATFORM_DRIVERS [=y]`) -> **NXP i.MX 95 CSI Pixel Formatter Driver** (`VIDEO_IMX_CSI_FORMATTER [=y]`)
  Select: `MEDIA_CONTROLLER [=y] && V4L2_FWNODE [=y] && VIDEO_V4L2_SUBDEV_API [=y]`
  – `imx-csi-formatter.c`

For all the other platforms, use the following kernel configurations and driver files:

- **Device Drivers** -> **Staging drivers** (`STAGING [=y]`) -> **Media staging drivers** (`STAGING_MEDIA [=y]`) -> **i.MX V4L2 media core driver** (`VIDEO_IMX_CAPTURE [=y]`) -> **i.MX 8QXP/QM Camera ISI/MIPI Features support** -> **i.MX 8 Image Sensor Interface hardware driver** (`IMX8_ISI_HW [=y]`)
  – `imx8-isi-hw.c`
- **Device Drivers** -> **Staging drivers** (`STAGING [=y]`) -> **Media staging drivers** (`STAGING_MEDIA [=y]`) -> **i.MX V4L2 media core driver** (`VIDEO_IMX_CAPTURE [=y]`) -> **i.MX 8QXP/QM Camera ISI/MIPI Features support** -> **i.MX 8 Image Sensor Interface Core Driver** (`IMX8_ISI_CORE [=y]`)
  – `imx8-isi-core.c`
- **Device Drivers** -> **Staging drivers** (`STAGING [=y]`) -> **Media staging drivers** (`STAGING_MEDIA [=y]`) -> **i.MX V4L2 media core driver** (`VIDEO_IMX_CAPTURE [=y]`) -> **i.MX 8QXP/QM Camera ISI/MIPI Features support** -> **i.MX 8 Image Sensor Interface Capture Device Driver** (`IMX8_ISI_CAPTURE [=y]`)
  – `imx8-isi-cap.c`

  - `imx8-isi-fmt.c`
- **Device Drivers** -> **Staging drivers** (`STAGING [=y]`) -> **Media staging drivers** (`STAGING_MEDIA [=y]`) -> **i.MX V4L2 media core driver** (`VIDEO_IMX_CAPTURE [=y]`) -> **i.MX 8QXP/QM Camera ISI/MIPI Features support** -> **i.MX 8 Image Sensor Interface Memory to Memory Device Driver** (`IMX8_ISI_M2M [=y]`)
  - `imx8-isi-m2m.c`
  - `imx8-isi-fmt.c`

6.1.2.4 Parallel CSI interface

The Parallel CSI interface driver enables a direct connection to external CMOS sensors and CCIR656 video sources. The CSI and sensor drivers are implemented in the Video for Linux Two (V4L2) driver framework consisting of the image capture driver and the video output driver.

The driver initializes the CSI interface and configures and operates with the hardware registers for the CSI module. The following features are supported:

- Configurable interface logic to support the most commonly available CMOS sensors.
- Full control of 8-bit/pixel, 10-bit/pixel, or 16-bit/pixel data format to 32-bit receive FIFO packing.
- 128x32 FIFO to store received image pixel data.
- Receive FIFO overrun protection mechanism.
- Embedded DMA controllers to transfer data from receive FIFO or statistic FIFO through AHB bus.
- Support for double buffering two frames in the external memory.
- Single interrupt source to interrupt controller from maskable interrupt sources: Start of Frame, End of Frame, and so on.
- Configurable master clock frequency output to sensor.

The V4L2 CSI capture device includes two interfaces: the capture and overlay interfaces. The capture and overlay interface use the CSI embedded DMA controller to implement the function using V4L2 APIs. The following is the data flow of capture and overlay.

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

i.MX 6 with IPU use an IPU-CSI driver that interfaces with the IPU directly. i.MX Quad Plus/Quad/Dual have support for two IPU-CSI scenarios. i.MX 6 without IPU and i.MX 7Dual/Solo and i.MX 93 9x9 QSB use a separate CSI sensor driver that interfaces directly to the sensor.

6.1.2.5 MIPI Camera Serial Interface (MIPI CSI)

There are four blocks in the MIPI CSI-2 D-PHY: PHY adaptation layer, packet analyzer, image data interface, and register bank. MIPI CSI-2 is a MIPI-Camera Serial Interface Host Controller with a high-performance serial interconnect bus for mobile applications, 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 the MIPI CSI-2-compliant Camera Sensor.

The MIPI CSI-2 driver is used to manage the MIPI D-PHY and lets it work with both MIPI sensor and IPU CSI. MIPI CSI-2 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, the MIPI sensor can set certain information about the sensor (such as datatype, lanes number, etc.) to the MIPI CSI-2 driver to configure D-PHY. 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 CSI-2 driver. Functions and operations are listed as follows:

- PHY Adaptation Layer handles managing the D-PHY interface including PHY error handling.
- Packet Analyzer handles data lane merging if needed, 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.

The MIPI CSI-2 driver for Linux OS has two parts:

- MIPI CSI-2 driver initialization operation, which initializes `mipi_csi2_info struct`.
- MIPI CSI-2 common APIs, which exports APIs for the CSI module driver and MIPI sensor driver.

6.1.2.6 HDMI receiver

The HDMI receiver allows capturing video with the Image Sensor Interface (ISI) from the HDMI RX. On i.MX 8QuadMax, the HDMI receiver video interface supports one port up to HDMI 1.4 4K30.

i.MX 8 QuadMax uses the following kernel configurations and driver files:

- **Device Drivers** -> **Multimedia support** (`MEDIA_SUPPORT [=y]`) -> **Media drivers** -> **Media platform devices** (`MEDIA_PLATFORM_DRIVERS [=y]`) -> **Cadence MHDP HDMIRX Controller** (`VIDEO_MHDP_HDMIRX [=m]`)
  - `cdns-hdmirx.c`
  - `cdns-hdmirx-hdcp.c`
  - `cdns-hdmirx-hw.c`
  - `cdns-hdmirx-phy.c`
  - `cdns-mhdp-hdmirx.c`

Execute the following commands on U-Boot to enable the HDMI RX driver:

```
U-Boot > setenv fdt_file imx8qm-mek-hdmi-rx.dtb
U-Boot > setenv hdprx_enable yes
U-Boot > saveenv
```

The following example is for creating a pipeline manually. The HDMI source must be configured to use 1080p60 in the RGB888 format.

```
$ media-ctl -d1 -r
$ media-ctl -d1 -R '"crossbar" [4/0 -> 6/0 [1]]'
$ media-ctl -d1 -l '"crossbar":6 -> "mxc_isi.0":0 [1]'
$ media-ctl -d1 -l '"mxc_isi.0":1 -> "mxc_isi.0.capture":0 [1]'
$ media-ctl -d1 -V '"crossbar":4 [fmt:RGB888_1X24/1920x1080 field:none]'
$ media-ctl -d1 -V '"crossbar":6 [fmt:RGB888_1X24/1920x1080 field:none]'
$ media-ctl -d1 -V '"mxc_isi.0":0 [compose:(0,0)/1920x1080]'
$ media-ctl -d1 -V '"mxc_isi.0":1 [fmt:RGB888_1X24/1920x1080 field:none]'
```

Create a GStreamer pipline to display the captured video.

```
$ gst-launch-1.0 -v v4l2src device=/dev/video2 '!' video/x-
raw,width=1920,height=1080,format=BGRA '!' waylandsink
```

6.1.3 Cameras

The i.MX Camera accessories support multiple interfaces and types of cameras. There is a wide range of lens and sensor resolutions.

The Cameras uses the serial camera control bus (SCCB) interface to control the sensor operation working as an I2C client for control operations. These cameras support transfer modes of CSI, MIPI-CSI2, and Parallel-CSI interfaces. When using MIPI mode, the camera connects to the i.MX chip through the MIPI CSI-2 interface. The MIPI receives the sensor data and transfers it to CSI. Additionally, a few of the MIPI CSI-2 cameras use serializer and de-serializer to extend the physical length on the camera connection from a few inches to several feet.

The following table lists the different i.MX Camera Accessories, supported interfaces, and pixel format.

Table 61. i.MX Camera Accessories

Camera	Controller Interface	Format
OV5640	MIPI-CSI2/Parallel CSI	YUV
OV10635	MIPI-CSI2 via Serializer/Deserializer	YUV
OS08A20	MIPI CSI2	Raw
OX05B1S	MIPI CSI2	Raw + Infrared
OX03C10	MIPI-CSI2 via Serializer/Deserializer	Raw
AP1302/AR0144	MIPI CSI2	YUV

6.1.3.1 YUV/RGB cameras

The following sections describe the kernel configurations and list the driver file locations for each of the camera drivers.

6.1.3.1.1 OV5640

The Omnivision OV5640 is used on multiple versions of i.MX SoCs. There are 4 NXP versions of the OV5640 driver in addition to the upstream kernel driver. Top-level selection is as follows:

Device Drivers -> **Multimedia support** (`MEDIA_SUPPORT [=y]`) -> **V4L platform devices**.

The next-level selections select the driver based on the different interfaces for each SoC. The source files for each driver are as follows.

- For i.MX 6 with IPU, select both **MXC Camera/V4L2 PRP Features support** and **OmniVision OV5640 camera support** (`MXC_CAMERA_OV5640`).
  – `drivers/media/platform/mxc/capture/ov5640.c`
- For i.MX 6 without IPU, select **OmniVision OV5640 camera support** (`MXC_CAMERA_OV5640_V2`).
  – `drivers/media/platform/mxc/capture/ov5640_v2.c`
- For i.MX 7, select **OmniVision OV5640 camera support using MIPI** (`MXC_CAMERA_OV5640_MIPI_V2`).
  – `drivers/media/platform/mxc/capture/ov5640_mipi_v2.c`
- For i.MX 8, i.MX 8X, and i.MX 8M, select **Media ancillary drivers** -> **Camera sensor devices** -> **OmniVision OV5640 sensor support** (`VIDEO_OV5640`).

Note: *This is the upstream driver version for OV5640.*

– `drivers/media/i2c/ov5640.c`

The following are examples for creating a pipeline manually.

- On i.MX 8M Quad with dual OV5640, input size: 640x480, output size: 640x480, output format: YUYV

```
media-ctl -d /dev/media0 -l "'ov5640 1-003c':0 -> 'imx8mq-mipi-csi2
 30a70000.csi':0 [1]"
media-ctl -d /dev/media0 -V "'ov5640 1-003c':0 [fmt:YUYV8_1X16/640x480
 field:none]"
media-ctl -d /dev/media0 -V "'imx8mq-mipi-csi2 30a70000.csi':0
 [fmt:YUYV8_1X16/640x480 field:none]"
media-ctl -d /dev/media0 -V "'csi':0 [fmt:YUYV8_1X16/640x480 field:none]";

media-ctl -d /dev/media1 -l "'ov5640 0-003c':0 -> 'imx8mq-mipi-csi2
 30b60000.csi':0 [1]"
media-ctl -d /dev/media1 -V "'ov5640 0-003c':0 [fmt:YUYV8_1X16/640x480
 field:none]"
media-ctl -d /dev/media1 -V "'imx8mq-mipi-csi2 30b60000.csi':0
 [fmt:YUYV8_1X16/640x480 field:none]"
media-ctl -d /dev/media1 -V "'csi':0 [fmt:YUYV8_1X16/640x480 field:none]";
```

- On i.MX 8M Mini, input size: 1920x1080, output size: 1920x1080, output format: YUYV

```
media-ctl -l "'ov5640 2-003c':0 -> 'csis-32e30000.mipi-csi':0 [1]"
media-ctl -V "'ov5640 2-003c':0 [fmt:YUYV8_1X16/1920x1080 field:none]"
media-ctl -V "'csis-32e30000.mipi-csi':0 [fmt:YUYV8_1X16/1920x1080 field:none]"
media-ctl -V "'csi':0 [fmt:YUYV8_1X16/1920x1080 field:none]";
```

- On i.MX 8M Nano, input size: 1920x1080, output size: 1920x1080, out format: UYVY

```
media-ctl -l "'ov5640 2-003c':0 -> 'csis-32e30000.mipi-csi':0 [1]"
media-ctl -V "'ov5640 2-003c':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
media-ctl -V "'csis-32e30000.mipi-csi':0 [fmt: UYVY8_1X16/1920x1080
 field:none]"
media-ctl -V "'crossbar':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
media-ctl -V "'mxc_isi.0':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
```

- On i.MX 8ULP, input size: 1920x1080, output size: 1920x1080, out format: UYVY

```
media-ctl -l "'ov5640 0-003c':0 -> 'imx8mq-mipi-csi2 2daf0000.csi':0 [1]"
media-ctl -V "'ov5640 0-003c':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
media-ctl -V "'imx8mq-mipi-csi2 2daf0000.csi':0 [fmt: UYVY8_1X16/1920x1080
 field:none]"
media-ctl -V "'crossbar':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
media-ctl -V "'mxc_isi.0':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
```

6.1.3.1.2 OV10635

The Omnivison OV10635 camera is supported on i.MX 8QuadX/QuadXPlus. The camera is connected through a Maxim MAX9271 Serializer and a Maxim MAX9286 Deserializer. Up to four OV10635 cameras may be connected simultaneously per MIPI CSI2 port.

- To select the kernel configuration, **Device Drivers** -> **Multimedia support** -> **Media ancillary drivers** -> **Video decoders** -> **Maxim MAX9286 GMSL deserializer support (VIDEO_MAX9286) and Device Drivers** -> **Multimedia support** -> **Media ancillary drivers** -> **Camera sensor devices** -> **IMI RDACM20 camera support (VIDEO_RDACM20)**.
  – `drivers/media/i2c/max9286.c`
  – `drivers/media/i2c/rdacm20.c`

6.1.3.1.3 AP1302/AR0144

The Onsemi AR0144 camera is supported on the i.MX 9 family. The AR0144 is a raw camera, but it uses an external ISP, Onsemi AP1302, to process the raw image and provide an sRGB image.

- For i.MX 93 and i.MX 95, select **Media ancillary drivers** -> **Camera sensor devices** -> **ON Semiconductor's Advanced Image Coporcessor AP1302 support (VIDEO_AP1302)** .
  - `drivers/media/i2c/ap1302.c`

Support V4L2 controls as follows:

- `V4L2_CID_AUTO_N_PRESET_WHITE_BALANCE`
- `V4L2_CID_GAMMA`
- `V4L2_CID_CONTRAST`
- `V4L2_CID_BRIGHTNESS`
- `V4L2_CID_SATURATION`
- `V4L2_CID_EXPOSURE`
- `V4L2_CID_EXPOSURE_METERING`
- `V4L2_CID_GAIN`
- `V4L2_CID_ZOOM_ABSOLUTE`
- `V4L2_CID_COLORFX`
- `V4L2_CID_SCENE_MODE`
- `V4L2_CID_POWER_LINE_FREQUENCY`
- `V4L2_CID_LINK_FREQ`

A firmware named `ap1302_ar0144_single_fw.bin` is needed. Put it under the `/lib/firmware` folder.

Configure the pipeline manually, for example:

1. YUV color space.
   a. Set up the link:

```
$ media-ctl -l "'ap1302 2-003c':2->'csidev-4ad30000.csi':0 [1]"
$ media-ctl -l "'csidev-4ad30000.csi':1 ->
 '4ac10000.syscon:formatter@20':0 [1]"
```

   b. Set up the formats.

```
$ media-ctl -V "'ap1302 2-003c':2 [fmt: UYVY8_1X16/1920x1080 field:none]"
$ media-ctl -V "'csidev-4ad30000.csi':0 [fmt: UYVY8_1X16/1920x1080
 field:none]"
$ media-ctl -V "'4ac10000.syscon:formatter@20':0 [fmt:
 UYVY8_1X16/1920x1080 field:none]"
$ media-ctl -V "'crossbar':2 [fmt: UYVY8_1X16/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.0':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.1':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.2':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.3':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.4':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.5':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.6':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.7':0 [fmt: UYVY8_1X16/1920x1080 field:none]"
```

   c. Set the YUV colorspace output (i.MX 93 only supports one).

```
$ media-ctl -V "'mxc_isi.0':1 [fmt: YUV8_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.1':1 [fmt: YUV8_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.2':1 [fmt: YUV8_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.3':1 [fmt: YUV8_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.4':1 [fmt: YUV8_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.5':1 [fmt: YUV8_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.6':1 [fmt: YUV8_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.7':1 [fmt: YUV8_1X24/1920x1080 field:none]"
```

2. RGB color space (i.MX 93 only supports one).

```
$ media-ctl -V "'mxc_isi.0':1 [fmt: RGB888_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.1':1 [fmt: RGB888_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.2':1 [fmt: RGB888_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.3':1 [fmt: RGB888_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.4':1 [fmt: RGB888_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.5':1 [fmt: RGB888_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.6':1 [fmt: RGB888_1X24/1920x1080 field:none]"
$ media-ctl -V "'mxc_isi.7':1 [fmt: RGB888_1X24/1920x1080 field:none]"
```

3. Down scaling (resize), 1920x1080 -> 640x480 YUYV output (i.MX 93 only supports one).

```
$ media-ctl -V "'mxc_isi.0':0 [fmt:UYVY8_1X16/1920x1080 field:none compose:
(0,0)/640x480]"
$ media-ctl -V "'mxc_isi.1':0 [fmt:UYVY8_1X16/1920x1080 field:none compose:
(0,0)/640x480]"
$ media-ctl -V "'mxc_isi.2':0 [fmt:UYVY8_1X16/1920x1080 field:none compose:
(0,0)/640x480]"
$ media-ctl -V "'mxc_isi.3':0 [fmt:UYVY8_1X16/1920x1080 field:none compose:
(0,0)/640x480]"
$ media-ctl -V "'mxc_isi.4':0 [fmt:UYVY8_1X16/1920x1080 field:none compose:
(0,0)/640x480]"
$ media-ctl -V "'mxc_isi.5':0 [fmt:UYVY8_1X16/1920x1080 field:none compose:
(0,0)/640x480]"
$ media-ctl -V "'mxc_isi.6':0 [fmt:UYVY8_1X16/1920x1080 field:none compose:
(0,0)/640x480]"
$ media-ctl -V "'mxc_isi.7':0 [fmt:UYVY8_1X16/1920x1080 field:none compose:
(0,0)/640x480]"
```

4. Stream duplicated (only supported by i.MX 95), eight 1080P streams.
If the user does not change the default route table, it does not need to configure the route table.

```
$ gst-launch-1.0 v4l2src device=/dev/video0 ! video/x-
raw,width=1920,height=1080,format=YUY2 ! waylandsink window-width=480 window-
height=540 &
$ gst-launch-1.0 v4l2src device=/dev/video1 ! video/x-
raw,width=1920,height=1080,format=YUY2 ! waylandsink window-width=480 window-
height=540 &
$ gst-launch-1.0 v4l2src device=/dev/video2 ! video/x-
raw,width=1920,height=1080,format=YUY2 ! waylandsink window-width=480 window-
height=540 &
$ gst-launch-1.0 v4l2src device=/dev/video3 ! video/x-
raw,width=1920,height=1080,format=YUY2 ! waylandsink window-width=480 window-
height=540 &
$ gst-launch-1.0 v4l2src device=/dev/video4 ! video/x-
raw,width=1920,height=1080,format=YUY2 ! waylandsink window-width=480 window-
height=540 &
$ gst-launch-1.0 v4l2src device=/dev/video5 ! video/x-
raw,width=1920,height=1080,format=YUY2 ! waylandsink window-width=480 window-
height=540 &
$ gst-launch-1.0 v4l2src device=/dev/video6 ! video/x-
raw,width=1920,height=1080,format=YUY2 ! waylandsink window-width=480 window-
height=540 &
```

```
$ gst-launch-1.0 v4l2src device=/dev/video7 ! video/x-
raw,width=1920,height=1080,format=YUY2 ! waylandsink window-width=480 window-
height=540 &
```

5. Set the route table.

```
$ media-ctl -R "'crossbar' [2/0->5/0 [1], 2/0->6/0 [1], 2/0->7/0 [1], 2/0-
>8/0 [1], 2/0->9/0 [1], 2/0->10/0 [1], 2/0->11/0 [1], 2/0->12/0 [1]]"
```

6.1.3.1.4 MT9M114

The Onsemi MT9M114 is supported on i.MX 91 and i.MX 93.

- To select the kernel configuration, **Device Drivers** -> **Multimedia support** (`MEDIA_SUPPORT [=y]`) -> **Media ancillary drivers** -> **Camera sensor devices** (`VIDEO_CAMERA_SENSOR [=y]`) -> **MT9M114x support** (`VIDEO_MT9M114X [=y]`)
  – `drivers/media/i2c/mt9m114x.c`

The following is an example for creating a pipeline manually.

```
input size: 1280x720, Output size: 1280x720 out format: UYVY
media-ctl -l "'mt9m114 7-0048':0 -> 'parallel-4ac10070.pcsi':0 [1]"
media-ctl -V "'mt9m114 7-0048':0 [fmt: UYVY8_2X8/1280x720 field:none]"
media-ctl -V "'parallel-4ac10070.pcsi':0 [fmt: UYVY8_2X8/1280x720 field:none]"
media-ctl -V "'crossbar':0 [fmt: UYVY8_2X8/1280x720 field:none]"
media-ctl -V "'mxc_isi.0':0 [fmt: UYVY8_2X8/1280x720 field:none]"
```

6.1.3.2 Raw cameras

6.1.3.2.1 OX05B1S

The Omnivision OX05B1S is an 1/2.5-inch optical format, 2592x1944, stacked-chip, low power, CMOS, 2.2 µm, RGB-IR raw output sensor. It outputs data on the MIPI-CSI2 interface, up to 4 lanes, 10-bit, with a maximum image transfer rate of 1944 @ 60 fps.

The Omnivision OX05B1S sensor is supported on i.MX 95 on the MIPI-CSI port using an NXP adapter board (SCH-89961 Rev. B).

To activate the OX05B1S sensor, the following kernel modules need to be selected in the Linux kernel configuration:

- **Device Drivers** -> **Multimedia support** -> **Media ancillary drivers** -> **Video decoders** -> **OmniVision raw sensor support OX05B1S** (`CONFIG_VIDEO_OX05B1S`)
  – `drivers/media/i2c/ox05b1s/ox05b1s_mipi.c`

The formats supported for this sensor are:

- 2592 x 1944, GRBG10 @ 30 fps

The driver is V4L2 compliant, using common V4L2 controls:

- `V4L2_CID_VBLANK`
- `V4L2_CID_HBLANK`
- `V4L2_CID_PIXEL_RATE`
- `V4L2_CID_ANALOGUE_GAIN`
- `V4L2_CID_EXPOSURE`

The media controller graph for OX05B1S may look like as shown in the following figure.

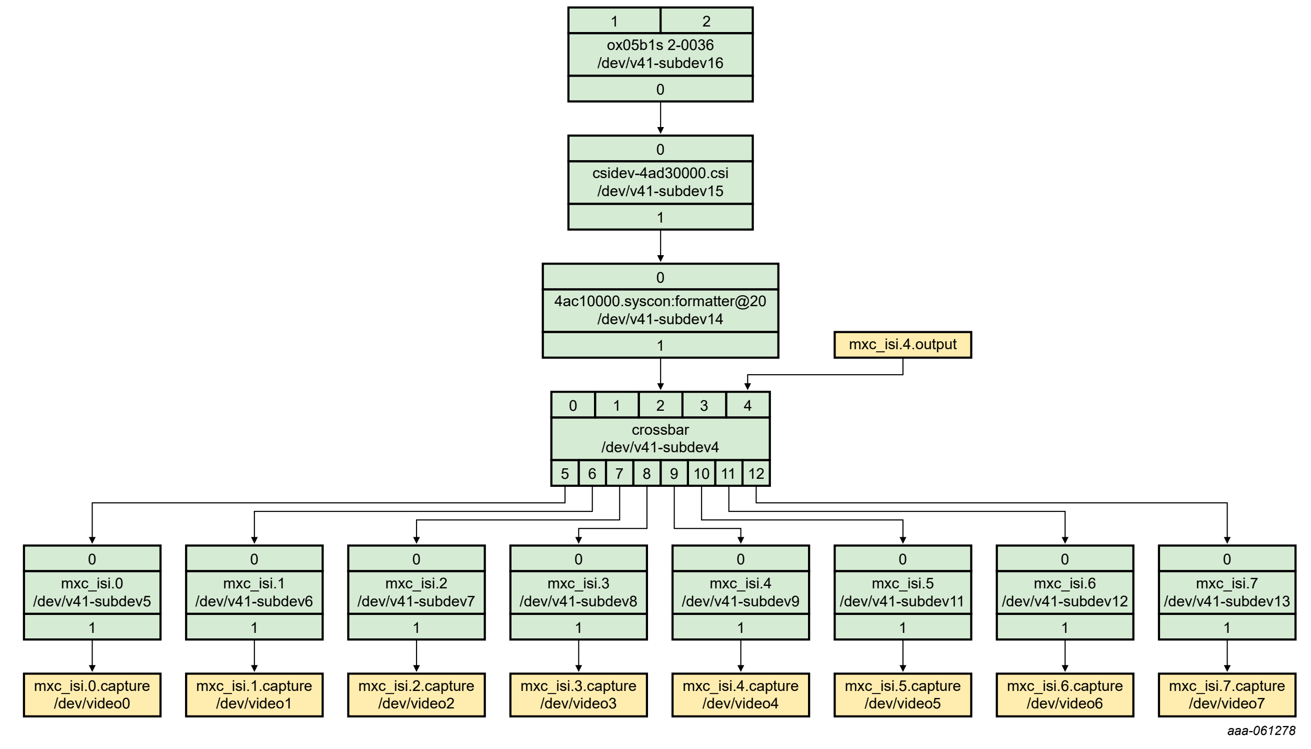

Figure 18. Media controller graph for OX05B1S

The pipeline may be configured manually like this, for example, for SGRBG10/2592x1944:

- Set links:

```
media-ctl -d /dev/media1 -l "ox05b1s 2-0036":0 -> "csidev-4ad30000.csi":0[1]
media-ctl -d /dev/media1 -l "csidev-4ad30000.csi":1 ->
 "4ac10000.syscon:formatter@20":0[1]
media-ctl -d /dev/media1 -l "4ac10000.syscon:formatter@20":1 -> "crossbar":2[1]
media-ctl -d /dev/media1 -l "crossbar":5 -> "mxc_isi.0":0[1]
```

- Set routes:

```
media-ctl -d /dev/media1 -R "crossbar" [2/0 -> 5/0 [1], 2/1 -> 6/0 [0], 2/2 ->
 7/0 [0], 2/3 -> 8/0 [0]]
media-ctl -d /dev/media1 -R "csidev-4ad30000.csi" [0/0 -> 1/0 [1], 0/1 -> 1/1
 [0], 0/2 -> 1/2 [0], 0/3 -> 1/3 [0]]
media-ctl -d /dev/media1 -R "4ac10000.syscon:formatter@20" [0/0 -> 1/0 [1], 0/1
 -> 1/1 [0], 0/2 -> 1/2 [0], 0/3 -> 1/3 [0]]
```

- Set formats:

```
media-ctl -d /dev/media1 -V "ox05b1s 2-0036":0/0 [fmt:SGRBG10/2592x1944
 field:none]
media-ctl -d /dev/media1 -V "csidev-4ad30000.csi":0/0 [fmt:SGRBG10/2592x1944
 field:none]
media-ctl -d /dev/media1 -V "4ac10000.syscon:formatter@20":0/0
 [fmt:SGRBG10/2592x1944 field:none]
media-ctl -d /dev/media1 -V "crossbar":2/0 [fmt:SGRBG10/2592x1944 field:none]
media-ctl -d /dev/media1 -V "mxc_isi.0":0/0 [fmt:SGRBG10/2592x1944 field:none]
```

- Frames then may be captured from the `/dev/video0` node:

```
v4l2-ctl --device /dev/video0 --set-fmt-
video=width=2592,height=1944,pixelformat=BA10 --stream-mmap --stream-count=5
```

6.1.3.2.2 OS08A20

The Omnivision OS08A20 is an 1/1.8-inch optical format, 3840 x 2160, CMOS, 2.2 µm, raw RGB sensor. The OS08A20 has an 8-megapixel array image capable of operating at up to 60 fps in 10-bit resolution. It outputs data on the MIPI-CSI2 interface, up to 4 lanes, 10/12-bit.

The Omnivision OS08A20 sensor is supported on i.MX 95 on the MIPI-CSI port using an NXP adapter board (SCH-89961 Rev. B), same as the OX05B1S. The kernel module is also the same as for OX05B1S, so follow the same steps to activate the OS08A20 sensor:

- **Device Drivers** -> **Multimedia support** -> **Media ancillary drivers** -> **Video decoders** -> **OmniVision raw sensor support OX05B1S** (`CONFIG_VIDEO_OX05B1S`)
  - `drivers/media/i2c/ox05b1s/ox05b1s_mipi.c`

The formats supported for this sensor are:

- 1920 x 1080, BGGR10, no HDR @ 60 fps
- 1920 x 1080, BGGR10, HDR @ 30 fps
- 3840 x 2160, BGGR12, no HDR @ 30 fps
- 3840 x 2160, BGGR10, HDR @ 15 fps
- 3840 x 2160, BGGR12, HDR @ 15 fps
- 3840 x 2160, BGGR10, no HDR @ 30 fps

The driver is V4L2 compliant, using common V4L2 controls:

- `V4L2_CID_VBLANK`
- `V4L2_CID_HBLANK`
- `V4L2_CID_PIXEL_RATE`
- `V4L2_CID_ANALOGUE_GAIN`
- `V4L2_CID_EXPOSURE`

The HDR mode can be enabled or disabled by enabling the short exposure stream and route (on VC1). In this case, the standard V4L2 control `V4L2_CID_HDR_SENSOR_MODE` is automatically enabled and the sensor operates in staggered HDR mode.

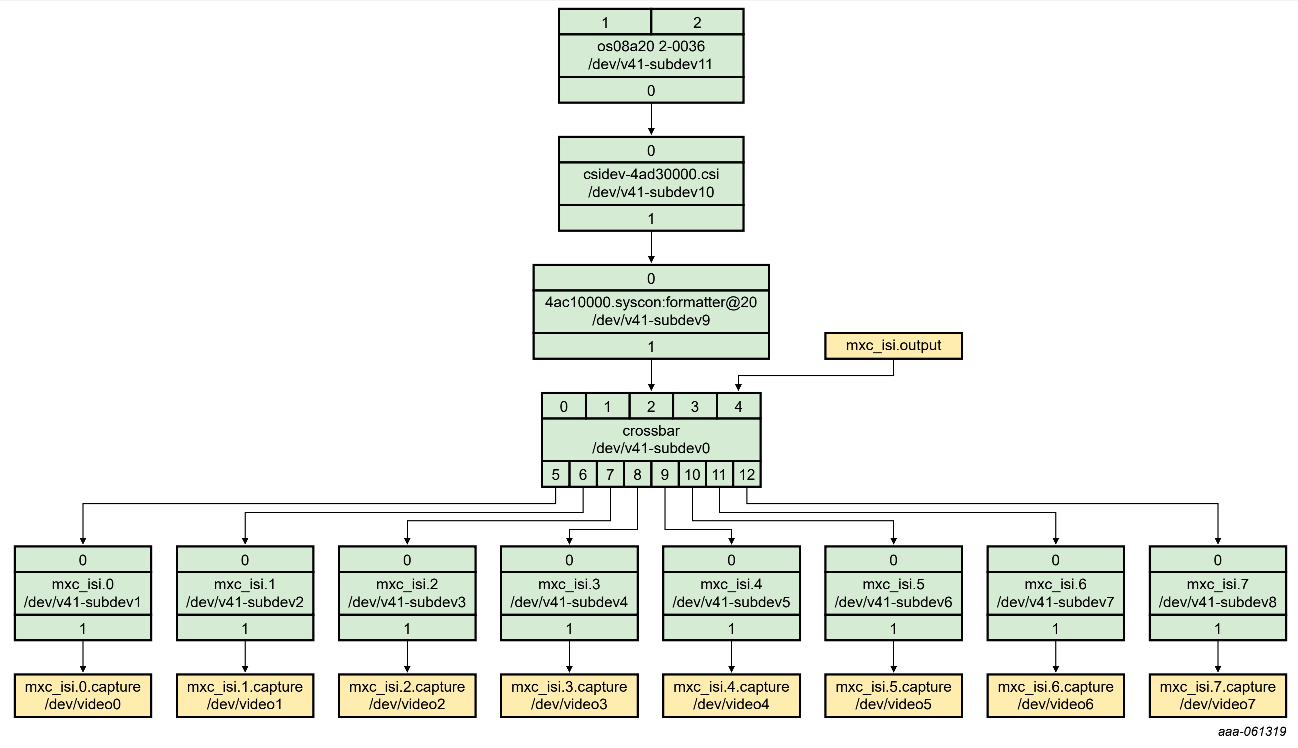
1
2
os08a20 2-0036
/dev/v41-subdev11
0
csidev-4ad30000.csi
/dev/v41-subdev10
1
4ac10000.syscon:formatter@20
/dev/v41-subdev9
mxc_isi.output
0 1 2 3 4
crossbar
/dev/v41-subdev0
5 6 7 8 9 10 11 12
mxc_isi.0 /dev/v41-subdev1
mxc_isi.1 /dev/v41-subdev2
mxc_isi.2 /dev/v41-subdev3
mxc_isi.3 /dev/v41-subdev4
mxc_isi.4 /dev/v41-subdev5
mxc_isi.5 /dev/v41-subdev6
mxc_isi.6 /dev/v41-subdev7
mxc_isi.7 /dev/v41-subdev8
mxc_isi.0.capture /dev/video0
mxc_isi.1.capture /dev/video1
mxc_isi.2.capture /dev/video2
mxc_isi.3.capture /dev/video3
mxc_isi.4.capture /dev/video4
mxc_isi.5.capture /dev/video5
mxc_isi.6.capture /dev/video6
mxc_isi.7.capture /dev/video7
aaa-061319

Figure 19. Media controller graph for OS08A20

The pipeline may be configured manually like this, for example, for SGRBG10/3840x2160 with HDR enabled:

Set links:

```
./media-ctl -d /dev/media0 -l "os08a20 2-0036":0 -> "csidev-4ad30000.csi":0[1]
./media-ctl -d /dev/media0 -l "csidev-4ad30000.csi":1 ->
 "4ac10000.syscon:formatter@20":0[1]
./media-ctl -d /dev/media0 -l "4ac10000.syscon:formatter@20":1 ->
 "crossbar":2[1]
./media-ctl -d /dev/media0 -l "crossbar":5 -> "mxc_isi.0":0[1]
./media-ctl -d /dev/media0 -l "mxc_isi.0":1 -> "mxc_isi.0.capture":0[1]
./media-ctl -d /dev/media0 -l "crossbar":7 -> "mxc_isi.2":0[1]
./media-ctl -d /dev/media0 -l "mxc_isi.2":1 -> "mxc_isi.2.capture":0[1]
```

Set routes:

```
./media-ctl -d /dev/media0 -R "os08a20 2-0036" [1/0 -> 0/0 [1], 2/0 -> 0/1 [1]]
./media-ctl -d /dev/media0 -R "crossbar" [2/0 -> 5/0 [1], 2/1 -> 7/0 [1]]
./media-ctl -d /dev/media0 -R "csidev-4ad30000.csi" [0/0 -> 1/0 [1], 0/1 -> 1/1
 [1]]
./media-ctl -d /dev/media0 -R "4ac10000.syscon:formatter@20" [0/0 -> 1/0 [1],
 0/1 -> 1/1 [1]]
```

Set formats for stream 0 (long exposure):

```
./media-ctl -d /dev/media0 -V "os08a20 2-0036":0/0 [fmt:SBGGR10/3840x2160
 field:none]
./media-ctl -d /dev/media0 -V "csidev-4ad30000.csi":0/0 [fmt:SBGGR10/3840x2160
 field:none]
./media-ctl -d /dev/media0 -V "4ac10000.syscon:formatter@20":0/0
 [fmt:SBGGR10/3840x2160 field:none]
```

```
./media-ctl -d /dev/media0 -V "crossbar":2/0 [fmt:SBGGR10/3840x2160 field:none]
./media-ctl -d /dev/media0 -V "mxc_isi.0":0/0 [fmt:SBGGR10/3840x2160 field:none]
```

Set formats for stream 1 (short exposure):

```
./media-ctl -d /dev/media0 -V "os08a20 2-0036":0/1 [fmt:SBGGR10/3840x2160
 field:none]
./media-ctl -d /dev/media0 -V "csidev-4ad30000.csi":0/1 [fmt:SBGGR10/3840x2160
 field:none]
./media-ctl -d /dev/media0 -V "4ac10000.syscon:formatter@20":0/1
 [fmt:SBGGR10/3840x2160 field:none]
./media-ctl -d /dev/media0 -V "crossbar":2/1 [fmt:SBGGR10/3840x2160 field:none]
./media-ctl -d /dev/media0 -V "mxc_isi.2":0/0 [fmt:SBGGR10/3840x2160 field:none]
```

The Omnivision OS08A20 sensor is supported on i.MX 95 (using the OX05B1S driver) in conjunction with the NEO ISP, but it is also supported on i.MX 8M Plus, in which case the VSI ISP is used, and there is another sensor driver, which is built outside the kernel.

6.1.3.2.3 0X03C10

The Omnivision OX03C10 sensor is supported on i.MX 95 using serializer/deserializer solutions from the following vendors:

- Analog Devices: MAX96717/MAX96724
- Texas Instruments: DS0UB953/DS0UV960

Up to 4 sensors are supported by each SerDes solution. The following subsections provide a brief overview of each solution. Both solutions share the same OX03C10 driver and custom controls.

6.1.3.2.3.1 Analog Devices SerDes

The Analog Devices SerDes solution is made up of one deserializer board having a MAX96724 chip and up to 4 sensor modules, each having an Omnivision OX03C10 sensor and one MAX96717 serializer chip. The modules are connected by Gigabit Multimedia Serial Links (GMSLs) over coax cables.

To activate the Analog Devices solution, the following kernel modules need to be selected in the Linux kernel configuration:

- `max96724`: **Device Drivers** -> **Multimedia Support** -> **Media ancillary drivers** -> **Video decoders** -> **Maxim MAX96724 GMSL deserializer support**
  - `drivers/media/i2c/max96724.c`
- `mx95mbcam`: **Device Drivers** -> **Multimedia Support** -> **Media ancillary drivers** -> **Video decoders** -> **NXP OX03C10 + MAX961717 combo GMSL2 camera support**
  - `drivers/media/i2c/mx95mbcam.c`
  - `drivers/media/i2c/max96717.c`
  - `drivers/media/i2c/ox03c10.c`

The drivers are V4L2 compliant, using common V4L2 controls (documented in Section 6.1.5), with a few custom controls added to the OX03C10 sensor driver to be able to set exposure, analog gain, digital gain, and white balance gain for each mode of operation in one single-system call. The custom controls UAPI (user-space API) is documented in the following header:

- `Include/uapi/linux/ox03c10.h`

The following is a list of the custom controls added. For the full description of the data structures needed to be passed to each control, see the UAPI header files:

- `V4L2_CID_OX03C10_EXPOSURE`: set exposure parameters for SPD, LCG, and VS modes.
- `V4L2_CID_OX03C10_ANALOGUE_GAIN`: set analog gains for HCG, LCG, SPD, and VS modes.
- `V4L2_CID_OX03C10_DIGITAL_GAIN`: set digital gains for HCG, LCG, SPD, and VS modes.
- `V4L2_CID_OX03C10_WB_GAIN`: set white B, Gb, Gr and R gains for HCG, LCG, SPD, and VS modes.
- `V4L2_CID_OX03C10_PWL_EN`: enable/disable PWL (piecewise linear) compression.
- `V4L2_CID_OX03C10_PWL_CTRL`: change PWL mode.
- `V4L2_CID_OX03C10_PWL_KNEE_POINTS_LUT`: set the PWL knee points.
- `V4L2_CID_OX03C10_OTP_CORRECTION`: read OTP correction parameters.

6.1.3.2.3.2 Texas Instruments SerDes

The Texas Instruments SerDes solution is made up of one deserializer board having a DSOUV960 chip and up to 4 sensor modules, each having an Omnivision OX03C10 sensor and one DSOU953 a serializer chip. The modules are connected by FPD-Link III (Flat Panel Display Link) over coax cables.

To activate the Texas Instruments solution, the following kernel modules need to be selected in the Linux kernel configuration:

- `mx95mbcam`: **Device Drivers** -> **Multimedia support** -> **Media ancillary drivers** -> **Camera sensor devices** -> **OmniVision OX03C10 sensor support**
- `ds90ub960-q1`: **Device Drivers** -> **Multimedia support** -> **Media ancillary drivers** -> **Video serializers and deserializers** -> **TI FPD-Link III/IV Deserializers**

6.1.4 Software operation

The V4L2 operations for capture support modes, picture formats, and picture sizes vary for each capture interface.

The imx-test repository has unit tests for these interfaces in the `mxc_v4l2_test`. See README for details on how to run tests.

6.1.5 V4L2 capture

Video for Linux Two (V4L2) is a Linux standard. The API specification is available at https://www.kernel.org/doc/html/latest/userspace-api/media/v4l/v4l2.html.

The V4L2 capture device includes two interfaces: the capture and overlay interfaces using the V4L2 API for capture and overlay devices.

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

1. Sets the capture pixel format and size using `IOCTL VIDIOC_S_FMT`.
2. Sets the control information using `IOCTL VIDIOC_S_CTRL` for rotation.
3. Requests a buffer using `IOCTL VIDIOC_REQBUFS`.
4. Memory maps the buffer to its user space.
5. Executes the `IOCTL VIDIOC_DQBUF`.
6. Passes the data that requires post-processing to the buffer.
7. Queues the buffer using the IOCTL command `VIDIOC_QBUF`.
8. Starts the stream by executing `IOCTL VIDIOC_STREAMON`.
   `VIDIOC_STREAMON` and `VIDIOC_OVERLAY` cannot be enabled simultaneously.

The following table lists the V4L2 capture IOCTLs used in the i.MX Capture Drivers. For more information, see Section 6.5.

Table 62. V4L2 capture API IOCTLs

IOCTL	Description
`VIDIOC_QUERYCAP`	Query Device Capabiities
`VIDIOC_G_FMT VIDIOC_S_FMT`	Get or Set Data format
`VIDIOC_S_DEST_CROP`	Set cropping rectange
`VIDIOC_REQBUFS`	Initiate Memory Mapping
`VIDIOC_QueryBUF`	Query status of buffer
`VIDIOC_QBUF`, `VIDIOC_DQBUF`	Exchange buffer with driver
`VIDIOC_STREAMON`, `VIDIOC_STREAMOFF`	Start or stop streaming
`VIDIOC_G_CTRL`, `VIDIOC_S_CTRL`	Get or set the value of a control
`VIDIOC_CROPCAP`	Query cropping capabilities
`VIDIOC_G_CROP`, `VIDIOC_S_CROP`	Get or set Cropping
`VIDIOC_OVERLAY`	Start or stop video overlay
`VIDIOC_G_FBUF`, `VIDIOC_S_FBUF`	Get or set frame buffer ovrelay parameters
`VIDIOC_G_PARM`, `VIDIOC_S_PARM`	Get or set streaming parameters
`VIDIOC_G_STD`, `VIDIOC_S_STD`	Get or Set the video standard
`VIDIOC_G_OUTPUT`, `VIDIOC_S_OUTPUT`	Get or Set the video output
`VIDIOC_G_INPUT`, `VIDIOC_S_INPUT`	Get or set the video input
`VIDIOC_ENUMSTD`	Enumerate video standards
`VIDIOC_ENUMOUTPUT`, `VIDIOC_ENUMINPUT`	Enumerate output and inputs
`VIDIOC_ENUM_FMT`	Enumerate image formats
`VIDIOC_ENUM_FRAMESIZE`, `VIDIOC_ENUM_FRAMEINTERVALSS`	Enumerate frame sizes and intervals
`VIDIOC_DBG_G_CHIP_IDENT`	Chip Identification

6.1.6 Source code structure

The table below lists the capture driver source files.

- For i.MX 6 and i.MX 7, the source files are in `drivers/media/platform/mxc/capture`.
- For i.MX 8 series, the source files are in `drivers/media/platform/imx8`.
- For MIPI-CSI, the source files are in `drivers/mxc/mipi`.

Table 63. Omnivision V4L2 camera driver files

File	Description
• `drivers/media/platform/nxp/dwc-mipi-csi2.c` • `drivers/phy/freescale/phy-fsl-imx9-dphy-rx.c`	i.MX 93/95 MIPI CSI Interface driver
• `drivers/media/platform/nxp/imx8mq-mipi-csi2.c` • `drivers/staging/media/imx/imx8-mipi-csi2-sam.c` • `drivers/media/platform/nxp/imx-mipi-csis.c`	i.MX 8 MIPI-CSI2 Capture Interface driver
• `drivers/media/platform/nxp/imx-parallel-csi.c`	i.MX 8 Parallel-CSI Interface driver
• `drivers/staging/media/imx/imx8-isi-core.c` • `drivers/staging/media/imx/imx8-isi-cap.c` • `drivers/staging/media/imx/imx8-isi-hw.c`	i.MX 8M Plus ISI Capture Controller driver

Table 63. Omnivision V4L2 camera driver files*...continued*

File	Description
• `drivers/staging/media/imx/imx8-isi-m2m.c` • `drivers/staging/media/imx/imx8-isi-fmt.c`	
• `drivers/media/platform/nxp/imx8-isi/`	i.MX 8, i.MX 8M (except for i.MX 8M Plus), and i.MX 8X ISI Capture Controller driver
• `drivers/media/platform/nxp/imx8- isi/imx8-isi-core.c` • `drivers/media/platform/nxp/imx8-isi/imx8-isi-video.c` • `drivers/media/platform/nxp/imx8-isi/imx8-isi-crossbar.c` • `drivers/media/platform/nxp/imx8-isi/imx8-isi-m2m.c` • `drivers/media/platform/nxp/imx8-isi/imx8-isi-pipe.c`	i.MX 8 ISI Capture Controller driver
• `drivers/media/i2c/ov5640.c` • `drivers/media/i2c/max9286.c` • `drivers/media/i2c/rdacm20.c`	i.MX 8 Omnivision Camera V3 Camera interface
• `drivers/media/platform/nxp/imx-jpeg/`	i.MX 8 JPEG hardware interface
• `drivers/mxc/mipi/mxc_mipi_csi2.c` • `drivers/mxc/mipi/mxc_mipi_csi2.h`	i.MX 6 and i.MX 7 MIPI-CSI2 interface core driver
• `drivers/media/platform/mxc/capture/ipu_bg_overlay_sdc.c` • `drivers/media/platform/mxc/capture/ipu_csi_enc.c` • `drivers/media/platform/mxc/capture/ipu_fg_overlay_sdc.c` • `drivers/media/platform/mxc/capture/ipu_prp_enc.c` • `drivers/media/platform/mxc/capture/ipu_prp_vf_sdc_bg.c` • `drivers/media/platform/mxc/capture/ipu_prp_vf_sdc.c` • `drivers/media/platform/mxc/capture/ipu_still.c` • `drivers/media/platform/mxc/capture/v4l2-int-device.c`	i.MX 6 IPU V4L2 plugin
• `drivers/media/platform/mxc/capture/mx6s_capture.c` • `drivers/media/platform/mxc/capture/ov5640.c`	CSI Omnivision Camera V4L2 plugin
• `drivers/media/platform/mxc/capture/ov5640_v2.c`	Paralllel CSI Omnivision Camera V4L2 plugin
• `drivers/media/platform/mxc/capture/ov5640_mipi.c` • `drivers/media/platform/mxc/capture/ov5640_camera_mipi_int.c`	MIPI-CSI Omnivision Camera V4L2 plugin
• `drivers/media/platform/mxc/capture/ov5640_mipi_v2.c`	MIPI-CSI2 Omnivision Camera V4L2 plugin
• `drivers/media/platform/mxc/capture/adv7180.c`	TV Decoder ADV7180 V4L2
• `drivers/staging/media/imx/hdmirx/cdns-hdmirx-audio.c` • `drivers/staging/media/imx/hdmirx/cdns-hdmirx-hdcp.c` • `drivers/staging/media/imx/hdmirx/cdns-hdmirx-hw.c` • `drivers/staging/media/imx/hdmirx/cdns-hdmirx-phy.c`	i.MX 8 HDMI RX

Table 63. Omnivision V4L2 camera driver files*...continued*

File	Description
• `drivers/staging/media/imx/hdmirx/cdns-hdmirx-phy.h` • `drivers/staging/media/imx/hdmirx/cdns-hdmirx.c` • `drivers/staging/media/imx/hdmirx/cdns-mhdp-hdmirx.c` • `drivers/staging/media/imx/hdmirx/cdns-mhdp-hdmirx.h`	

6.2 Display overview

6.2.1 Introduction

The i.MX Display system uses display controllers to optimize video data movement to display interfaces and graphics processing. Each display controller is implemented through a Linux driver and into a display framework either frame buffer or DRM. In some cases, a display controller includes authentication ensuring a secure video pipeline. In others, the display controller includes additional features for scaling, de-interlacing, tiling, and color conversion during transfer. For i.MX 8, supporting multiple displays is done with use of two controllers working together. This section provides a high level overview of i.MX display controllers and interfaces and the difference between frame buffer and DRM display drivers. The following display controllers are used.

- IPU
- PXP
- eLCDIF
- DPU
- DCSS: on i.MX 8M only

A display interface interfaces to the display controller, display panel and sometimes encoders display bridges. The following display interfaces are supported.

- EPDC: supporting E Ink displays
- Parallel: supporting LCD displays
- LVDS: supporting LVDS displays
- HDMI: supporting both on chip and external HDMI
- Display Port: supporting eDP panels
- MIPI-DSI: supporting MIPI displays

Note: *Analog display is no longer supported. Analog interface was used in i.MX 37 and i.MX 5 families.*

The following HDMI display bridges/encoder are supported.

- Parallel to HDMI: using Silicon Image si902x
- LVDS to HDMI: using ITE it6263
- MIPI-DSI to HDMI: using Analog Devices adv7535

Each SOC supports different display features. Some of these are configured in the device trees located at `arch/arm/boot/dts and arch/arm64/boot/dts`. See the hardware reference manual for more details on the following.

- Throughput
  – Number of outputs
  – Pixel clock rate
  – Maximum number of displays and corresponding resolution

  - Resolution at 60 Hz
- Interface
  - Parallel: number of ports and bit size
  - LVDS: number of lanes and channels
  - MIPI-DSI: number of ports, lanes channels and speed
- Processing
  - On the fly combining, including high resolution displays
  - Off-line combining speed

6.2.2 Frame buffer

Frame buffer drivers are supported for i.MX 6 and i.MX 7, but not for i.MX 8. The frame buffer drivers are supported using the `imxfb` driver in `drivers/video/fbdev`. The frame buffer kernel `fbdev` structure is defined here or here on kernel.org. For more information on i.MX V4LS, see Section 6.5.

The panels are supported with the frame buffer driver for the TRULY and E Ink panels in the `video/fbdev/mxc` folder. See the panels supported by searching for "PANEL" in the `imx_v7_defconfig`. The Truly panels are only supported with the MIPI DSI interface. The E Ink panels are only supported with the EPDC interface.

6.2.3 Direct Rendering Model (DRM)

Direct Rendering Model (DRM) is the new display driver used for i.MX. The i.MX DRM driver is in `drivers/gpu/drm/imx`. Other components have DRM interfaces, such as GPU and DCSS. The DRM framework is documented here on kernel.org.

The i.MX DRM drivers are implemented with the following drivers.

- Hardware library support files
- Core DRM drivers
- Hardware dependent DRM drivers
- HDMI DRM drivers supporting HDP HDMI/Display port

The DRM driver uses the DPU for the i.MX 8QuadMax and i.MX 8QuadXPlus, uses LCDIF for the i.MX 8M Quad and i.MX 8M Mini, uses DCNANO for the i.MX 8ULP, and uses LCDIFv3 for i.MX 8M Plus, i.MX 93, and i.MX 91.

The i.MX DRM framework also includes panel drivers, which exist in `driver/gpu/panel`. The supported DRM panels are Simple Panel, Raydium RM67191, Raydium RM68200, and Raydium RM67199.

6.2.4 Display resolution

The display resoluton calculation uses the following factors.

- Frame Width
- Frame Height
- Frame rate (fps)
- Blanking Interval: provided in the display's DS up to 35% (1.35), using the minimum values

The pixel clock [MHz] is calculated according to Frame Width x Frame Height x Frames Rate x Blanking Interval.

Consider the following points:

- Data format (pixel per clock)
- Display's source clock (`DI#_CLK_EXT` bit)
- Load on the display controller (DC)

6.2.5 Authentication

Display authentication allows hardware processing to ensure that the display content is not compromised. This is done through a display authentication CRC using the authentication hardware. This hardware is the DCIC integrated through the frame buffer display framework on i.MX 6 and the DPU implemented in the DRM display framework for i.MX 8.

Display authentication CRC is supported on the following SoCs:

- i.MX 6 SoloX supports authentication using DCIC for 1 display.
- i.MX 6 QuadPlus/Quad/Dual support authentication using DCIC with 2 displays.
- i.MX 8QuadXPlus can authenticate 2 display using the DPU.
- i.MX 8QuadMax can authenticate 4 displays using DPU.

6.2.6 Tiling

Tiling through hardware provides optimized video data display. This is implemented through different hardware blocks. The newest feature is the Display Prefetch Resolve (DPR), which increases performance on the i.MX 6 QuadPlus, i.MX 8QuadMax, and i.MX 8QuadXPlus.

Tile support is enabled on the following:

- i.MX 6Quad/Dual supports tiling using Video Data Order Adapter (VDOA).
- i.MX 6QuadPlus supports both tiling VDOA and Display Prefetch Resolve (DPR) version 1.
- i.MX 8QuadXPlus and i.MX 8QuadMax support tiling using Display Prefetch Resolve (DPR) version 2.

6.3 Display Controllers

6.3.1 Display Processing Unit (DPU)

6.3.1.1 Introduction

The display processing unit (DPU) is designed to support video and graphics processing functions and to interface with video and still display sensors and displays. The DPU driver provides internal kernel-level APIs to manipulate logical channels. A logical channel represents a complete DPU processing flow. For example, a complete DPU processing flow (logical channel) might consist of reading a YUV buffer from memory and displaying it to an external interface. The DPU API consists of a set of common functions for all channels. Its functions are to initialize channels, set up buffers, enable and disable channels and set up interrupts.

Typical logical channels include:

- CSI direct to memory
- Memory to synchronous frame buffer background
- Memory to synchronous frame buffer foreground

The higher-level drivers are responsible for memory allocation and providing a user-level API. DPU interfaces are available for capture in the V4L2 framework and for display using the DRM display framework. DPU interfaces with LVDS, MIPI-DSI, HDMI, and Parallel display interfaces.

The DPU display controller supports a 32bit display composition engine that includes the following:

- Two Display output streams on independent panels.
- Two-layer composition
- Automatic safety stream panic plus detection using CRC matching using a Region CRC checker

The DPU display controller supports a 2D composition engine, which provides efficiency, performance, and safety. The DPU 2D graphics engine support reduces the burden on the GPU so it only does 3D GPU. Video efficiency with overlay native video and graphics uses minimal access to system memory. Power efficiencies allow the 3D engine to be off for windowing GUI's like the Android Hardware Composer.

The DPU also supports the following for authentication.

- CRC checker with 8 stackable regions maskable, exclusive top-to-bottom priority
- CRC check can be inserted after any stage in the post-processing pipe
- CRC failure can generate SW interrupt, or switch the Frame Gen to either Safety Stream or Constant Plane

The DPU display interface cache supports the following.

- Each display plane has a multi-line cache.
- This contains 8 lines of pixels for each plane.
- RGB and YUV formats supported.
- Supports Video and GPU tile formats
- Contents are fetched from memory to fill the cache ahead of time.
- Horizontal and vertical fetches supported.
- Warp fetches not supported, require bypass.

6.3.1.2 DRM

The display processing unit (DPU) interfaces with the DRM driver supporting video display.

6.3.1.3 Source code structure

The DPU drivers are grouped into DRM, blitting, and main processing. Common functions are provided in the `drivers/gpu/drm/imx/dpu` and `drivers/gpu/imx/dpu-blit` while the main driver exists in `drivers/gpu/imx/dpu`. The following table lists the source files.

Table 64. DPU driver source files

File	Description
DRM Source	
`drivers/gpu/drm/imx/dpu/dpu-plane`	DRM DPU Plane
`drivers/gpu/drm/imx/dpu/dpu-crtc`	DRM DPU CRTC
`drivers/gpu/drm/imx/dpu/dpu-blit`	DRM DPU blitter
`drivers/gpu/drm/imx/dpu/dpu-kms`	DRM DPU KMS
DPU Blitter Source	
`drivers/gpu/imx/dpu-blit/dpu-blit`	DPU Bliter
`drivers/gpu/imx/dpu-blit/dpu-blit-registers.h`	DPU Blit registers
DRM Core Source	
`drivers/gpu/imx/dpu/dpu-vscaler.c`	DPU VScaler
`drivers/gpu/imx/dpu/dpu-fetchwarp.c`	DPU Fetchwarp
`drivers/gpu/imx/dpu/constframe.c`	DPU Const Frame
`drivers/gpu/imx/dpu/dpu-prv.h`	DPU Private headers
`drivers/gpu/imx/dpu/dpu-disengcfg.c`	DPU Display Configurations
`drivers/gpu/imx/dpu/dpu-fetchunit.c`	DPU Fetch Unit
`drivers/gpu/imx/dpu/dpu-framegen.c`	DPU Frame Generator

Table 64. DPU driver source files*...continued*

File	Description
`drivers/gpu/imx/dpu/dpu-hscaler.c`	DPU HScaler
`drivers/gpu/imx/dpu/dpu-extdst.c`	DPU External Destination
`drivers/gpu/imx/dpu/dpu-common.c`	DPU Common
`drivers/gpu/imx/dpu/dpu-fetchlayer.c`	DPU Fetch Layer
`drivers/gpu/imx/dpu/dpu-tcon.c`	DPU TCon
`drivers/gpu/imx/dpu/dpu-layerblend.c`	DPU Layer Blend
`drivers/gpu/imx/dpu/dpu-fetcheco.c`	DPU Fetch Encode
`drivers/gpu/imx/dpu/dpu-fetchdecode.c`	DPU Decode

6.3.1.4 Menu configuration options

The following Linux kernel configuration options are provided for the DPU module:

Device Drivers -> **i.MX DPU core support**

6.3.2 Image Processing Unit (IPU)

6.3.2.1 Introduction

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

The IPU is a display controller and supports the following display interfaces, which are supported through the frame buffer display framework. The access is only exposed through the frame buffer fbdev application framework.

- Parallel
- LVDS
- HDMI
- MIPI-DSI

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
- Enable or disable prefetching linear frames by using PRE/PRG
- Enable or disable resolving tiled frames by using PRE/PRG

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

6.3.2.2 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.

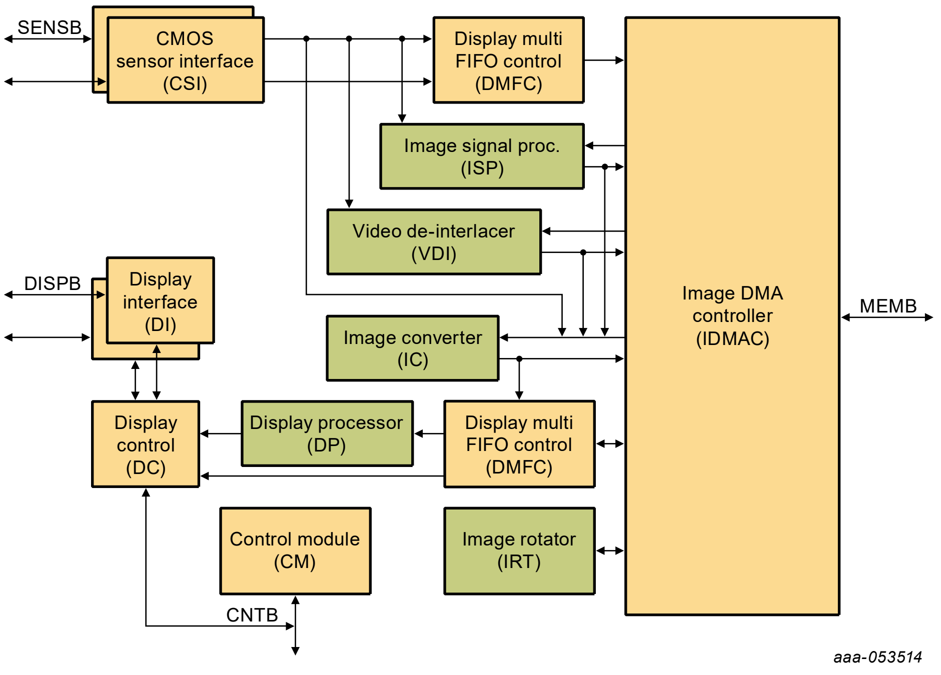

Figure 20. IPUv3EX/IPUv3H IPU module overview

6.3.2.3 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)
- Prefetch/Resolve Engine/Gasket (PRE/PRG)

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

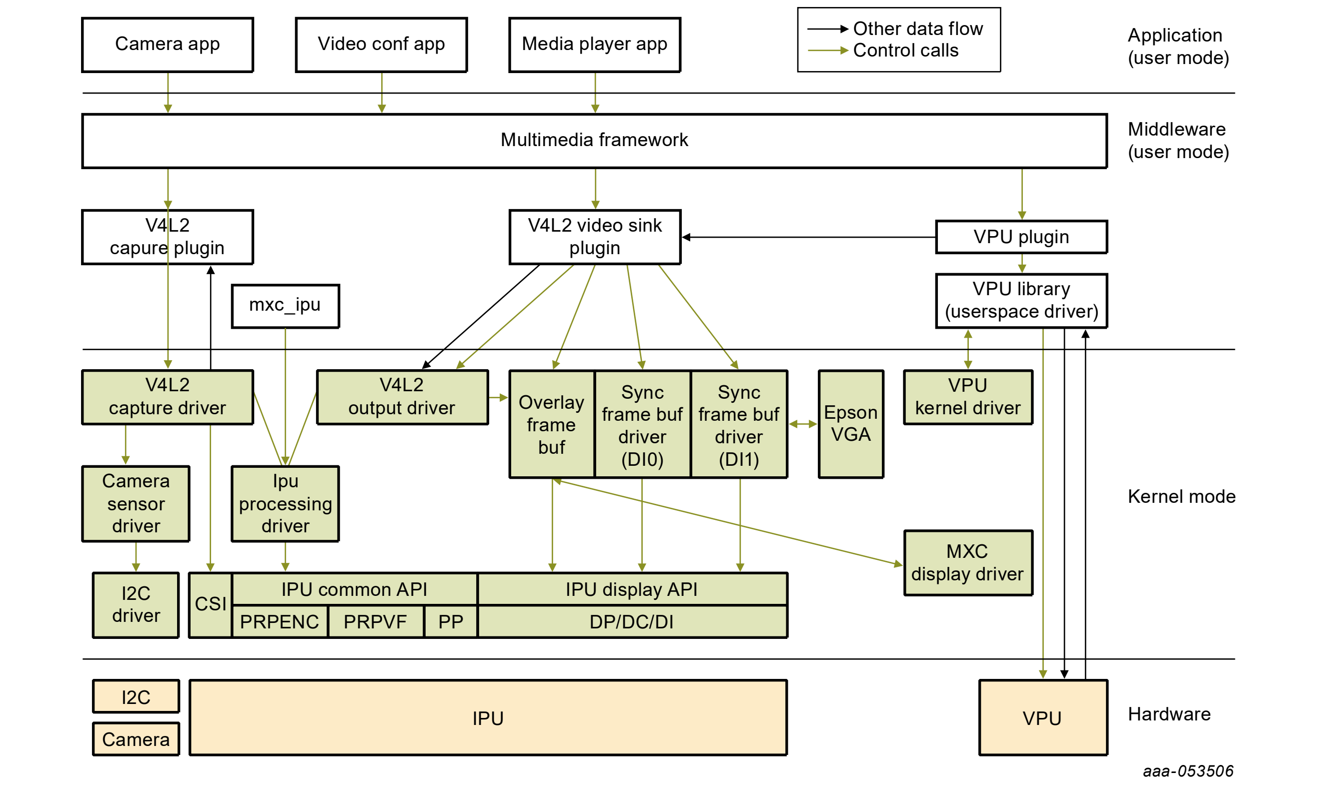

Figure 21. Graphics/Video drivers software interaction for IPUv3

The drivers for IPUv1 are named simply `ipu`. Drivers for IPUv3 contain `3` or `v3` in the name. 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, which provide system interface to user space or V4L2 drivers. The frame buffer device drivers are available in `drivers/video/mxc`. The V4L2 device drivers are available in `drivers/media/platform/mxc`.
- The MXC display driver is introduced as a simple framework to manage interaction between IPU and display device drivers (e.g., LCD, LVDS, HDMI, MIPI, etc.)

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

6.3.2.4 IPU frame buffer drivers overview

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

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

Other than the physical memory allocation and LCD panel configuration, the common kernel video API is 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.

6.3.2.5 IPU frame buffer hardware operation

The frame buffer interacts with the IPU hardware driver module.

6.3.2.6 IPU frame buffer software operation

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

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

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

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

The MXC frame buffer driver (`drivers/video/mxc/mxc_ipuv3_fb.c`) interacts closely with the generic Linux frame buffer driver (`drivers/video/fbdev/core/fbmem.c`).

6.3.2.7 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 setting colors, palette registration, image blitting, and memory mapping. The IPU reads the raw pixel data from the frame buffer memory and sends it to the panel for display.

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

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

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

The following features are supported:

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

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 using `mmap` for performing direct blitting operations

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

6.3.2.8 IPU backlight driver

IPU drivers also control the backlight. 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.**

6.3.2.9 IPU device driver

IPU (processing) device driver provide image processing features: 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 and queue task, and then the block waits for the task to finish. The driver now supports only blocking method, 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 the task, the users only need to fill `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 the kernel level (V4L2 driver for example), or by IPU_QUEUE_TASK IOCTL under `/dev/mxc_ipu` if from the application level.

6.3.2.10 Source code structure

The source files associated with the IPU, Sensor, V4L2, and Panel drivers are available in the following folders.

- `drivers/mxc/ipu3`
- `drivers/video/mxc`
- `drivers/video/fbdev/mxc`
- `drivers/video/backlight`

See Section 6.5 for more information on the IPU V4L2 driver files.

Table 65. IPU driver files

File	Description
`driveers/mxc/ipu3/ipu_common.c`	IPU common library functions
`driveers/mxc/ipu3/ipu_common.c`	IPU common library functions
`drivers/mxc/ipu3/ipu_ic.c`	IPU IC base driver
`drivers/mxc/ipu3/ipu_device.c`	IPU driver device interface and FOPS functions

Table 65. IPU driver files*...continued*

File	Description
`drivers/mxc/ipu3/ipu_capture.c`	IPU CSI capture base driver
`drivers/mxc/ipu3/ipu_disp.c`	IPU display functions
`drivers/mxc/ipu3/ipu_calc_stripes_sizes.c`	Multistripes method functions for `ipu_device.c`
`drivers/mxc/ipu3/pre.c`	i.MX 6 QuadPlus Prefetch/Resolve the engine driver
`drivers/mxc/ipu3/prg.c`	i.MX 6 QuadPlus Prefetch/Resolve the Gasket driver
`drivers/mxc/ipu3/mxc_ipuv3_fb.c`	Driver for synchronous frame buffer
`drivers/mxc/ipu3/vdoa.c`	VDOA post-processing driver, used by `ipu_device.c`
`drivers/video/fbdev/mxc/mxc_lcdif.c`	Display Driver for CLAA-WVGA and SEIKO-WVGA LCD support
`drivers/video/fbdev/mxc/mxc_hdmi.c`	Display Driver for HDMI interface
`drivers/video/fbdev/mxc/ldb.c`	Driver for synchronous frame buffer for on chip LVDS
`drivers/video/fbdev/mxc/mxc_dispdrv.c`	Display Driver framework for synchronous frame buffer
`drivers/video/fbdev/mxc/mxc_edid.c`	Driver for EDID

The table below lists the header files associated with the IPU and Panel drivers.

Table 66. IPU Global header files

File	Description
`drivers/mxc/ipu3/ipu_param_mem.h`	Helper functions for IPU parameter memory access
`drivers/mxc/ipu3/ipu_prv.h`	Header file for pre-processing drivers
`drivers/mxc/ipu3/ipu_regs.h`	IPU register definitions
`drivers/mxc/ipu3/pre-regs.h`	Prefetch/Resolve Engine register definitions
`drivers/mxc/ipu3/prg-regs.h`	Prefetch/Resolve Gasket register definitions
`drivers/mxc/ipu3/vdoa.h`	Header file for VDOA drivers
`drivers/video/fbdev/mxc/mxc_dispdrv.h`	Header file for display driver
`include/linux/uapi/mxcfb.h`	Header file for the synchronous frame buffer driver
`include/linux/uapi/ipu.h`	Header file for IPU APIs

6.3.2.11 Menu configuration options

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

In menu configuration, enable the following modules:

- **CONFIG_MXC_IPU_V3**: Includes support for the Image Processing Unit. In `menuconfig`, this option is available under:
  Device Drivers -> **MXC support drivers** -> **Image Processing Unit Driver**
  By default, this option is **Y** for all architectures.
  If `ARCH_MXC` is true, **CONFIG_MXC_IPU_V3** will be set.
- **CONFIG_MXC_IPU_V3_PRG**: This enables support for the IPUv3 prefetch gasket engine to support double buffer handshake control between IPUv3 and prefetch engine (PRE), snoop the AXI interface for display

refresh requests to memory, and modify the request address to fetch the double buffered row of blocks in OCRAM.
Device Drivers -> **MXC support drivers** -> **i.MX IPUv3 prefetch gasket engine**
This option depends on **CONFIG_MXC_IPU_V3** and **CONFIG_MXC_IPU_V3_PRE**.

- **CONFIG_MXC_IPU_V3_PRE**: This enables support for the IPUv3 prefetch engine to improve the system memory performance. The engine has the capability to resolve framebuffers in tile pixel format to linear.
  Device Drivers -> **MXC support drivers** -> **i.MX IPUv3 prefetch engine**
  This option depends on **CONFIG_MXC_IPU_V3**. To enable this option, select **CONFIG_MXC_IPU_V3_PRG**.
- **CONFIG_MXC_CAMERA_OV5640_MIPI**: Option for both the OV5640 MIPI sensor driver and the use case driver. This option is dependent on the **VIDEO_MXC_CAPTURE** option. In `menuconfig`, this option is available under:
  Device Drivers -> **Multimedia support** -> **V4L platform devices** -> **MXC Video For Linux Video Capture** -> **MXC Camera/V4L2 PRP Features support** -> **OmniVision 5640 Camera support using mipi**
- **CONFIG_MXC_CAMERA_OV5640**: Option for both the OV5640 sensor driver and the use case driver. This option is dependent on the **VIDEO_MXC_CAPTURE** option. In `menuconfig`, this option is available under:
  Device Drivers -> **Multimedia platform** -> **V4L platform devices** -> **MXC Video For Linux Video Capture** -> **MXC Camera/V4L2 PRP Features support** -> **OmniVision ov5640 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 hardware blocks):
  CSI -> **IC** -> **MEM MEM** -> **IC (PRP VF)** -> **MEM**
  Use case driver for dumb sensor or **CSI** -> **IC(PRP VF)** -> **MEM** for smart sensors. In `menuconfig`, this option is available under:
  Multimedia devices -> **Video capture adapters** -> **MXC Video For Linux Camera** -> **MXC Camera/V4L2 PRP Features support** -> **Pre-Processor VF SDC library**
  By default, this option is **M** for all.
- **CONFIG_MXC_IPU_PRP_ENC**: Option for the IPU:
  Use case driver for dumb sensors **CSI** -> **IC** -> **MEM MEM** -> **IC (PRP ENC)** -> **MEM**, or for smart sensors, **CSI** -> **IC(PRP ENC)** -> **MEM**.
  In `menuconfig`, this option is available under:
  Device Drivers -> **Multimedia Devices** -> **Video capture adapters** -> **MXC Video For Linux Camera** -> **MXC Camera/V4L2 PRP Features support** -> **Pre-processor Encoder library**
  By default, this option is set to **M** for all.
- **CONFIG_VIDEO_MXC_CAMERA**: This is configuration option for V4L2 capture Driver. This option is dependent on the following expression:
  VIDEO_DEV && MXC_IPU && MXC_IPU_PRP_VF_SDC && MXC_IPU_PRP_ENC
  In `menuconfig`, this option is available under:
  Device Drivers -> **Multimedia devices** -> **Video capture adapters** -> **MXC Video For Linux Camera**
  By default, this option is **M** for all.
- **CONFIG_VIDEO_MXC_OUTPUT**: This is configuration option for V4L2 output Driver. This option is dependent on **VIDEO_DEV && MXC_IPU** option. In `menuconfig`, this option is available under:
  Device Drivers -> **Multimedia devices** -> **Video capture adapters** -> **MXC Video for Linux Video Output**
  By default, this option is **Y** for all.
- **CONFIG_FB**: This is the configuration option to include frame buffer support in the Linux kernel. In `menuconfig`, this option is available under:
  Device Drivers -> **Graphics support** -> **Support for frame buffer devices**
  By default, this option is **Y** for all architectures.
- **CONFIG_FB_MXC**: This is the configuration option for the MXC Frame buffer driver. This option is dependent on the **CONFIG_FB** option. In `menuconfig`, this option is available under:
  Device Drivers -> **Graphics support** -> **MXC Framebuffer support**
  By default, this option is **Y** for all architectures.

- **CONFIG_FB_MXC_SYNC_PANEL**: This is the configuration option that chooses the synchronous panel frame buffer. This option is dependent on the **CONFIG_FB_MXC** option. In `menuconfig`, this option is available under:
  Device Drivers -> **Graphics support** -> **MXC Framebuffer support** -> **Synchronous Panel Framebuffer**
  By default this option is **Y** for all architectures.
- **CONFIG_FB_MXC_LDB**: This configuration option selects the LVDS module on i.MX 6 chip. This option is dependent on **CONFIG_FB_MXC_SYNC_PANEL** and **CONFIG_MXC_IPUV3 || FB_MXS** options. In `menuconfig`, this option is available under:
  Device Drivers -> **Graphics support** -> **MXC Framebuffer support** -> **Synchronous Panel Framebuffer** -> **MXC LDB**
- **CONFIG_FB_MXC_SII9022**: This configuration option selects the SII9022 HDMI chip. This option is dependent on **CONFIG_FB_MXC_SYNC_PANEL** option. In `menuconfig`, this option is available under:
  Device Drivers -> **Graphics support** -> **MXC Framebuffer support** -> **Synchronous Panel Framebuffer** -> **Si Image SII9022 DVI/HDMI Interface Chip**

6.3.3 Pixel Pipeline (PxP)

6.3.3.1 Introduction

The PxP is a display controller that works with the EPDC display interface. The Pixel Pipeline (PxP) DMA engine driver provides a unique API, which is implemented as a DMA Engine client that smooths over the details of different hardware offload engine implementations. Typically, the users of PxP DMA-ENGINE driver include EPDC driver, V4L2 Output driver, and the PxP user-space library.

The PxP driver uses PxP registers to interact with the hardware. For detailed hardware operations, see the Applications Processor Reference Manual document associated with SoC.

6.3.3.2 Software operation

There are different versions of PxP IP. To ease the maintenance for the new version of PxP used on i.MX 7Dual, which has new features mainly for EPDC like hardware collision detection, E Ink Gen-II waveform algorithm (REAGL/-D) processing in hardware, and hardware dithering support, there are different drivers (`drivers/dma/pxp/pxp_dma_v3.c`). However, each version uses the DMA Engine framework.

6.3.3.3 Key data structs

The PxP DMA Engine driver implementation depends on the DMA Engine Framework. There are three important structs in the DMA Engine Framework, which are extended by the PxP driver: `struct dma_device`, `struct dma_chan`, `struct dma_async_tx_descriptor`. The PxP driver implements several callback functions, which are called by the DMA Engine Framework (or DMA slave) when a DMA slave (client) interacts with the DMA Engine.

The PxP driver implements the following callback functions in `struct dma_device`:

- `device_alloc_chan_resources /* allocate resources and descriptors */`
- `device_free_chan_resources /* release DMA channel's resources */`
- `device_tx_status /* poll for transaction completion */`
- `device_issue_pending /* push pending transactions to hardware */`

and

- `device_prep_slave_sg /* prepares a slave DMA operation */`

- `device_terminate_all/* manipulate all pending operations on a channel, returns zero or error code */`

The first four functions are used by the DMA Engine Framework, and the last two are used by the DMA slave (DMA client). Notably, `device_issue_pending` is used to trigger the start of a PxP operation.

The PxP DMA driver also implements the interface `tx_submit in struct dma_async_tx_descriptor`, which is used to prepare the descriptor(s), which will be executed by the engine. When tasks are received in `pxp_tx_submit`, they are not configured and executed immediately. Instead, they are added to a task queue and the function call is allowed to return immediately.

6.3.3.4 Channel management

Although ePxP does not have multiple channels in hardware, the virtual channels are supported in the driver. This provides flexibility in the multiple instance/client design. At any time, a user can call `dma_request_channel()` to get a free channel, and then configure this channel with several descriptors. A descriptor is required for each input plane and for the output plane. When the PxP is no longer being used, the channel should be released by calling `dma_release_channel()`. Detailed elements of channel management are handled by the driver and are transparent to the client.

6.3.3.5 Descriptor management

The DMA Engine processes the task based on the descriptor. One DMA channel is usually associated with several descriptors. Descriptors are recycled resources, under the control of the offload engine driver, to be reused as operations complete. The extended TX descriptor packet (`pxp_tx_desc`), allows the user to pass the PxP configuration information to the driver. This includes everything that the PxP needs to execute a processing task.

6.3.3.6 Completion notification

There are two ways for an application to receive notification that a PxP operation has completed.

- Call `dma_wait_for_async_tx()`. This call causes the CPU to spin while it polls for the completion of the operation.
- Specify a completion callback.

The latter method is recommended. After the PxP operation completes, the PxP output buffer data can be retrieved.

For general information for the DMA Engine Framework, see `Documentation/dmaengine.txt` in the Linux kernel source tree.

6.3.3.7 Limitations

- The driver currently does not support scatterlist objects in the way they are traditionally used. Instead of using the scatterlist parameter object to provide a chain of memory sources and destinations, the driver currently uses it to provide the input and output buffers (and overlay buffers, if needed) for one transfer.
- The PxP driver may not properly execute a series of transfers that is queued in rapid sequence. It is recommended to wait for each transfer to complete before submitting a new one.

6.3.3.8 Menu configuration options

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

- For i.MX 7Dual, i.MX 8ULP, and i.MX 93, select **Device Drivers** -> **DMA Engine support** -> **[*] MXC PxP V3 support** -> **[*] MXC PxP Client Device**.

- For i.MX 6, select **Device Drivers** -> **DMA Engine support** -> **[*] MXC PxP V2 support** -> **[*] MXC PxP Client Device**.

6.3.3.9 Source code structure

The PxP driver source code is located in `drivers/dma/pxp`.

Table 67. PxP source

File	Description
`drivers/dma/pxp/pxp_device.c`	PxP Device
`drivers/dma/pxp/pxp_dma_v2.c`	PxP DMA for i.MX 6
`drivers/dma/pxp/pxp_dma_v3.c`	PxP DMA for i.MX 7, i.MX 8ULP, i.MX 93, i.MX 943
`drivers/dma/pxp/regs-pxp_v2.h`	PxP registers for i.MX 6
`drivers/dma/pxp/regs-pxp_v3.h`	Pxp registers for i.MX 7, i.MX 8ULP, i.MX 93, i.MX 943
`include/linux/pxp_dma.h`	PxP DMA kernel header
`include/linux/pxp_device.h`	PxP Device kernel header
`include/uapi/linux/pxp_dma.h`	PxP DMA user space header
`include/uapi/linux/pxp_device.h`	PxP Device user space header
`drivers/media/platform/mxc/output/mxc_pxp_v4l2.c`	PxP V4L2 driver for i.MX 6 and i.MX 7
`drivers/media/platform/mxc/output/mxc_pxp_v4l2.h`	PxP V4L2 header

6.3.4 eLCDIF frame buffer

6.3.4.1 Introduction

The eLCDIF is a display controller that works with the Parallel LCD interface. The driver is implemented as a display subsystem driver either frame buffer or DRM, which controls generic LCD low-level operations allowing low level hardware control. Only DOTCLK mode of the ELCDIF is tested, so theoretically the ELCDIF frame buffer driver can work with a synchronous LCD panel driver to support a frame buffer device. The synchronous LCD driver is organized in a flexible and extensible manner and is abstracted from any specific synchronous LCD panel support. To support another synchronous LCD panel, the user can write a synchronous LCD driver by referring to the existing ones.

6.3.4.2 Software operation

For the eLCDIF implemented as a frame buffer driver, the frame buffer device is a memory device similar to `/dev/mem`. It can be read from, written to, or some location in it can be sought and mapped using `mmap()`. The difference is that the memory available to the user is not the entire allocated memory, but only the frame buffer of the video hardware. The device is accessed through special device nodes, usually located in the `/dev` directory, `/dev/fb*`. `/dev/fb*` also has several IOCTLs, which act on it and through which information about the hardware can be queried and set. The color map handling operates through IOCTLs as well. See `linux/fb.h` for more information on which IOCTLs there are and which data structures are used.

The i.MX ELCDIF frame buffer driver implementation is abstracted from the actual hardware. The default panel driver is picked up by video mode defined in platform data or passed in with the `video=mxc_elcdif_fb:resolution, bpp=bits_per_pixel kernel bootup` command during probing. The resolution should be in the common frame buffer video mode pattern and `bits_per_pixel` should be the frame buffer's color depth.

6.3.4.3 Menu configuration options

The following menu options are used to configure the MXC ELCDIF frame buffer driver. This option depends on FB and `ARCH_MXS || ARCH_MXC`.

Frame buffer Devices -> **MXS LCD framebuffer support (CONFIG_FB_MXS)**

6.3.4.4 Source code structure

The source for frame buffer is in `drivres/video/fbdev/mxc` and the DRM driver is in `drivers/gpu/drm/imx/lcdif` and `drivers/gpu/drm/imx/lcdifv3`.

Table 68. ELCIF source

File	Description
`drivers/video/fbdev/mxsfb.c`	ELCDIF frame buffer driver
`drivers/video/fbdev/mxc/mxc_lcdif.c`	ELCDIF frame buffer driver
`drivers/gpu/drm/imx/lcdif/lcdif-crtc.c`	ELCDIF DRM Authentication
`drivers/gpu/drm/imx/lcdif/lcdif-kms.c`	ELCDIF DRM KMS
`drivers/gpu/drm/imx/lcdif/lcdif-kms.h`	ELCDIF DRM KMS Header
`drivers/gpu/drm/imx/lcdif/lcdif-plane.c`	ELCDIF DRM Plane
`drivers/gpu/drm/imx/lcdif/lcdif-plane.h`	ELCDIF DRM Plane header
`drivers/gpu/drm/lcdifv3/lcdifv3-crtc.c`	LCDIFv3 DRM CRTC
`drivers/gpu/drm/lcdifv3/lcdifv3-kms.c`	LCDIFv3 DRM KMS
`drivers/gpu/drm/lcdifv3/lcdifv3-kms.h`	LCDIFv3 DRM KMS Header
`drivers/gpu/drm/lcdifv3/lcdifv3-plane.c`	LCDIFv3 DRM Plane
`drivers/gpu/drm/lcdifv3/lcdifv3-plane.h`	LCDIFv3 DRM Plane header

6.3.5 Display Control Subsystem (DCSS)

6.3.5.1 Introduction

The Display control subsystem (DCSS) is a display control for i.MX 8M Quad that integrates through the DRM display framework. The DCSS provides a mechanism to display frame buffers in memory out to UltraHD or HDTVs with the capability to combine up to 3 layers of graphics or video overlay to the HDMI output. The key featuers of the DCSS controller include:

- Supports up to 3 layers of graphics or video
  - Arbitrary offset
  - One plane can be graphics with 8 bit alpha support
  - Upscale 1920 x 1080p60 video or graphics to 3840 x 2160p60
  - Downscale 3840 x 2160p30 video to 1920 x 1080p30 or 1280 x 720p30
- HDR support:
  - HDR10 with 2084 and 2020 color spaces
  - Dolby Vision single and dual layer formats

  - HLG
- HDMI 2.0a supporting one display:
  - Resolutions: 640 x 480p60, 720 x 480p60, 1280 x 720p60, 1920 x 1080p60, 3840 x 2160p60
  - HDCP 2.2 and HDCP 1.4
- Pixel clock up to 596 MHz
- Output can also go to MIPI DSI output
- Frame Buffer Compression: Lossless compression of buffers

6.3.5.2 Source code structure

The DCSS drm driver is located in `drivers/gpu/drm/imx/dcss` and the DCSS core driver is in `drivers/gpu/imx/dcss`.

Table 69. DCSS driver source

File	Description
`drivers/gpu/drm/imx/dcss/dcss-plane`	DRM DCSS Plane
`drivers/gpu/drm/imx/dcss/dcss-kms`	DRM DCSS KMS
`drivers/gpu/drm/imx/dcss/dcss-crtc`	DRM DCSS CRTC header
`drivers/gpu/drm/imx/dcss/dcss-dec400d.c`	DCSS DEC400D
`drivers/gpu/drm/imx/dcss/dcss-scaler`	DCSS Scaler
`drivers/gpu/drm/imx/dcss/dcss-ss.c`	DCSS SS
`drivers/gpu/drm/imx/dcss/dcss-hdr10.c`	DCSS HDR10
`drivers/gpu/drm/imx/dcss/dcss-wtsc1.c`	DCSS WTSC1
`drivers/gpu/drm/imx/dcss/dcss-dtg.c`	DCSS DTG
`drivers/gpu/drm/imx/dcss/dcss-ctx1d.c`	DCSS CTX1D
`drivers/gpu/drm/imx/dcss/dcss-dtrc.c`	DCSS DTRC
`drivers/gpu/drm/imx/dcss/dcss-dpr.c`	DCSS CTX1D
`drivers/gpu/drm/imx/dcss/dcss-blkctr.c`	DCSS CTX1D

6.3.6 DCNANO

6.3.6.1 Introduction

The LCDIF is a high-performance graphics core that can be used for reading rendered images from the frame buffer. In addition to providing hardware cursor patterns, the display controller performs format conversions, dithering, and gamma corrections.

Display controller key features are:

- Video Timing Generation
  - HSYNC, VSYNC, DE signals
  - Programmable timers
- MIPI Display Protocols
  - Display Pixel Interface-2 (DPI-2) formats
  - DPI 24-bit, 18-bit (2 configs), and 16-bit support (3 configs)

  - (Optional) Display Bus Interface 2.0 (DBI-2)
- Display Interface
  - Parallel Pixel Output with 24-bit Data, HSync, VSync, Data enable
  - Easily adaptable to external serialization logic, e.g., HDMI
- Display
  - Display sizes to 1024x480
  - Synchronous and blank signals
  - Gamma and dither tables
- Input Formats
  - ARGB2101010/ARGB8888/ARGB1555/RGB565/ARGB4444
  - YUV422 packed and semi planar (YUY2, UYVY)
- Format Conversion
  - Pixel inputs accepted from multiple RGB formats
  - Pixel output is 24 bit RGB in multiple formats
- Output Formats
  - RGB888/DPI_D16CFG1/DPI_D16CFG2/DPI_D16CFG3/DPI_D18CFG1/ DPI_D18CFG2/ DPI_D24
- Hardware Cursor
  - Supports ARGB888 and Mask cursor formats
- Color
  - Overlay with coordinate generator
  - Alpha Blending: 8 Porter Duff Blending modes
- Dither and Gamma Correction
  - A separate Look Up Table for Dither
  - A separate Look Up Table for Gamma Correction

6.3.6.2 Source code structure

The DCNANO DRM driver is located in `drivers/gpu/drm/imx/dcnano`.

Table 70. DCNANO driver source

File	Description
`drivers/gpu/drm/imx/dcnano/dcnano-crtc.c`	DRM DCNANO CRTC
`drivers/gpu/drm/imx/dcnano/dcnano-drv.c`	DRM DCNANO core
`drivers/gpu/drm/imx/dcnano/dcnano-drv.h`	DRM DCNANO header
`drivers/gpu/drm/imx/dcnano/dcnano-kms.c`	DRM DCNANO KMS
`drivers/gpu/drm/imx/dcnano/dcnano-plane.c`	DRM DCNANO plane
`drivers/gpu/drm/imx/dcnano/dcnano-reg.h`	DCNANO register header

6.4 Display interfaces

6.4.1 Parallel LCD interface

6.4.1.1 Introduction

The Parallel interface supports display to LCDs. The Parallel Display interface is supported through the display controllers and implemented using the display framework, which is the `fbdev` framework on i.MX 6 and i.MX 7 and the DRM framework for i.MX8.

The following controllers support the parallel interface:

- IPU on i.MX with IPU
- DPU on all i.MX 8
- ElCDIF on i.MX with PxP

The Parallel interface supports at least one port on the i.MX SoC that enables the parallel interface and supports two ports for i.MX with IPU. The enabled SoC has various bitrates from 18 bits to 24 bits per port. On i.MX 6 with IPU, the Parallel interface also supports a synchronous mode for display refresh and asynchronous mode to memory and is very flexible with a glue-less connection to RAM-less displays, display controllers, and TV encoders.

6.4.2 MIPI DSI interface

6.4.2.1 Introduction

The MIPI Display Interface (MIPI DSI) is a driver interface used to communicate with MIPI device controller on the display panel. The MIPI DSI display panel driver provides an interface to configure the display panel through MIPI DSI.

The MIPI DSI Interface is a digital core accompanied with a multi-lane D-PHY that implements all protocol functions defined in the MIPI DSI Specification, providing an interface between the System and MIPI DSI-compliant Display. The MIPI DSI overview can be found here. However, specifications are only available to MIPI members.

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 parallel bus. It is designed to be compatible with the standard MIPI DSI protocol and is built on the existing MIPI DPI-2, MIPI DBI-2, and MIPI DCS standards. The module 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 write register and memory to the display controller while reading display module status information. It also uses video mode to transmit a real-time pixel streams from the host to peripheral in high-speed mode and generates an interrupt when an error occurs.

For i.MX MIPI DSI is supported by various drivers, which are described in the following sections. The MIPI DSI drivers support the following features:

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

The MIPI DSI driver used frame buffer driver for i.MX 6 and i.MX 7, and the DRM driver for i.MX 8 and i.MX 93. Both drivers support the following.

- Drivers are not exposed to the user interface but through the drm or frame buffer interface.
- 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.

The driver enables the platform-related regulators and clock. It requests OS-related system resources and registers buffer event notifier for blank/unblank operation. 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, Version1.01.0r11.
- Compatibility with MIPI Alliance Specification for D-PHY, Version 1.00.00.

- Supports 1 to 4 D-PHY data lanes depending on SoC capabilities.
- Bidirectional Communication and Escape Mode Support through Data Lane 0.
- Programmable display resolutions.
- Video Mode Pixel Formats, 16bpp (565 RGB),18bpp (666 RGB) packed, 18 bpp (666 RGB) loosely, 24bpp (888 RGB).
- Supports the transmission of all generic commands.
- Supports ECC and checksum capabilities.
- End-of-Transmission Packet (EoTp) support.
- Supports ultra-low power mode.
- Supports PMS control interface for PLL to configure byte clock frequency.
- Supports Prescaler to generate escape clock from a byte clock.

The number of ports and lanes are specified in the device trees located in `arch/arm/boot/dts` and `arch/arm64/boot/dts`.

6.4.2.2 Software operation

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

The MIPI DSI IP driver has a private structure called `mipi_dsi_info`. The instance to which the MIPI DSI IP is attached is described in field `int dev_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 frame buffer driver through the field `struct mxc_dispdrv_handle` when the driver is loaded. It also registers a frame buffer event notifier with the frame buffer 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 information 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 frame buffer blank/unblank event, the registered notifier will be called to enter/leave low power mode.

The MIPI DSI IP driver provides 3 APIs for the MIPI DSI display panel driver to configure the display module.

The driver 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 power-on reset, the MIPI DSI display panel needs to be configured through standard MIPI DCS command or MIPI DSI Generic command according to the manufacturer's specification.

6.4.2.3 Source code structure

The table below shows the MIPI DSI driver source files available in `drivers/video/fbdev/mxc`.

Table 71. MIPI DSI Driver Files

File	Description
`drveirs/video/fbdev/mxc/mipi_dsi.c`	MIPI DSI IP Frame buffer driver source file
`drivers/video/fbdev/mxc/mipi_dsi.h`	MIPI DSI IP Frame bufferdriver header file
`drivers/video/fbdev/mxc/mxcfb_hx8369_wvga.c`	MIPI DSI Frame buffer Display Panel driver source file
`drivers/video/fbdev/mxc/mipi_dsi_samsung.c`	MIPI DSI Frame buffer Samsung source file
`drivers/video/fbdev/mxc/mipi_dsi_northwest.c`	MIPI DSI Frame buffer Northwest source file
`drivers/video/fbdev/mxc/mxcfb_hx8363_wvga.c`	i.MX 7 Frame buffer Truly WVGA Panel TFT3P5581E

Table 71. MIPI DSI Driver Files*...continued*

File	Description
`drivers/video/fbdev/mxc/mxcfb_hx8369_wvga.c`	i.MX 6 Frame buffer Truly WVGA sync panel
`drivers/video/fbdev/mxc/mxcfb_otm808b_wvga.c`	Truly Frame buffer WVGA Panel TFT3P5079E
`drivers/gpu/drm/imx/sec_mipi_dsmi-imx.c`	Samsung DRM driver
`drivers/gpu/drm/imx/nwl_dsi-imx.c`	Northwest DRM driver
`drivers/gpu/drm/imx/dw_mipi_dsi-imx.c`	Synopsys DesignWare MIPI DSI DRM driver

6.4.2.4 Menu configuration options

In menu configuration, enable the following modules:

- **Device Drivers** -> **Graphics support** -> **MXC Framebuffer support** -> **Synchronous Panel Framebuffer** -> **MXC MIPI_DSI**
- **Device Drivers** -> **Graphics support** -> **MXC Framebuffer support** -> **Synchronous Panel Framebuffer** -> **MXC MIPI_DSI_SAMSUNG**
- **Device Drivers** -> **Graphics support** -> **DRM Support for Freescale i.MX** -> **Support for Northwest Logic MIPI DSI displays**
- **Device Drivers** -> **Graphics support** -> **DRM Support for Freescale i.MX** -> **Support for Samsung MIPI DSIM displays**
- **Device Drivers** -> **Graphics support** -> **DRM Support for Freescale i.MX** -> **Freescale i.MX DRM Synopsys DesignWare MIPI DSI**

6.4.3 LVDS interface

6.4.3.1 Introduction

Low Voltage Differential Signaling (LVDS) supports high bandwidth and high definition graphics and fast frame rate with lower power consumption. The implementation uses pairs of wires where each wire in the pair carries the inverse signal of the other. This creates less interference and noise. The LVDS interface uses four, six, or eight pairs of wires with additional ones carrying clock and ground wires.

The purpose of the LVDS interface is to support the flow of synchronous RGB data from the display controller 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. Data arrangement required by the external display receiver and by LVDS display standards.
3. Synchronization and control capabilities.

The LVDS interface supports multiple controllers listed below:

- LDB: double on i.MX 6 with IPU
- Mixel on i.MX 8QuadMax
- Mixel Combo on i.MX 8QuadXPlus

The LVDS driver works with the supported display framework, which is frame buffer for i.MX 6 and i.MX 7, and DRM for i.MX 8 and i.MX 93.

The LVDS interface has the following structure of support:

- Channels: usually 2 channels
- Each channel supports a number of data pairs

- Data pixel rate, which can vary on each data pair
- Control signals for HSYNC, VSYNC, DE

The LVDS interface supports the following displays:

- IT6263 LVDS to HDMI bridge: implemented with our LDB driver
- LVDS dual-channel panel

The relevant standards for LVDS are the following:

- PHY Standard: ANSI EIA-644A
- Display Protocol Standards:
  - SPWG: Standard Panel Working Group Specification 3.8 (May 2007)
  - VESA PSWG: Panel Standardization Working Group – set of standards for panels using LVDS.
  - JEIDA/JEITA DISM Standard JEIDA-59-1999
  - OpenLDI (National): Revision 0.95 13/May/1999. Only unbalanced operating mode supported (aligned with the vast majority of LCD vendors).

The LVDS interface is supported through the frame buffer framework on i.MX 6 and i.MX 7, and the DRM framework on i.MX 8 and i.MX 93.

6.4.3.2 Software operation

The LVDS driver is functional if the driver is built in and the device tree status is set to “okay”.

When the LVDS device driver is probed properly, the driver configures the clocks for the LVDS. The LVDS driver probe function sets the default mode to 1080p60. The LVDS channel mapping mode and bit mapping mode are set to use 30-bit JEIDA mode.

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

1. Enable the power to the LVDS.
2. Set LDB_DI_CLK's parent CLK and the parent CLK's rate.
3. Set LDB_DI_CLK's rate.
4. Enable both LDB_DI_CLK and its parent CLK.
5. Set the LVDS in a proper mode including the polarities of the display signals, channel mapping mode, and bit mapping mode.
6. Enable the related i.MX LVDS channels.

6.4.3.3 Source code structure

Table 72. LVDS Source

File	Description
`drivers/gpu/drm/imx/imx*-ldb.c`	LDB driver with i.MX SoC information
`drivers/gpu/drm/bridge/fsl-imx-ldb.c`	LDB bridge driver

6.4.3.4 Menu configuration options

In menu configuration, enable the following modules:

Device Drivers -> **Graphics support** -> **DRM Support for Freescale i.MX** -> **Support for LVDS displays**

6.4.4 LVDS Display Bridge (LDB)

6.4.4.1 Introduction

This section describes the LVDS Display Bridge (LDB) driver, which controls the LDB module to connect with the external display devices with the LVDS interface. The purpose of the LDB is to support the flow of synchronous RGB data from IPU or LCDIF to external display devices through the LVDS interface.

This support covers the following:

- Connectivity to relevant devices: Displays with LVDS receivers.
- Arranging data as required by the external display receiver and by LVDS display standards.
- Synchronization and control capabilities.

6.4.4.2 Software operation

The LDB driver is functional if the driver is built in.

When the LDB device is probed properly, the driver configures the LDB reference resistor mode and the LDB regulator by using platform data information. The LDB driver probe function tries to match video modes for external display devices to LVDS interface. The display signal polarities control bits of the LDB are set according to the matched video modes. LVDS channel-mapping mode and bit mapping mode of the LDB are set according to the LDB device tree node set by the user. The LDB is fully enabled in the probe function if the driver identifies a display device with an LVDS interface as the primary display device.

The steps the driver takes to enable an LVDS channel are:

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

6.4.4.3 Source code structure

Table 73. LDB Source

File	Description
`drivers/video/fbdev/mxc/ldb.c`	LDB frame buffer driver

6.4.4.4 Menu configuration options

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

In menu configuration, enable the following module:

Device Drivers -> **Graphics support** -> **MXC Framebufer support** -> **Synchronous Panel Framebuffer** -> **MXC LDB**

6.4.5 Electrophoretic Display Controller (EPDC) Interface

6.4.5.1 Introduction

The Electrophoretic Display Controller (EPDC) is a direct-drive active matrix EPD controller designed to drive E Ink EPD panels supporting a wide variety of TFT backplanes. The EPDC frame buffer driver acts as a standard

Linux frame buffer device. This driver supports a set of custom API extensions, accessible from user space (via IOCTL) or another kernel module (via direct function call) to provide the user with access to EPD-specific functionality. The EPDC driver is abstracted from any specific E Ink panel type, providing flexibility to work with a range of E Ink panel types and specifications.

The EPDC driver supports the following features:

- EPDC driver as a loadable or built-in module.
- RGB565, RGB24, RGB32, and Y8 frame buffer formats.
- Full and partial EPD screen updates.
- Up to 256 panel-specific waveform modes.
- Automatic optimal waveform selection for a given update.
- Synchronization by waiting for a specific update request to complete.
- Screen updates from an alternate (overlay) buffer.
- Automated collision handling.
- 64 simultaneous update regions.
- Pixel inversion in a Y8 frame buffer format.
- 90 degree, 180 degree, and 270 degree hardware-accelerated frame buffer rotation.
- Panning (y-direction only).
- Automated full and partial screen updates through the Linux `fb_deferred_io` mechanism.
- Three EPDC driver display update schemes: Snapshot, Queue, and Queue and Merge.
- Setting the ambient temperature through either a one-time designated API call or on a per-update basis.
- User control of the delay between completing all updates and powering down the EPDC.

6.4.5.2 EPDC frame buffer driver overview

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, abstracting from software how to manage the low-level hardware registers. The EPDC driver supports this model with one key caveat: The contents of the frame buffer are not automatically updated to the E Ink display. Instead, a custom API function call is required to trigger an update to the E Ink display. The details of this process are described in the Section 6.4.5.3.

The frame buffer 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 includes support for fonts and a startup logo. The frame buffer 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.

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

The EPDC frame buffer driver (`drivers/video/fbdev/mxc/mxc_epdc_fb.c` on i.MX 6DualLite or `drivers/video/fbdev/mxc/mxc_epdc_v2_fb.c` for generation-II EPDC on i.MX 7Dual) interacts closely with the generic Linux frame buffer driver (`drivers/video/fbmem.c`).

For more details on the frame buffer device, see the documentation in the Linux kernel found in `Documentation/fb/framebuffer.txt`.

6.4.5.3 EPDC frame buffer driver extensions

E Ink display technology, in conjunction with the EPDC, has several features that distinguish it from standard LCD-based frame buffer devices. These differences introduce the need for API extensions to the frame buffer interface. The EPDC refreshes the E Ink display asynchronously and supports partial screen updates. Therefore, the EPDC requires notification from the user when the frame buffer contents have been modified and which region needs to update. Another unique characteristic of EPDC updates to the E Ink display is the long screen update latencies (between 300-980 ms), which introduces the need for a mechanism to allow the user to wait for a given screen update to complete.

The custom API extensions to the frame buffer device are accessible both from user space applications and from within kernel space. The standard device IOCTL interface provides access to the custom API for user space applications. The IOCTL extensions, along with relevant data structures and definitions, can be found in `include/linux/mxcfb_epdc.h`. A full description of these IOCTLs can be found in Section 6.4.5.11.

For kernel mode access to the custom API extensions, the IOCTL interface should be bypassed in favor of direct access to the underlying functions.

6.4.5.4 EPDC panel configuration

The EPDC driver is designed to flexibly support E Ink panels with various panel resolutions, timing parameters, and waveform modes. The EPDC driver is kept panel-agnostic by using an EPDC panel mode structure, `imx_epdc_fb_mode`, which can be found in `include/linux/mxcfb_epdc.h`.

```
struct imx_epdc_fb_mode {
struct fb_videomode *vmode;
int vscan_holdoff;
int sdoed_width;
int sdoed_delay;
int sdoez_width;
int sdoez_delay;
int gdclk_hp_offs;
int gdsp_offs;
int gdoe_offs;
int gdclk_offs;
int num_ce;
};
```

The `imx_epdc_fb_mode` structure consists of an `fb_videomode` structure reference and a set of EPD timing parameters. The `fb_videomode` structure defines the panel resolution and the basic timing parameters (pixel clock frequency, HSYNC and VSYNC margins) and the additional timing parameters in `imx_epdc_fb_mode` define EPD-specific timing parameters, such as the source and gate driver timings. For details on how to configure E Ink panel timing parameters, see the EPDC programming model section in the *i.MX 6DualLite Applications Processor Reference Manual* (IMX6DLRM) or *i.MX 7Dual Applications Processor Reference Manual* (IMX7DRM).

In addition to the EPDC panel mode data, functions may be passed to the EPDC driver to define how to handle the EPDC pins when the EPDC driver is enabled or disabled. These functions should disable the EPDC pins for purposes of power savings.

6.4.5.5 Boot command line parameters

Additional configuration for the EPDC driver is provided through boot command line parameters. The format of the command line option is as follows:

```
epdc video=mxcepdcfb:[panel_name],bpp=16
```

The EPDC driver parses these options and tries to match `panel_name` to the name of video mode specified in the `imx_epdc_fb_mode` panel mode structure. If no match is found, the first panel mode provided in the platform data is used by the EPDC driver. The `bpp` setting from this command line sets the initial bits per pixel setting for the frame buffer. A setting of 32 or 24 selects the RGB888 pixel format, and one of 16 selects the RGB565 pixel format, while a setting of 8 selects the 8-bit grayscale (Y8) format.

6.4.5.6 EPDC waveform loading

The EPDC driver requires a waveform file for proper operation. This waveform file contains the waveform information needed to generate the waveforms that drive updates to the E Ink panel. A pointer to the waveform file data is programmed into the EPDC before the first update is performed.

There are two options for selecting a waveform file:

1. Select one of the default waveform files included in this BSP release.
2. Use a new waveform file that is specific to the E Ink panel being used.

The waveform file is loaded by the EPDC driver using the Linux firmware APIs.

6.4.5.7 Using a default waveform file

The quickest and easiest way to get started using an E Ink panel and the EPDC driver is to use one of the default waveform files provided in the Linux BSP. This should enable updates to several different types of E Ink panel without a panel-specific waveform file. The drawback is that optimal quality should not be expected. Typically, using a non-panel-specific waveform file for an E Ink panel results in more ghosting artifacts and overall poorer color quality.

The following default waveform files included in the BSP reside in `/lib/firmware/imx/epdc`:

- `epdc_E60_V110.fw`: Default waveform for the 6.0 inch V110 E Ink panel.
- `epdc_E60_V220.fw`: Default waveform for the 6.0 inch V220 E Ink panel (supports animation mode updates).
- `epdc_E97_V110.fw`: Default waveform for the 9.7 inch V110 E Ink panel.
- `epdc_E060SCM.fw`: Default waveform for the 6.0 inch Pearl E Ink panel (supports animation mode updates).
- `epdc_ED060XH2C1.fw`: Default waveform for the 6.0 inch E Ink panel (No Reagl/-D Support by default. For Reagl/-D support, contact NXP support.)

The EPDC driver attempts to load a waveform file with the name `epdc_[panel_name].fw` under the directory `/lib/firmware/imx/epdc` in rootfs, where `panel_name` refers to the string specified in the `name` field of `fb_videomode`. This `panel_name` information should be provided to the EPDC driver through the kernel command line parameters described in the preceding section. For example, to load the `epdc_E060SCM.fw` default firmware file for a Pearl panel, set the EPDC kernel command line parameter to the following:

```
video=mxcepdcfb:E060SCM,bpp=16
```

6.4.5.8 Using a custom waveform file

To ensure the optimal E Ink display quality, use a waveform file specific to the E Ink panel being used. The raw waveform file type (`.wbf`) requires conversion to a format that can be understood and read by the EPDC. This conversion script is not included as part of the BSP. Therefore, contact NXP to acquire this conversion script.

Once the waveform conversion script has been run on the raw waveform file, the converted waveform file should be renamed so that the EPDC driver can find it and load it. The driver is going to search for a waveform file with the name `epdc_[panel_name].fw` under the directory `/lib/firmware/imx/epdc` in rootfs, where `panel_name` refers to the string specified in the `name` field of `fb_videomode`. For example, if the panel is named `E60_ABCD`, the converted waveform file should be named `epdc_E60_ABCD.fw`.

Note: *If the EPDC driver searches for a firmware waveform file that matches the names of one of the default waveform files (see the preceding section), it chooses the default firmware files that are built into the BSP over* *any firmware file that has been added in the firmware search path. Therefore, if you leave the BSP so that it uses the default firmware files, make sure to use a panel name other than those associated with the default firmware files, as those default waveform files will be preferred and selected over a new waveform file placed in the firmware search path.*

6.4.5.9 EPDC panel initialization

The frame buffer driver does not typically (see note below for exceptions) go through any hardware initialization steps when the frame buffer driver module is loaded. Instead, a subsequent user mode call must be made to request that the driver initialize itself for a specific EPD panel. To initialize the EPDC hardware and E Ink panel, an `FBIOPUT_VSCREENINFO` IOCTL call must be made, with the `xres` and `yres` fields of the `fb_var_screeninfo` parameter set to match the X and Y resolution of a supported E Ink panel type. To ensure that the EPDC driver receives the initialization request, the activated field of the `fb_var_screeninfo` parameter should be set to `FB_ACTIVATE_FORCE`.

Note: *The exception is when the FB Console driver is included in the kernel. When the EPDC driver registers the frame buffer device, the FB Console driver will subsequently make an `FBIOPUT_VSCREENINFO` IOCTL call. This will in turn initialize the EPDC panel.*

6.4.5.10 Grayscale frame buffer selection

The EPDC frame buffer driver supports the use of 8-bit grayscale (Y8) and 8-bit inverted grayscale (Y8 inverted) pixel formats for the frame buffer (in addition to the more common RGB565 pixel format). To configure the frame buffer format as 8-bit grayscale, the application would call the `FBIOPUT_VSCREENINFO` frame buffer IOCTL. This IOCTL takes an `fb_var_screeninfo` pointer as a parameter. This parameter specifies the attributes of the frame buffer and allows the application to request changes to the frame buffer format. There are two key members of the `fb_var_screeninfo` parameter that must be set to request a change to 8-bit grayscale format: `bits_per_pixel` and `grayscale`. `bits_per_pixel` must be set to **8** and `grayscale` must be set to one of the two valid grayscale format values: `GRAYSCALE_8BIT` or `GRAYSCALE_8BIT_INVERTED`.

The following code snippet demonstrates a request to change the frame buffer to use the Y8 pixel format:

```
fb_screen_info screen_info;
screen_info.bits_per_pixel = 8;
screen_info.grayscale = GRAYSCALE_8BIT;
retval = ioctl(fd_fb0, FBIOPUT_VSCREENINFO, &screen_info);
```

6.4.5.11 Software operation

The EPDC frame buffer is accessible from the user space and kernel space. A single set of functions describes the EPDC frame buffer driver extension. There are two modes for accessing these functions with user space using the IOCTL interface and kernel space using functions directly. Each IOCTL and function combination is described as follows.

MXCFB_SET_WAVEFORM_MODES/`mxc_epdc_fb_set_waveform_modes()`

Description:

Defines a mapping for common waveform modes.

Parameters:

`mxcfb_waveform_modes *modes`

Pointer to a structure containing the waveform mode values for common waveform modes. These values must be configured in order for automatic waveform mode selection to function properly.

MXCFB_SET_TEMPERATURE/`mxc_epdc_fb_set_temperature`

Description:

Sets the temperature to be used by the EPDC driver in subsequent panel updates.

Parameters:

`int32_t temperature`

Temperature value, in degrees Celsius. This temperature setting may be overridden by setting the temperature value parameter to anything other than **TEMP_USE_AMBIENT** when using the **MXCFB_SEND_UPDATE** IOCTL.

MXCFB_SET_AUTO_UPDATE_MODE/`mxc_epdc_fb_set_auto_update`

Description:

Selects between automatic and region update mode.

Parameters:

`__u32 mode`

In region update mode, updates must be submitted through the **MXCFB_SEND_UPDATE** IOCTL.

In automatic mode, updates are generated automatically by the driver by detecting pages in the frame buffer memory region that have been modified.

MXCFB_SET_UPDATE_SCHEME/`mxc_epdc_fb_set_upd_scheme`

Description:

Selects a scheme that dictates how the flow of updates within the driver.

Parameters:

`__u32 scheme`

Select the following updates schemes:

- **UPDATE_SCHEME_SNAPSHOT**: In the Snapshot update scheme, the contents of the frame buffer are immediately processed and stored in a driver-internal memory buffer. By the time the call to **MXCFB_SEND_UPDATE** has completed, the frame buffer region is free and can be modified without affecting the integrity of the last update. If the update frame submission is delayed due to other pending updates, the original buffer contents are displayed when the update is finally submitted to the EPDC hardware. If the update results in a collision, the original update contents are resubmitted when the collision has cleared.
- **UPDATE_SCHEME_QUEUE**: The Queue update scheme uses a work queue to asynchronously handle the processing and submission of all updates. When an update is submitted through **MXCFB_SEND_UPDATE**, the update is added to the queue and then processed in order as EPDC hardware resources become available. As a result, the frame buffer contents processed and updated are not guaranteed to reflect what was present in the frame buffer when the update was sent to the driver.
- **UPDATE_SCHEME_QUEUE_AND_MERGE**: The Queue and Merge scheme uses the queueing concept from the Queue scheme, but adds a merging step. This means that, before an update is processed in the work queue, it is first compared with other pending updates. If any update matches the mode and flags of the current update and also overlaps the update region of the current update, that update is merged with the current update. After attempting to merge all pending updates, the final merged update is processed and submitted.

MXCFB_SEND_UPDATE/`mxc_epdc_fb_send_update`

Description:

Requests a region of the frame buffer be updated to the display.

Parameters:

`mxcfb_update_data *upd_data`

Pointer to a structure defining the region of the frame buffer, waveform mode, and collision mode for the current update. This structure also includes a flags field to select from one of the following update options:

EPDC_FLAG_ENABLE_INVERSION: Enables inversion of all pixels in the update region.

EPDC_FLAG_FORCE_MONOCHROME: Enables full black/white posterization of all pixels in the update region.

EPDC_FLAG_USE_ALT_BUFFER: Enables updating from an alternate (non-framebuffer) memory buffer.

If enabled, the final *upd_data* parameter includes detailed configuration information for the alternate memory buffer.

MXCFB_WAIT_FOR_UPDATE_COMPLETE/`mxc_epdc_fb_wait_update_complete`

Description:

Blocks and waits for a previous update request to complete.

Parameters:

`mxfb_update_marker_data marker_data`

The `update_marker` value used to identify a particular update (passed as a parameter in **MXCFB_SEND_UPDATE** IOCTL call) should be reused here to wait for the update to complete. If the update was a collision test update, the `collision_test` variable returns the result indicating whether a collision occurred.

MXCFB_SET_PWRDOWN_DELAY/`mxc_epdc_fb_set_pwrdown_delay`

Description:

Sets the delay between the completion of all updates in the driver and when the driver should power down the EPDC and the E Ink display power supplies.

Parameters:

`int32_t delay`

Input delay value in milliseconds. To disable EPDC power down altogether, use **FB_POWERDOWN_DISABLE** (defined below).

MXCFB_GET_PWRDOWN_DELAY/`mxc_epdc_fb_get_pwrdown_delay`

Description:

Retrieves the driver's current power down delay value.

Parameters:

`int32_t delay`

Output delay value in milliseconds.

6.4.5.12 Structures and defines

```
#define GRAYSCALE_8BIT
       0x1
#define GRAYSCALE_8BIT_INVERTED
       0x2
#define AUTO_UPDATE_MODE_REGION_MODE
       0
```

```
#define AUTO_UPDATE_MODE_AUTOMATIC_MODE
      1
#define UPDATE_SCHEME_SNAPSHOT
      0
#define UPDATE_SCHEME_QUEUE
      1
#define UPDATE_SCHEME_QUEUE_AND_MERGE
      2
#define UPDATE_MODE_PARTIAL
      0x0
#define UPDATE_MODE_FULL
      0x1
#define WAVEFORM_MODE_AUTO
      257
#define TEMP_USE_AMBIENT
      0x1000
#define EPDC_FLAG_ENABLE_INVERSION
      0x01
#define EPDC_FLAG_FORCE_MONOCHROME
      0x02
#define EPDC_FLAG_USE_ALT_BUFFER
      0x100
#define EPDC_FLAG_TEST_COLLISION
      0x200
#define FB_POWERDOWN_DISABLE
      -1
struct mxcfb_rect {
__u32 left; /* Starting X coordinate for update region */
__u32 top; /* Starting Y coordinate for update region */
__u32 width; /* Width of update region */
__u32 height; /* Height of update region */
};
struct mxcfb_waveform_modes {
int mode_init; /* INIT waveform mode */
int mode_du; /* DU waveform mode */
int mode_gc4; /* GC4 waveform mode */
int mode_gc8; /* GC8 waveform mode */
int mode_gc16; /* GC16 waveform mode */
int mode_gc32; /* GC32 waveform mode */
};
struct mxcfb_alt_buffer_data {
__u32 phys_addr; /* physical address of alternate image buffer */
__u32 width; /* width of entire buffer */
__u32 height; /* height of entire buffer */
struct mxcfb_rect alt_update_region; /* region within buffer to update */
};
struct mxcfb_update_data {
struct mxcfb_rect update_region; /* Rectangular update region bounds */
__u32 waveform_mode; /* Waveform mode for update */
__u32 update_mode; /* Update mode selection (partial/full) */
__u32 update_marker; /* Marker used when waiting for completion */
int temp; /* Temperature in Celsius */
uint flags; /* Select options for the current update */
struct mxcfb_alt_buffer_data alt_buffer_data; /* Alternate buffer data */
};
struct mxcfb_update_marker_data { __u32 update_marker; __u32 collision_test; };
```

6.4.5.13 Source code structure

The table below lists the source files associated with the EPDC driver and headers for programming access.

Table 74. EPDC source

File	Description
`drivers/video/fbev/mxc/mxc_epdc_v2_fb.c`	EPDC Generation-II V2 frame buffer driver for i.MX 7Dual
`drivers/video/fbdev/mxc/epdc_v2_regs.h`	EPDC Generation-II Register definition
`drivers/video/fbdev/mxc/mxc_epdc_fb.c`	Generation-I EPDC frame buffer driver for i.MX 6Sololite, 6SLL, and 6 DualLite
`drivers/video/fbdev/mxc/epdc_regs.h`	EPDC Generation-IRegister definitions
`drivers/video/fbdev/mxc/epdc_v2_regs.h`	Generation-II EPDC v2 register definitions
`include/linux/uapi/mxcfb.h`	Header file for the EPDC IOCTLs and frame buffer driver
`include/linux/mxcfb_epdc.h`	Header file for direct kernel access to the EPDC API extension

6.4.5.14 Menu configuration options

The following Linux kernel configuration options are provided for the EPDC module:

- **CONFIG_FB_MXC_EINK_PANEL**: support for the Electrophoretic Display Controller. In `menuconfig`, select **Device Drivers** -> **Graphics Support** -> **E-Ink Panel Framebuffer**.
- **CONFIG_FB_MXC_EINK_V2_PANEL**: support for v2 Electrophoretic Display Controller. In `menuconfig`, this option is available with **Device Drivers** -> **Graphics support** -> **E-Ink Panel Framebuffer based on EPDC V2**.
- **CONFIG_FB**: includes frame buffer support and is enabled by default. In `menuconfig`, select **Device Drivers** -> **Graphics support** -> **Support for frame buffer devices**.
- **CONFIG_MXC_PXP_V2**: support for the PxP and required by the EPDC driver for processing (color space conversion, rotation, auto-waveform selection) frame buffer update regions. In `menuconfig`, select **Device Drivers** -> **DMA Engine support** -> **MXC PxP support**.
- **CONFIG_MXC_PXP_V3**: support for next level PxP and required by Generation-II EPDC driver for processing frame buffer update regions. In `menuconfig`, select **Device Drivers** -> **DMA Engine support** -> **MXC PxP V3 support**.

6.4.6 High-Definition Multimedia Interface (HDMI) and Display Port (DP) overview

6.4.6.1 Introduction

High-Definition Multimedia Interface (HDMI) and Display Port (DP) present high definition video. The HDMI module is supported on some i.MX chips either with on-chip solutions or external solutions. The Display Port DP provides an embedded Display Port (eDP) Transmitter including HDMI Transmit (TX) Controller and PHY.

The following are compliance versions:

- HDMI 1.4 and 2.0
- DVI 1.0
- DP 1.3
- eDP 1.4
- HDCP 1.4/2.2

Each SoC HDMI solution is presented in separate chapters. The Display Port on i.MX uses the same IP block but has a different specification. The following table lists which SoCs support HDMI and Display Port and its supported version.

Table 75. HDMI Support

SoC	Features
i.MX 6QuadPlus/Quad/Dual	HDMI 1.4 on chip
i.MX 7ULP	HDMI 1.4 external chip
i.MX 8M Quad	HDMI 2.0/Display Port 1.3 on chip
i.MX 8QuadMax	HDMI 2.0/Display Port 1.3 on chip

HDMI Audio data source comes from S/PDIF TX.

6.4.6.2 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.

6.4.6.3 Core

The on-chip HDMI i.MX solutions support a core driver that 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:

- Map the HDMI register region and provide APIs for reading and writing to HDMI registers.
- Perform one-time initialization of key HDMI registers.
- Initialize the HDMI IRQ and provide shared APIs for enabling and disabling the IRQ.
- Provide a means for sharing information between the audio and video drivers (for example, the HDMI pixel clock).
- Provide a means for synchronization between HDMI video and HDMI audio while blank/unblank, plug in/plug out events happen. HDMI audio cannot start work while the HDMI cable is in the state of plug-out or HDMI is in a state of blank. Every time HDMI audio starts a playback, the HDMI audio driver should register its PCM into the core driver and unregister PCM when the playback is finished. Once an HDMI video blank or cable plug-out event happens, core driver would pause the HDMI audio DMA controller if its PCM is registered. When HDMI is unblanked or cable plug-in event happens, the core driver would first check if the cable is in the state of plug-in, the video state is unblank and the PCM is registered. If the items listed above are all yes, the core driver would restart HDMI audio DMA.

6.4.6.4 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 i.MX Frame Buffer 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 i.MX Frame Buffer 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 i.MX Frame Buffer Driver containing frame buffer information (`fbi`). The HDMI driver registers itself to receive notifications for FB driver events. Finally, the HDMI driver completes initialization by configuring the HDMI to receive a hotplug interrupt.

Note: *All display device drivers must be initialized before the i.MX Frame Buffer Driver for all display devices to be registered as MXC Display Driver devices.*

6.4.6.5 Hotplug handling and video mode changes

Once the connection between the i.MX Frame Buffer Driver 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 i.MX Frame Buffer Driver are conducted through the Linux Frame Buffer APIs. The following list demonstrates the software flow to recognize an HDMI sink device and configure the ELCDIF FB driver to drive video output:

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 i.MX Frame Buffer Driver. The i.MX Frame Buffer Driver completes its reconfiguration for the new mode.
3. As a result of calling `fb_set_var()`, a Frame Buffer 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. Finally, the HDMI module is enabled to generate output to the HDMI sink device.

The i.MX Frame Buffer Driver aligns to the display interface specific to each SoC as noted for each SoC HDMI section.

6.4.6.6 Audio

Since the HDMI TX audio driver uses the ALSA SoC framework, it is broken into several files as listed in the source code structure sections of each HDMI section. 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`) allocates the SoC audio device and links all the SoC components together. The DAI driver (`sound/soc/imx/imx-hdmi-dai.c`) is a SoC requirements. It is primarily used to get the platform data.

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, and so on). 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. Since the struct is hooked into the ALSA layer, IEC settings are accessible in the userspace using the `iecset` utility.

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

The HDMI audio playback depends on the HDMI pixel clock. Therefore, while in the state of HDMI blank and cable plug out, HDMI audio is either stopped or cannot be played. See detailed information in `software_operation_core`.

Because the HDMI audio driver needs to add the IEC header, the driver needs to know the amount of data already written into the HDMI audio driver. If the application is not able to decipher the amount of data written, for example, DMIX plugin in ALSA LIB, the HDMI audio driver is not able to work properly. There will be no sound heard.

The HDMI audio supports the features below:

- Playback sample rate:
  - 32k, 44.1k, 48k, 88.2k, 96k, 176.4k, 192k
  - HDMI sink capability
- Playback Channels:
  - 2, 4, 6, 8
  - HDMI sink capability
- Playback audio formats:
  - `SNDRV_PCM_FMTBIT_S16_LE`

6.4.6.7 i.MX 8 Display Port

6.4.6.7.1 Introduction

The High-Definition Multimedia Interface (HDMI) driver supports the on-chip Cadence HDTX IP module on the i.MX 8QuadMax and i.MX 8M Quad providing capability to transfer uncompressed video, audio, and data using a single cable. The HDMI driver is divided into three sub-components: A video display device driver that integrates with the DPU/DCSS DRM driver, an audio driver that integrates with the ALSA/SoC sub-system, and a core API driver, which manages the shared software and hardware resources of the HDMI driver.

HDTX IP supports the following features:

- Compliant with HDMI 2.0 Specification.
- Supports up to 600 MHz pixel CLK.
- All video formats are supported, including dual-vide, stereo, and all colorimetry options (RGB, YCbCr444/422, and YCbCr420).
- These audio formats are supported: PCM, HBR, DST, one-bit-audio, multi-stream, and 3D audio.
- All info-frames are supported.
- The APB interface is used to control and read status information.
- Embedded-CPU performs all protocol-specific tasks that simplify SoC integration:
  - HDCP 1.4/2.2
  - Audio Return Channel (ARC)

The HD Display TX Controller supports one or more of the protocols, such as HDMI, DisplayPort, or eDP. Each protocol requires a different firmware binary. The figure below describes this.

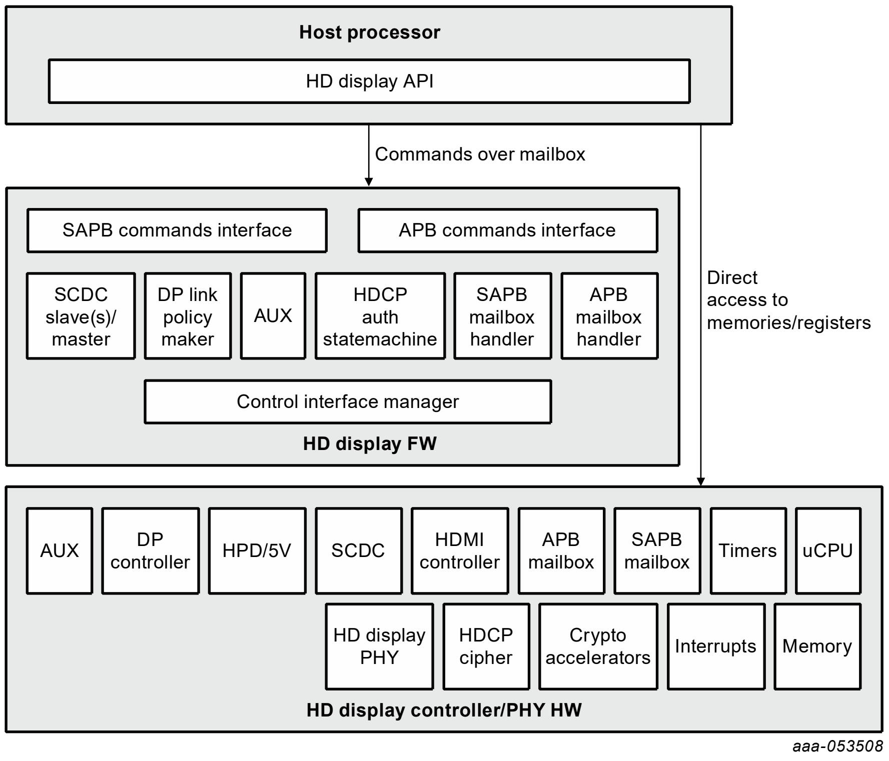

Figure 22. HDMI HW integration

The HD Display controller integrates a CPP (uCPU) running the embedded Firmware (FW). The firmware manages the HD Display link and provides side-band channel communication. The firmware is not involved in the data-path (video, audio, or info-frames).

The host processor interfaces to the HD Display controller over an APB-interface. The host processor manages the HD Display Controller in one or more of the following methods:

- Direct access to the HW registers for debugging purposes.
- Direct access to I-MEM and D-MEM (during boot) for FW download.
- Direct access to the HW registers of designated HW modules during operational mode (modules that are not controlled by the FW).
- Indirect access to the HW registers of designated HW modules during operational mode, by communicating with FW over the command interface (using `GENERAL_WRITE_REGISTER` and `GENERAL_READ_REGISTER` commands).
- Communication with different FW modules over a mailbox using the command interface.

6.4.6.7.2 Software operation

The HDMI driver is divided into sub-components based on its two primary purposes: providing HDP DRM driver and Core API driver.

The HDP DRM driver requires a Core API driver component to the configurated HDMI FW.

6.4.6.7.3 Source code structure

The HDMI driver has three software components:

- MHDP DRM Bridge and Core API driver
- MHDP i.MX 8 platform driver

- HDMI audio driver

Table 76. HDP Core API driver files

Files	Description
`drivers/gpu/drm/bridge/cadence`	MHDP DRM Bridge and Core API driver
`drivers/gpu/drm/imx/mhdp`	MHDP i.MX 8 platform driver
• `sound/soc/fsl/fsl_hdmi.c` • `sound/soc/fsl/imx-hdmi.c` • `sound/soc/fsl/hdmi_pcm.S` • `sound/soc/fsl/imx-hdmi-dma.c` • `sound/soc/fsl/imx-cdnhdmi.c`	HDMI Sound Driver

6.4.6.7.4 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_DRM_CDNS_MHDP** option provides support for the MHDP DRM Bridge and Core API driver, and can be selected in `menuconfig` at the following menu location:
  Device Drivers -> **Graphics support** -> **Display Interface Bridges** -> **Cadence MHDP COMMON API driver**
- The **CONFIG_DRM_IMX_CDNS_MHDP** option provides support for the i.MX 8 HDMI/DP video driver, and can be selected in `menuconfig` at the following menu location:
  Device Drivers -> **Graphics support** -> **NXP i.MX MX8 DRM HDMI/DP**
- The **CONFIG_SND_SOC_IMX_CDNHDMI** option provides support for HDMI audio through the ALSA/SoC subsystem, and can be found in `menuconfig` at the following location:
  Device Drivers -> **Sound card support** -> **Advanced Linux Sound Architecture** -> **ALSA for SoC audio support** -> **SoC Audio support for CDN - HDMI**

6.4.6.8 i.MX 6 On Chip High-Definition Multimedia Interface (HDMI)

6.4.6.8.1 Introduction

The High-Definition Multimedia Interface (HDMI) driver supports the on-chip DesignWare HDMI hardware module on the i.MX 6QuadPlus, 6Quad, and 6Dual SoC. This driver provides the capability to transfer uncompressed video, audio, and data using a single cable.

The HDMI driver is divided into four sub-components:

- Video display device driver that integrates with the Linux Frame Buffer API
- Audio driver that integrates with the ALSA/SoC sub-system
- CEC driver
- Multifunction 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 (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 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 using the `iecset` utility

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 figure below. Output data is transmitted via three Transition-Minimized Differential Signaling (TMDS) channels to an HDMI sink device external to the SoC. The HDMI also 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.

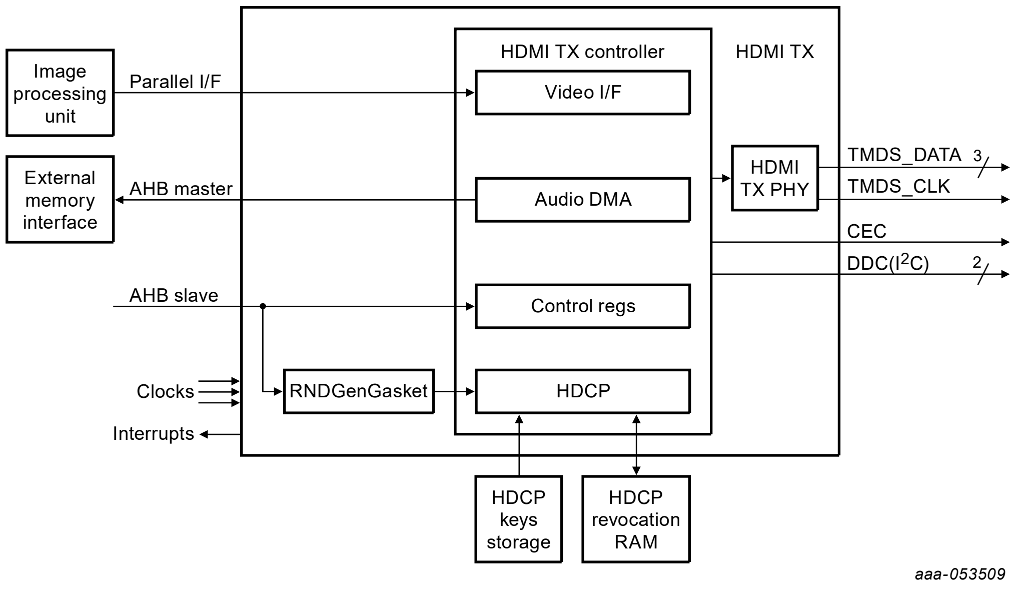

Figure 23. HDMI hardware integration

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

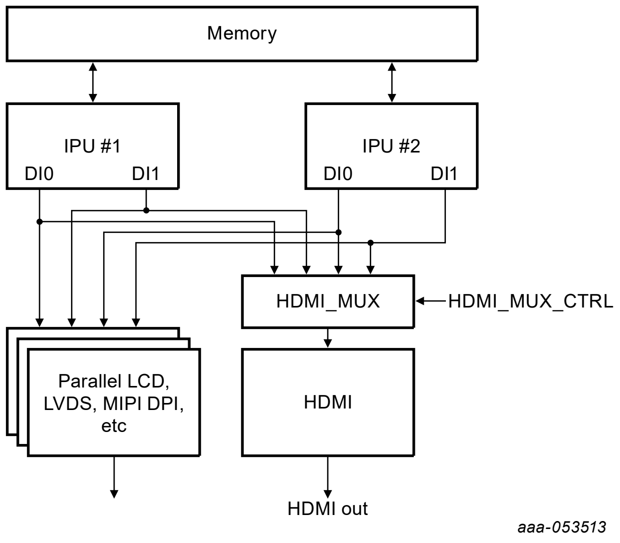

Figure 24. IPU-HDMI hardware interconnection

6.4.6.8.2 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. 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.

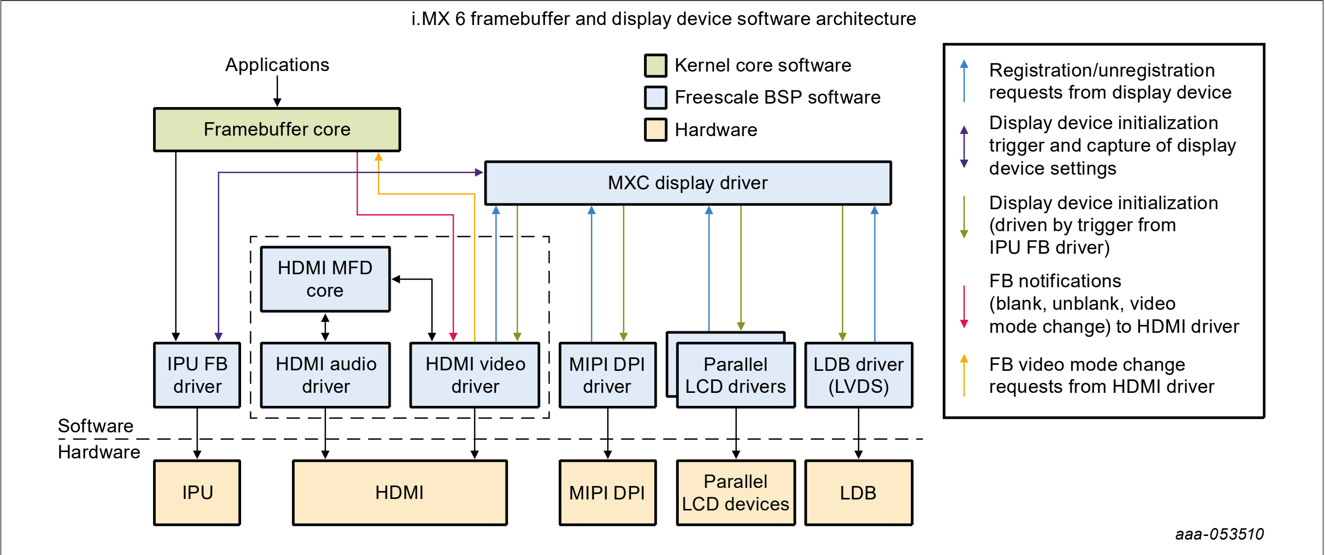
i.MX 6 framebuffer and display device software architecture
Applications
Framebuffer core
Kernel core software
Freescale BSP software
Hardware
MXC display driver
HDMI MFD core
IPU FB driver
HDMI audio driver
HDMI video driver
MIPI DPI driver
Parallel LCD drivers
LDB driver (LVDS)
Software
Hardware
IPU
HDMI
MIPI DPI
Parallel LCD devices
LDB
Registration/unregistration requests from display device
Display device initialization trigger and capture of display device settings
Display device initialization (driven by trigger from IPU FB driver)
FB notifications (blank, unblank, video mode change) to HDMI driver
FB video mode change requests from HDMI driver
aaa-053510

Figure 25. HDMI Video SW architecture

The i.MX 6Dual/6Quad/6QuadPlus/6Solo/6DualLite supports many different types of display output devices (for example, 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 using this API to receive the correct video input from the IPU.

6.4.6.8.3 CEC

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

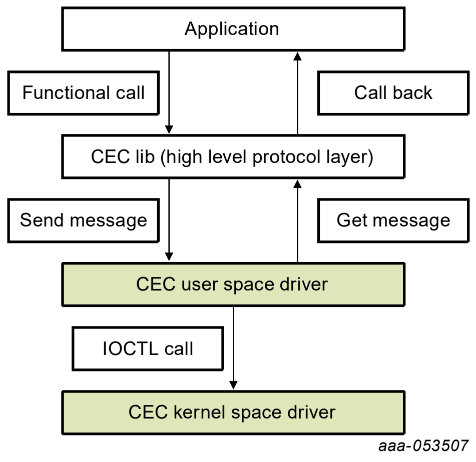

Figure 26. HDMI CEC SW architecture

6.4.6.8.4 Source code structure

The HDMI source code is provided in the HDMI core driver, the HDMI display driver, and the HDMI audio driver.

Table 77. HDMI source

File	Description
`drivers/mfd/mxc-hdmi-core.c`	HDMI core driver implementation.
`include/linux/mfd/mxc-hdmi-core.h`	HDMI core driver header file.
`drivers/video/fbdev/mxc/mxc_hdmi.c`	HDMI display driver implementation.
`sound/soc/fsl/fsl_hdmi.c`	HDMI Audio SoC DAI driver implementation.
`sound/soc/fsl/imx-hdmi-dma.c`	HDMI Audio SoC platform DMA driver implementation.
`drivers/mxc/hdmi-cec/hdmi-cec.c`	HDMI CEC driver implementation. The HDMI CEC library files are provided in imx-lib repo on codeaurraforum.
`drivers/mxc/hdmi-cec/hdmi-cec.c`	HDMI CEC driver implementation. The HDMI CEC library files are provided in imx-lib on GitHub nxp-imx project.

6.4.6.8.5 Menu configuration options

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

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

- **CONFIG_FB_MXC_HDMI** provides support for the HDMI video driver and can be selected with **Device Drivers** -> **Graphics support** -> **Support for frame buffer devices** -> **MXC HDMI driver support**.
- **CONFIG_SND_SOC_IMX_HDMI** provides support for HDMI audio through the ALSA/SoC subsystem, and can be selected with **Device Drivers** -> **Sound card support** -> **Advanced Linux Sound Architecture** -> **ALSA for SoC audio support** -> **SoC Audio support for IMX - HDMI**.
  Selecting either of the previous two configuration options will cause the MXC HDMI Core configuration option, **CONFIG_MFD_MXC_HDMI**, to be selected. This option can be selected wtih **Device Drivers** -> **Multifunction device drivers** -> **MXC HDMI Core**.

- **CONFIG_MXC_HDMI_CEC** option provides support for the HDMI CEC driver, and can be selected with **Device Drivers** -> **MXC support drivers** -> **MXC HDMI CEC (Consumer Electronic Control) support**.

6.4.6.9 External HDMI

6.4.6.9.1 Introduction

The High Definition Multimedia Interface (HDMI) driver supports the external SiI9022 HDMI hardware module providing capability to transfer uncompressed video, audio, and data using a single cable.

The HDMI driver is divided into two sub-components: a video display device driver that integrates with the Linux Frame Buffer API and an S/PDIF audio driver that transfers S/PDIF audio data to the SiI9022 HDMI hardware module.

The HDMI driver is only for demo application and supports the following features:

- HDMI video output supports 1080p60 and 720p60 resolutions.
- Support for reading EDID information from an HDMI sink device for video.
- Hotplug detection
- HDMI audio playback (2 channels, 16/24 bit, 44.1 kHz sample rate)

External HDMI is supported on i.MX 6 7ULP SoC.

Output data is transmitted via 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). 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 devices.

6.4.6.9.2 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 audio output depends on video display.

6.4.6.9.3 Source code structure

The source code for the HDMI driver is divided into the HDMI display driver and HDMI audio driver.

The HDMI display driver source is available in `drivers/video/fbdev/mxc`. The HDMI Audio driver source is in `sound/soc/fsl`.

Table 78. HDMI source

File	Description
`drivers/video/fbdev/mxc/mxsfb_sii902x.c`	HDMI display driver implementation.
`sound/soc/fsl/imx-spdif.c`	S/PDIF Audio SoC Machine driver implementation.
`sound/soc/fsl/fsl_spdif.c`	S/PDIF Audio SoC DAI driver implementation.

6.4.6.9.4 Menu configuration options

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

The following configuration options are required to enable HDMI support.

The **CONFIG_FB_MXS_SII902X** option provides support for the Sii902x HDMI video driver and can be selected with **Device Drivers** -> **Support for frame buffer devices** -> **Si Image SII9022 DVI/HDMI Interface Chip**.

HDMI video on i.MX 6Sololite depends on MXC ELCDIF Framebuffer.

The **CONFIG_SND_SOC_IMX_SII902X** option provides support for the HDMI Audio driver and can be selected with **Device Drivers** -> **Sound card support** -> **ALSA for SoC audio support** -> **Common SoC Audio options for Freescale CPUs:** -> **SoC Audio support for i.MX boards with sii902x**

6.5 Video for Linux Two (V4L2)

6.5.1 Introduction

The Video for Linux Two (V4L2) driver is a plug-in for the V4L2 framework that enables support for camera capture and display.

Some i.MX SoC support V4L2 based on the associated images processing units and capture hardware.

For more information on V4L2, see the API specification for Linux Video for Linux Two available at Linux Media Subsystem Documentation.

The V4L2 APIs enable camera and display controls but i.MX 8 only supports V4L2 capture and not display, using the DPU instead for display control. i.MX 6 and i.MX 7 use both capture and display V4L2.

6.5.1.1 i.MX 8 DPU V4L2

The Video for Linux Two (V4L2) driver on i.MX 8 is plug-in for the V4L2 framework that enables support for camera capture only with the Display Processing Unit (DPU).

The V4L2 DPU camera driver supports only basic capture. The V4l2 capture device takes incoming video images, either from a camera or a TV decoder, and captures the images to memory. The features supported by the V4L2 driver are as follows:

- RGB 24-bit and YUV 4:2:2 interleaved formats for capture interface
- Plug-in of different sensor drivers
- Streaming (queued) input buffer
- Programmable input and output pixel format and size
- RGB 16, 24, and 32-bit, YUV 4:2:0, and 4:2:2 interleaved input formats

The command `modprobe mxc_v4l2_capture` must be run before using V4L2 camera functions.

6.5.1.2 PxP V4L2

The Video for Linux Two (V4L2) drivers for PxP are used for display output only.

6.5.1.3 i.MX 6 with IPU V4L2

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 features supported by the IPU V4L2 driver are the 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 dependant)
- Programmable pixel format, size, frame rate for preview and capture
- Programmable rotation and flipping using a custom API
- RGB 16-bit, 24-bit, and 32-bit preview formats
- Raw Bayer (still only sensor dependent), RGB 16, 24, and 32-bit, YUV 4:2:0 and 4:2:2 planar, YUV 4:2:2 interleaved, and JPEG formats for capture
- Control of sensor properties including exposure, white-balance, brightness, contrast, and so on
- Plug-in of different sensor drivers
- 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 command `modprobe mxc_v4l2_capture` must be run before V4L2 functions.

6.5.1.4 IPU 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: `drivers/media/platform/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, and the AE and AWB turn on. The frame rate may change depending on light sensor samples.

Drivers for specific cameras can be found in `drivers/media/platform/mxc/capture/`.

6.5.2 V4L2 capture device

The V4L2 capture device includes two interfaces:

- Capture interface: uses i.MX processing engine to record the YCrCb video stream
- Overlay interface: uses i.MX processing engine to display the preview video to the SDC foreground and background panel.

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 low-level layer is the i.MX SoC implementation for the display engine associated with the SoC detailed in each V4L2 SoC chapter.

6.5.2.1 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

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

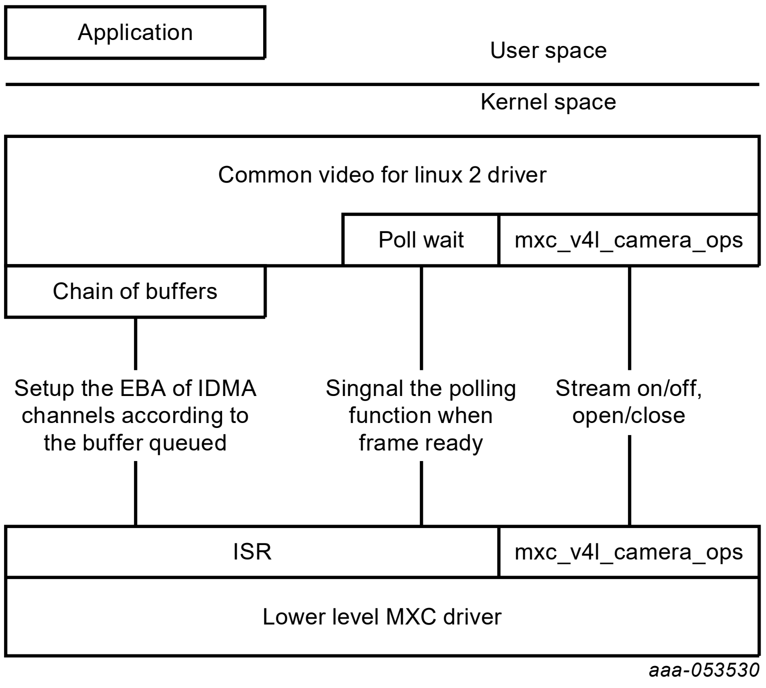
Application
User space
Kernel space
Common video for linux 2 driver
Poll wait
mxc_v4l_camera_ops
Chain of buffers
Setup the EBA of IDMA channels according to the buffer queued
Singnal the polling function when frame ready
Stream on/off, open/close
ISR
mxc_v4l_camera_ops
Lower level MXC driver
aaa-053530

Figure 27. Video4Linux2 Capture API interaction

6.5.2.2 Use of the V4L2 Capture APIs

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

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

6.5.3 V4L2 output device

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 `drivers/media/platform/mxc/output/mxc_vout.c`.

6.5.3.1 V4L2 output IOCTLs

Currently, the memory map stream API is supported. The 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 use, the ID is **V4L2_CID_MXC_MOTION**. The supported values include the following:

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

6.5.3.2 Use of 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 using IOCTL `VIDIOC_S_FMT`.
2. Sets the control information using IOCTL `VIDIOC_S_CTRL`, for rotation, de-interlace motion (if needed).
3. Sets the output information using IOCTL `VIDIOC_S_CROP`.
4. Requests a buffer 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 the IOCTL `VIDIOC_QUERYBUF` to query buffers.
7. Passes the data that requires post-processing to the buffer.
8. Queues the buffer 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 executing IOCTL `VIDIOC_STREAMOFF`.

6.5.4 Software operatoins

6.5.4.1 Source code structure

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

These files are available in `drivers/media/platform/mxc`.

Table 79. V4L2 Driver Files

File	Description
`drivers/media/platform/mxc/output/mxc_vout.c`	i.MX 6 and i.MX 7 V4L2 output device driver
`drivers/media/platform/mxc/output/mxc_pxp_v4l2.c`	V4L2 PxP output device driver
`drivers/media/platform/mxc/output/mxc_pxp_v4l2.h`	V4L2 PxP output device driver header
`drivers/media/platform/mxc/capture/mxc_v4l2_capture.c`	V4L2 capture device driver
`drivers/media/platform/mxc/capture/mxc_v4l2_capture.h`	Header file for V4L2 capture device driver
`drivers/media/platform/mxc/capture/ipu_bg_overlay_sdc.c`	IPU synchronous background driver
`drivers/media/platform/mxc/capture/ipu_fg_overlay_sdc.c`	IPU synchronous forground driver
`drivers/media/platform/mxc/capture/ipu_prp_sw.h`	IPU Pre-processing header
`drivers/media/platform/mxc/capture/ipu_still.c`	IPU Pre-processing still image capture driver
`drivers/media/platform/mxc/capture/ipu_prp_vf_sdc_bg.c`	IPU Pre-processing view finder (synchronous background)
`drivers/media/platform/mxc/capture/ipu_prp_enc.c`	IPU Pre-processing Encoder driver
`drivers/media/platform/mxc/capture/ipu_csi_enc.c`	IPU CSI interface driver

Drivers for V4L2 cameras can be found in `divers/media/platform/mxc/capture`.

Drivers for V4L2 output can be found in `drivers/media/platform/mxc/output`.

6.5.4.2 Menu configuration options

The kernel configuration options are provided below:

Device Drivers -> **V4L platform devices** -> **MXC Video For Linux Video Capture**

Device Drivers -> **V4L platform devices** -> **MXC Video For Linux Video Output**

6.6 Video Analog-to-Digital Converter (VADC)

6.6.1 Introduction

The video analog-to-digital converter (VADC) consists of an analog video front end (AFE), and a digital video decoder. The AFE accepts NTSC or PAL input from a device, such as an analog camera.

The two parts are configured in the VADC driver. The video decoder outputs the YUV444-formatted data.

The Video ADC has the following features:

- Internal voltage and current reference generator
- 10-bit resolution (9.5 bit ENOB at 66.5 Msps)

- 4 analog inputs, with all inputs available for CVBS
- Programmable anti-aliasing filter, gain, and clamp

The video decoder has the following features:

- NTSC/PAL decoder
- Direct data path (no complex resampling)
- Automatic standards detection
- 2D adaptive comb filter
- Datapath/clocking architecture encompasses a time base corrector for VCR signals
- Luma passband is flat to > 6 MHz

6.6.2 Software operation

The VADC 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 image capture.

The V4L2 capture supports the following operations:

- Capture stream mode
- YUV444
- PAL
- NTSC

6.6.3 Source code structure

The table below shows the VADC driver source files available in `drivers/media/platform/mxc/capture`.

Table 80. VADC Driver Files

File	Description
`drivers/media/platform/mxc/capture/mxc_vadc.c`	VADC driver source code
`drivers/media/platform/mxc/capture/mxc_vadc.h`	VADC driver Header

6.6.4 Menu configuration options

In menu configuration, enable the following module:

Device Drivers -> **Multimedia devices** -> **Video capture adapters** -> **MXC Video For Linux Camera** -> **MXC VADC support**

6.6.5 DTS configuration

VADC analog inputs can choose [0-3]. CSI1 or CSI2 can be used to capture the VADC data. They can be configured in the DTS file.

For example:

```
vadc_in = <0>;  /* VADC input select */
csi_id = <1>; /* CSI select */
```

The VADC input selected to `vin1` and CSI2 is used to capture the VADC data.

6.7 Video Processing Unit (VPU)

6.7.1 Introduction

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.

Different VPUs are supported on i.MX 6, i.MX 8, and i.MX 9 SoC. The following table lists the different VPUs.

Table 81. VPU

SoC	VPU Provider	Library
i.MX 6	Chips and Media	`imx-vpu.so`
i.MX 8M Quad, 8M Mini, and 8M Plus	Hantro	`imx-hantro.so`
i.MX 8QuadMax, i.MX 8QuadXPlus	Amphion	No library
i.MX 95	Chips and Media	No library

Note:

Malone is decoder while Windsor is encoder. Both come from Amphion.

Hantro stands for the following providers:

- *`hantro`/(8MQuad/8M Plus decoder)*
- *`hantro_845`/(8M Mini decoder)*
- *`hantro_845_h1`/(8M Mini encoder)*
- *`hantro_vc8000e`/(8M Plus encoder)*

6.7.2 Software operation

The VPU software can be divided into two parts: the kernel driver and the user-space library as well as the application in the 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 IOCTL to provide interface for memory allocating and releasing
  - Memory map for register and memory accessing in user space

The VPU user space driver has the following functions:

- Codec library
- 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

User space application for simple verification:

- Read video raw data
- YUV file dump
- General options to configure the codec behavior

The following figure shows a simple workflow shown in the H.264 example.

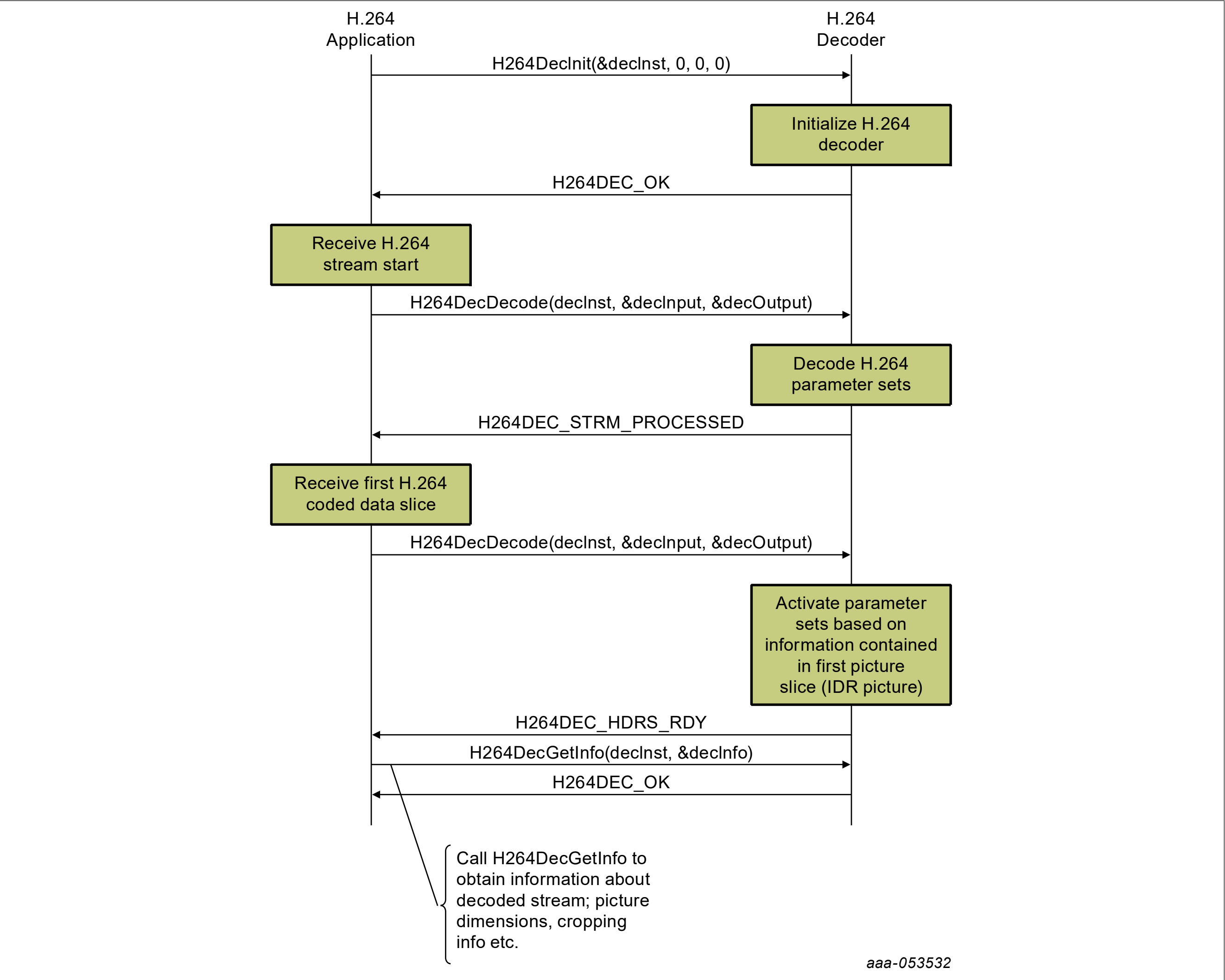
H.264 Application
H.264 Decoder
H264DecInit(&decInst, 0, 0, 0)
Initialize H.264 decoder
H264DEC_OK
Receive H.264 stream start
H264DecDecode(decInst, &decInput, &decOutput)
Decode H.264 parameter sets
H264DEC_STRM_PROCESSED
Receive first H.264 coded data slice
H264DecDecode(decInst, &decInput, &decOutput)
Activate parameter sets based on information contained in first picture slice (IDR picture)
H264DEC_HDRS_RDY
H264DecGetInfo(decInst, &decInfo)
H264DEC_OK
Call H264DecGetInfo to obtain information about decoded stream; picture dimensions, cropping info etc.
aaa-053532

Figure 28. Simple workflow shown in the H.264 example (1)

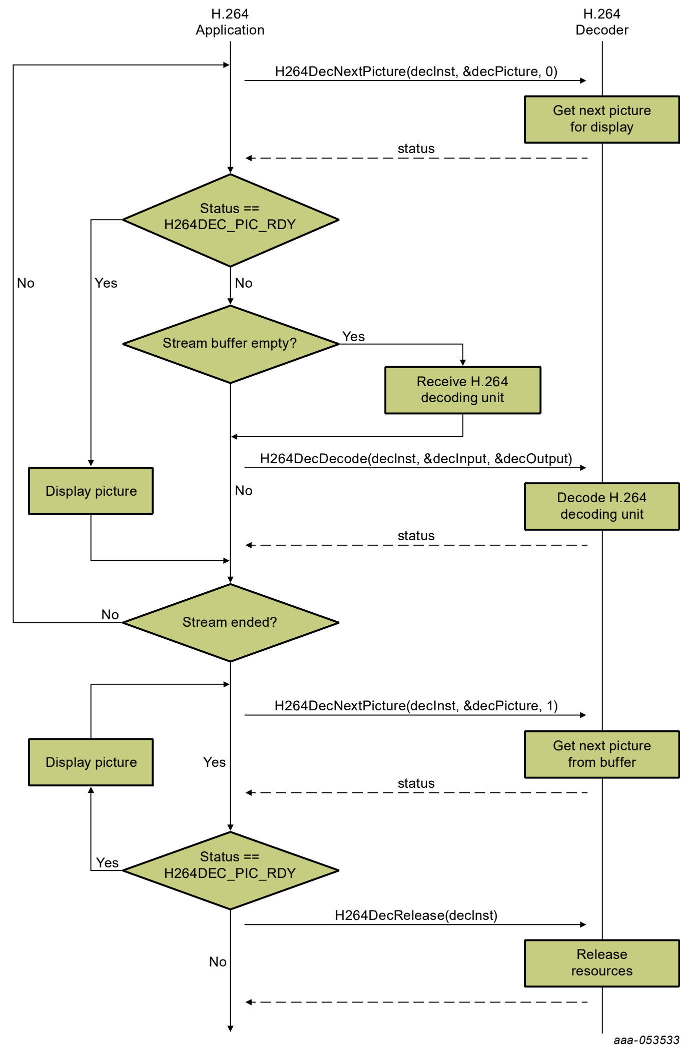
H.264
Application
H.264
Decoder
H264DecNextPicture(decInst, &decPicture, 0)
Get next picture for display
status
Status == H264DEC_PIC_RDY
No
Yes
No
Stream buffer empty?
Yes
Receive H.264 decoding unit
H264DecDecode(decInst, &decInput, &decOutput)
Display picture
No
Decode H.264 decoding unit
status
No
Stream ended?
H264DecNextPicture(decInst, &decPicture, 1)
Display picture
Yes
Get next picture from buffer
status
Yes
Status == H264DEC_PIC_RDY
H264DecRelease(decInst)
No
Release resources
aaa-053533

Figure 29. Simple workflow shown in the H.264 example (2)

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.

There is one unified interface to wrap all different video formats. The following are the related APIs:

```
CODEC_STATE decoder_decode_xxx(CODEC_PROTOTYPE * arg,STREAM_BUFFER * buf,
 OMX_U32 * consumed,FRAME * frame);
CODEC_STATE decoder_getinfo_xxx(CODEC_PROTOTYPE * arg,STREAM_INFO * pkg);
CODEC_STATE decoder_setppargs_xxx(CODEC_PROTOTYPE * codec,PP_ARGS * args);
CODEC_STATE decoder_setframebuffer_xxx(CODEC_PROTOTYPE * arg, BUFFER
 *buff,OMX_U32 available_buffers);
CODEC_STATE decoder_pictureconsumed_xxx(CODEC_PROTOTYPE * arg, BUFFER *buff);
CODEC_STATE decoder_getframe_mpeg4(CODEC_PROTOTYPE * arg, FRAME * frame,OMX_BOOL
 eos);
FRAME_BUFFER_INFO decoder_getframebufferinfo_xxx(CODEC_PROTOTYPE * arg);
CODEC_STATE decoder_endofstream_xxx(CODEC_PROTOTYPE * arg)
OMX_S32 decoder_scanframe_xxx(CODEC_PROTOTYPE * arg, STREAM_BUFFER * buf,OMX_U32
 * first, OMX_U32 * last);
CODEC_STATE decoder_abort_xxx(CODEC_PROTOTYPE * arg);
CODEC_STATE decoder_abortafter_xxx(CODEC_PROTOTYPE * arg);
CODEC_STATE decoder_setnoreorder_xxx(CODEC_PROTOTYPE * arg, OMX_BOOL
 no_reorder);
static void decoder_destroy_xxx(CODEC_PROTOTYPE * arg)
```

6.7.3 Source code structure

The table below lists the kernel space source files available in `drivers/mxc/vpu`.

Table 82. VPU driver files

File	Description
`drivers/mxc/vpu/mxc_vpu.c`	Chips and Media VPU Driver
`include/linux/mxc_vpu.h`	Chips and Media VPU Header
`drivers/media/platform/amphion/*`	All source code for Amphion malone decoder and windsor encoder
`drivers/mxc/hantro/hantrodec.c`	Hantro 8M Quad VPU decoder driver
`include/linux/hantrodec.h`	Hantro decoder kernel header
`include/uapi/linux/hantrodec.h`	Hantro decoder user space header
`drivers/mxc/hantro_845/hantrodec_845s.c`	Hantro 8M Mini VPU deocder driver
`drivers/mxc/hantro_845_h1/hx280enc.c`	Hantro 8M Mini HL encodeer driver
`drivers/mxc/hantro_845_h1/hx280enc.h`	Hantro 8M Mini HL encodeer header
`drivers/mxc/hantro_vc8000e/hx280enc_vc8000e.c`	Hantro 8M Plus VC8000E encoder driver
`drivers/mxc/hantro_v4l2/*`	Hantro V4L2 wrapper driver for decoder and encoder
`drivers/mxc/vpu/wave6/*`	WAVE6 i.MX 95 V4L2 driver for decoder and encoder (temporal directory before upstreaming)

The table below lists the user-space library source files available in the i.MX 6 `imx-vpu-(version)/vpu` directory.

Table 83. i.MX 6 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
`vpu_util.h`	Header file

The table below lists the firmware files available in the `firmware-imx-(version)/lib/firmware/vpu/` directory.

Table 84. VPU firmware files

File	Description
`vpu_fw_imx6xxx.bin`	i.MX 6 VPU firmware
`vpu_fw_imx8xxx.bin`	i.MX 8 VPU firmware
`wave633c_codec_fw.bin`	i.MX 95 VPU firmware

6.7.4 Menu configuration options

In menu configuration, enable the following module for the VPU driver:

For i.MX 6 with VPU, select **Device Drivers** -> **MXC support drivers** -> **Support for MXC VPU (Video Processing Unit)**.

For i.MX 8M, select **Device Drivers** -> **MXC support drivers** -> **MXC HANTRO (Video Processing Unit) support**.

For i.MX 8QuadMax and i.MX 8QuadXPlus, select **Device Drivers** -> **Multimedia support** -> **Media drivers** -> **V4L platform devices** -> **Amphion VPU (Video Processing Unit) Codec IP**.

For i.MX 95 WAVE6, select **Device Drivers** -> **MXC support drivers** -> **WAVE6 VPU**.

6.8 JPEG Encoder and Decoder

6.8.1 Introduction

The JPEG Encoder/Decoder is present on i.MX 8QuadMax/i.MX QuadXPlus and i.MX 95 SoCs. For i.MX 8, the JPEG is a part of the ISI domain, while for i.MX 9, it is a part of the VPU domain.

The JPEG Encoder consists of a JPEG-E-X core and a JPEG Encoder Wrapper (JPGENCWRP). Similarly, the JPEG Decoder consists of a JPEG decoder core (JPEG-D-X) and its corresponding wrapper.

The JPEG cores are compliant with the industry standards Baseline and Extended ISO/IEC 10918-1 JPEG, with some limitations documented in the *i.MX 8DualXPlus Applications Processor Reference Manual* (IMX8DQXPRM).

The JPEG encoder wrapper (JPGENCWRP) is used to work with the Cast JPEG Encoder Core. It has a configuration mode and an encoding mode.

- In configuration mode, it can fetch the configuration bitstream from the system memory and feed it to the encoder.
- In encoding mode, it can fetch the image pixel data through the AXI bus interface and feed to the Encoder Core for encoding.

Similarly, the JPEG Decoder Wrapper provides the interface for the Cast JPEG Decoder core.

The JPEG wrappers support multiple image encoding through context switching by the encoding descriptors. There are four bitstream slots. Each one can be enabled independently by chained descriptors.

The JPEG encoder and decoder support a maximum horizontal resolution of 8K (0x2000) pixels. The horizontal resolution needs to be an integer times of 8. It is the same for the vertical resolution. For YUV422 and YUV420, the resolution must be multiple of 16. The image size may be up to 64K x 64K.

The JPEG encoder and decoder support 8-bit and 12-bit precision.

6.8.2 Overview of the JPEG Encoder/Decoder Driver

The driver relies on the V4L2 framework.

The JPEG Encoder/Decoder Driver implements a subset of the IOCTLs exposed by the V4L2 framework, namely the following `v4l2_ioctl_ops`:

- VIDIOC_QUERYCAP
- VIDIOC_ENUM_FMT_VID_CAP
- VIDIOC_ENUM_FMT_VID_OUT
- VIDIOC_TRY_FMT_VID_CAP
- VIDIOC_TRY_FMT_VID_OUT
- VIDIOC_S_FMT_VID_CAP
- VIDIOC_S_FMT_VID_OUT
- VIDIOC_G_FMT_VID_CAP
- VIDIOC_G_FMT_VID_OUT
- VIDIOC_QBUF
- VIDIOC_DQBUF
- VIDIOC_CREATE_BUFS
- VIDIOC_PREPARE_BUF
- VIDIOC_REQBUFS
- VIDIOC_QUERYBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF

User applications may interact with the driver through the supported V4L2 IOCTLs.

The JPEG driver supports streaming I/O through memory mapping. This capability is exposed through the **V4L2_CAP_STREAMING** flag, when the **VIDIOC_QUERYCAP** is used. Streaming is an I/O method where only pointers to buffers are exchanged between the application and driver, but the data is not copied. Memory mapping is primarily intended to map buffers in the device memory into the application's address space.

The JPEG driver supports buffers memory-mapping through the multi-planar API.

For more information on streaming I/O, see Streaming I/O (Memory Mapping).

6.8.3 Limitations of the JPEG Encoder/Decoder Driver

The hardware, namely the JPEG wrappers, support multiple-image encoding through context switching. The driver does not use context switching, and only one of the available four slots is used. The hardware supports bitstream buffer half/full and returns features for bitstream buffer management, but the driver does not use them.

The hardware supports the following formats: YUV444, YUV420, YUV422, RGB, ARGB, and Gray.

The driver supports the same formats as the hardware.

The driver supports JPEG images encoding and decoding through gstreamer, but it does not yet support MJPEG videos.

The hardware has the limitation that the decoded image resolution should be larger than 64 x 64.

The hardware has the limitation that the decoded image should have at least a default Huffman table (DHT marker section should be present in the JPEG input stream).

If the decoded JPEG does not have a DHT, the driver provides a default one.

7 Audio

7.1 Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound

7.1.1 ALSA Sound Driver introduction

The Advanced Linux Sound Architecture (ALSA), now the most popular architecture in the 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. Allow machines to add controls to the sound card, for example, volume control for a speaker AMP.

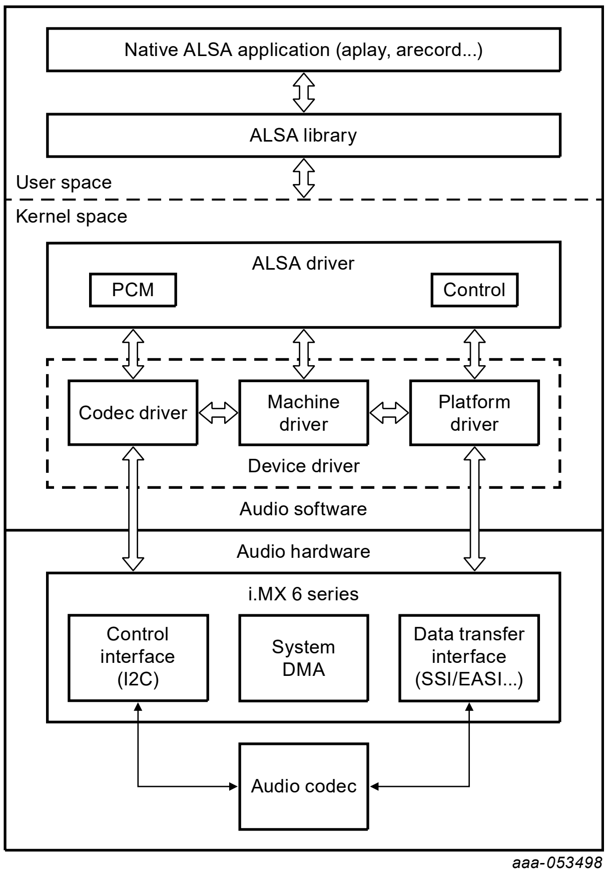

Figure 30. ALSA SoC software srchitecture

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, I2S, 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 OS source tree at `linux/Documentation/sound/alsa/soc` and at www.alsa-project.org/main/index.php/ASoC.

7.1.2 SoC sound card

Currently, the stereo codec (WM8958, WM8960, WM8962, and WM8524), 7.1 codec (CS42888), and AM/FM codec (SI4763) drivers are implemented using the ASoC 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, the 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 and i.MX 6Solo SABRE Auto platform.*
- *The AM/FM codec is only supported on the i.MX 6Quad and i.MX 6Solo SABRE Auto platform.*

7.1.2.1 Stereo codec features

The stereo Codec supports the following features:

- Sample rates for playback and capture are 8 KHz, 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`

7.1.2.2 7.1 audio codec features

- Sample rates for playback and record:
  - 48 KHz, 96 KHz, 192 KHz
  - Playback: 5.512 k, 8 k, 11.025 k, 16 k, 22 k, 32 k, 44.1 k, 48 k, 64 k, 88.2 k, 96 k, 176.4 k, 192 k (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`

7.1.2.3 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`

7.1.2.4 Sound card information

The registered sound card information can be listed as follows using the commands `aplay -l` and `arecord -l`. For example, the stereo sound card is registered as card 0.

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

7.1.3 Hardware operation

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

7.1.3.1 Stereo audio codec

The stereo audio codec is controlled by the I2C 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 WM8958, WM8960, and 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 I2C
- Mixers and audio controls
- Codec audio operations
- DAC digital mute control

The WM8958, WM8960, and WM8962 codec are registered as an I2C client when the module is initialized. 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.

7.1.3.2 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 or 6 channel playback through ASRC. On the i.MX 6 SABRE ARD 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 2 audio output port 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 is initialized. The APIs are exported to the upper layer by the structure `snd_soc_dai_ops`.

7.1.3.3 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.

7.1.4 Software operation

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

7.1.4.1 ASoC driver source architecture

The file `imx-pcm-dma.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. `fsl_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 Hi-Fi DAI drivers. The direct hardware operations on the stereo codec are in `wm8994.c`, `wm8960.c`, and `wm8962.c`.. `imx-wm8958.c`, `imx-wm8960.c`, and `imx-wm8962.c` are the machine layer codes, which create the driver device and register the stereo sound card.

The multichannel codec is connected to the CPU through the ESAI interface. `fsl_esai.c` registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip ESAI interface. `cs42888.c` registers the multichannel codec and Hi-Fi DAI drivers. The direct hardware operations on the multichannel 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. `fsl_ssi.c` registers the CPU DAI driver for the stereo ALSA SoC and configures the on-chip SSI interface. `si476x.c` registers the Tuner codec and Tuner DAI drivers. The direct hardware operations on the codec are in `si476x.c`. `imx-si476x.c` is the machine layer code, which creates the driver device and registers the sound card.

7.1.4.2 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.

7.1.4.3 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 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 the codec hardware.
- Triggers the transfer.

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

7.1.4.4 Device tree binding

See the following documents:

- `Documentation/devicetree/bindings/sound/fsl,ssi.txt`
- `Documentation/devicetree/bindings/sound/fsl-sai.txt`
- `Documentation/devicetree/bindings/sound/fsl,esai.txt`
- `Documentation/devicetree/bindings/sound/fsl,asrc.txt`
- `Documentation/devicetree/bindings/sound/wm8962.txt`
- `Documentation/devicetree/bindings/sound/wm8960.txt`
- `Documentation/devicetree/bindings/sound/wm8994.txt`
- `Documentation/devicetree/bindings/sound/cs42xx8.txt`
- `Documentation/devicetree/bindings/sound/imx-audmux.txt`
- `Documentation/devicetree/bindings/sound/imx-audio-wm8962.txt`
- `Documentation/devicetree/bindings/sound/imx-audio-cs42888.txt`
- `Documentation/devicetree/bindings/sound/imx-audio-si476x.txt`

7.1.4.5 Source code structure

The following table lists the stereo codec SoC driver source files in `sound/soc/fsl`.

Table 85. Stereo Codec SoC driver files

File	Description

`sound/soc/fsl/imx-wm8958.c`
`sound/soc/fsl/imx-wm8960.c`
`sound/soc/fsl/imx-wm8962.c`
	Machine layer for stereo codec ALSA SoC (codec as I2S Master)

`sound/soc/fsl/imx-pcm-dma.c`	Platform layer for stereo codec ALSA SoC
`sound/soc/fsl/imx-pcm.h`	Header file for PCM driver and AUDMUX register definitions
`sound/soc/fsl/fsl_ssi.c`	SSI CPU DAI driver for stereo codec ALSA SoC
`sound/soc/fsl/fsl_ssi.h`	Header file for SSI CPU DAI driver and SSI register definitions
`sound/soc/fsl/fsl_sai.c`	SAI CPU DAI driver for stereo codec ALSA SoC

Table 85. Stereo Codec SoC driver files*...continued*

File	Description
`sound/soc/fsl/fsll_sai.h`	Header file for SAI CPU DAI driver and SAI register definitions

`codecs/wm8994.c`
`codecs/wm8960.c`
`codecs/wm8962.c`
	codec layer for stereo codec ALSA SoC

`codecs/wm8994.h`
`codecs/wm8960.h`
`codecs/wm8962.h`
	Header file for stereo codec driver

The table below lists the AM/FM codec SoC driver source files. These files are under `sound/soc`.

Table 86. AM/FM Codec SoC driver source files

File	Description
`sound/soc/fsl/imx-si476x.c`	Machine layer for stereo codec ALSA SoC (codec as I2S Slave)
`sound/soc/fsl/imx-pcm-dma.c`	Platform layer for stereo codec ALSA SoC
`sound/soc/fsl/imx-pcm.h`	Header file for pcm driver and AUDMUX register definitions
`sound/soc/fsl/fsl_ssi.c`	SSI CPU DAI driver for stereo codec ALSA SoC
`sound/soc/fsl/fsl_ssi.h`	Header file for SSI CPU DAI driver and SSI register definitions
`sound/soc/codecs/si476x.c`	Codec layer for stereo codec ALSA SoC

The table below lists the multiple-channel ADC SoC driver source files.

Table 87. CS42888 ASoC driver source files

File	Description
`sound/soc/fsl/imx-cs42888.c`	Machine layer for multiple-channel codec ALSA SoC
`sound/soc/fsl/imx-pcm-dma.c`	Platform layer for multiple-channel codec ALSA SoC
`sound/soc/fsl/imx-pcm.h`	Header file for pcm driver
`sound/soc/fsl/fsl_esai.c`	ESAI CPU DAI driver for multiple-channel codec ALSA SoC
`sound/soc/fsl/fsl_esai.h`	Header file for ESAI CPU DAI driver
`sound/soc/codecs/cs42xx8.c`	codec layer for multiple-channel codec ALSA SoC
`sound/soc/codecs/cs42xx8.h`	Header file for multiple-channel codec driver
`sound/soc/fsl/fsl_asrc.c`	CPU DAI driver of ASRC P2P
`sound/soc/fsl/fsl_asrc.h`	Header file for CPU DAI driver of ASRC P2P
`sound/soc/fsl/fsl_asrc_pcm.c`	Platform layer for ASRC P2P

7.1.4.6 Menu configuration options

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

- SoC Audio supports for WM8958, WM8960, and WM8962 CODEC. In `menuconfig`, this option is available:

```
-> Device Drivers
   -> Sound card support
      -> Advanced Linux Sound Architecture
         -> ALSA for SoC audio support
            -> SoC Audio for Freescale CPUs
```

```
                -> SoC Audio support for i.MX boards with wm8962 (or wm8958,
  wm8960)
```

- SoC Audio supports for i.MX CS42888. In `menuconfig`, this option is available:

```
-> Device Drivers
   -> Sound card support
      -> Advanced Linux Sound Architecture
         -> ALSA for SoC audio support
            -> SoC Audio for Freescale CPUs
               -> SoC Audio support for i.MX boards with cs42888
```

- SoC Audio supports for AM/FM. In `menuconfig`, this option is available:

```
-> Device Drivers
   -> Sound card support
      -> Advanced Linux Sound Architecture
         -> ALSA for SoC audio support
            -> SoC Audio for Freescale CPUs
               -> SoC Audio support for i.MX boards with si476x
```

7.2 Asynchronous Sample Rate Converter (ASRC) on i.MX 6/i.MX 8QuadMax/i.MX 8QuadXPlust

7.2.1 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.

7.2.2 Hardware operation

ASRC includes the following features:

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

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

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

For more information, see the ASRC chapter in the Applications Processor documentation associated with the SoC.

7.2.3 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 two scenarios.

- **Memory** -> **ASRC** -> **Memory**: ASRC is controlled by the user application or ALSA plug-in.
- **Memory** -> **ASRC** -> **peripheral**: ASRC is controlled directly by other ALSA drivers.

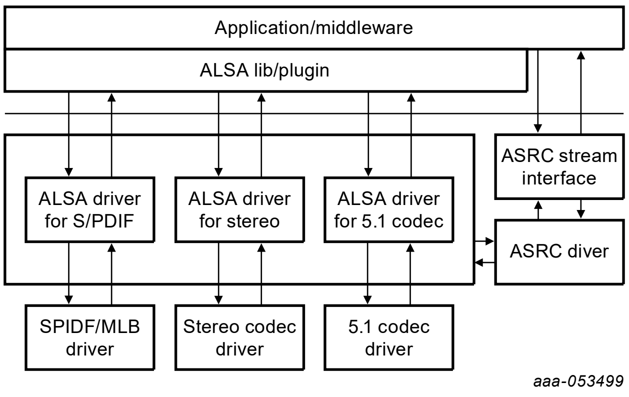

Figure 31. Audio driver interactions

As illustrated in the 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 number of the total channels should be ten, or else the adjusted value cannot be saved properly.

7.2.3.1 Sequence for Memory to ASRC to Memory

1. Open `/dev/mxc_asrc` device.
2. Request ASRC pair: `ASRC_REQ_PAIR`.
3. Configure ASRC pair: `ASRC_CONIFG_PAIR`.
4. Start ASRC: `ASRC_START_CONV`.
5. Write the raw audio data (to be converted) into the user maintained input buffer. Fill the `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`.
6. Stop the ASRC conversion: `ASRC_STOP_CONV`.
7. Release ASRC pair: `ASRC_RELEASE_PAIR`.
8. Close the `/dev/mxc_asrc` device.

7.2.3.2 Sequence for Memory to ASRC to Peripheral

Memory to ASRC to peripheral audio path is involved in the 7.1 audio codec driver. In the 7.1 audio sound card, a new device with the name `cs42888audio [cs42888-audio], device 1: HiFi-ASRC-FE (*)` is specified for playback and capture with ASRC. The steps below show the flow of calling ASRC to memory to peripheral:

1. The sound device (PCM) has been registered and starts to enable the DMA channel in the ALSA driver.
2. Request ASRC pair: `fsl_asrc_request_pair`.
3. Configure ASRC pair: `fsl_asrc_config_pair`.
4. Enable the DMA channel from Memory to ASRC and from ASRC to Memory.
5. Start DMA channel and start ASRC conversion: `fsl_asrc_start_pair`.
6. When audio data playback is complete, stop the DMA channel and ASRC: `fsl_asrc_stop_pair`.
7. Release ASRC pair: `fsl_asrc_release_pair`.

7.2.3.3 Source code structure

The table below lists the source files available in `sound/soc/fsl`.

Table 88. ASRC source files

File	Description
`sound/soc/fsl/fsl_asrc_m2m.c`	ASRC M2M driver implementation codes
`sound/soc/fsl/imx-cs42888.c`	Memory to ASRC to ESAI TX implementation in 7.1 audio codec machine driver
`sound/soc/fsl/imx-pcm-dma.c`	Memory to ASRC to ESAI TX implementation in 7.1 audio codec platform driver
`sound/soc/fsl/fsl_esai.c`	Memory to ASRC to ESAI TX implementation in 7.1 audio codec CPU driver
`sound/soc/fsl/cs42xx8`	Memory to ASRC to ESAI TX implementation in 7.1 audio codec codec driver
`sound/soc/fsl/fsl_asrc.c`	ALSA CPU DAI driver of ASRC P2P
`sound/soc/fsl/fsl_asrc.h`	Header file for ALSA CPU DAI driver of ASRC P2P
`sound/soc/fsl/fsl_asrc_dma.c`	ALSA platform layer for ASRC P2P
`sound/soc/fsl/sound/soc/fsl/fsl_asrc_dma.c`	ALSA platform layer for ASRC M2M

7.2.3.4 Menu configuration options

The menu configuration options are as follows:

```
-> Device Drivers
  -> Sound card support
    -> Advanced Linux Sound Architecture
      -> ALSA for SoC audio support
        -> SoC Audio for Freescale i.MX CPUs
          -> Asynchronous Sample Rate Converter (ASRC) module support
```

Then, the ASRC driver can only be configured with the build-in module.

7.2.3.5 Device tree binding

The functions of device tree bindings for ASRC M2M are as follows:

- `compatible`: Compatible list, must contain `fsl,imx6q-asrc`.
- `reg`: Offset and length of the register set for the device.
- `interrupts`: Contains the ASRC interrupt.
- `clocks`: Contains an entry for each entry in clock-names.

- `clock-names`: Must contain `mem`, `ipg`, `asrck`, and `dma`. (Generally, `dma` is used for SPBA clock.)
- `dmas`: Generic dma devicetree binding as described in `Documentation/devicetree/bindings/dma/dma.txt`.
- `dma-names`: Six dmas have to be defined, `txa`, `rxa`, `txb`, `rxb`, `txc`, `rxc`.
- `fsl,clk-map-version`: The mapping relationships in different SoCs are different. This version number can be used to indicate clock map information.
- `fsl,clk-channel-bits`: Indicates the channel bit information.

The functions of device tree bindings for ASRC P2P are as follows:

- `compatible`: Compatible list, must contain `fsl,imx6q-asrc-p2p`.
- `fsl,p2p-rate`: A valid sample rate for Back-End (I2S) playback and record.
- `fsl,p2p-width`: A valid sample width for Back-End (I2S) playback and record.
- `fsl,asrc-dma-rx-events`: Contains three SDMA event numbers for ASRC RX.
- `fsl,asrc-dma-tx-events`: Contains three SDMA event numbers for ASRC TX.

7.2.3.6 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, the ASRC core clock is enabled.

ASRC_CONFIG_PAIR:

Configure the ASRC pair allocated. User is responsible for providing parameters defined in struct `asrc_config`. Items in `asrc_config` are listed below:

- `pair`: ASRC pair allocated by the IOCTL (`ASRC_REQ_PAIR`).
- `channel_num`: channel number.
- `buffer_num`: buffer number need for input and output buffer use. The input/output buffers are allocated inside the ASRC driver. The user is responsible for remap it into the user space.
- `dma_buffer_size`: buffer size for input and output buffers. The buffer size should be in the unit of page size. Usually, 4k byte is used.
- `input_sample_rate`: input sampling rate. Input sample rate should be in 5.512k, 8k, 11.025k, 16k, 22k, 32k, 44.1k, 48k, 64k, 88.2k, 96k, 176.4k, 192k.
- `output_sample_rate`: output sampling rate. Output sampling rate should be in 32k, 44.1k, 48k, 64k, 88.2k, 96k, 176.4k, 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, NONE. If using clock except NONE, 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 the 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 copies the `output_buffer_length` to `output_buffer_vaddr`. However, before calling `ASRC_CONVERT`, the user is responsible for filling 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 the user-filled `output_buffer_size` (the difference should be 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.

7.3 HDMI audio

7.3.1 Introduction

HDMI Audio is described in HDMI Audio.

7.4 The Sony/Philips Digital Interface (S/PDIF)

7.4.1 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 received clock from the incoming S/PDIF stream.

7.4.1.1 S/PDIF overview

The figure below shows the block diagram of the S/PDIF interface.

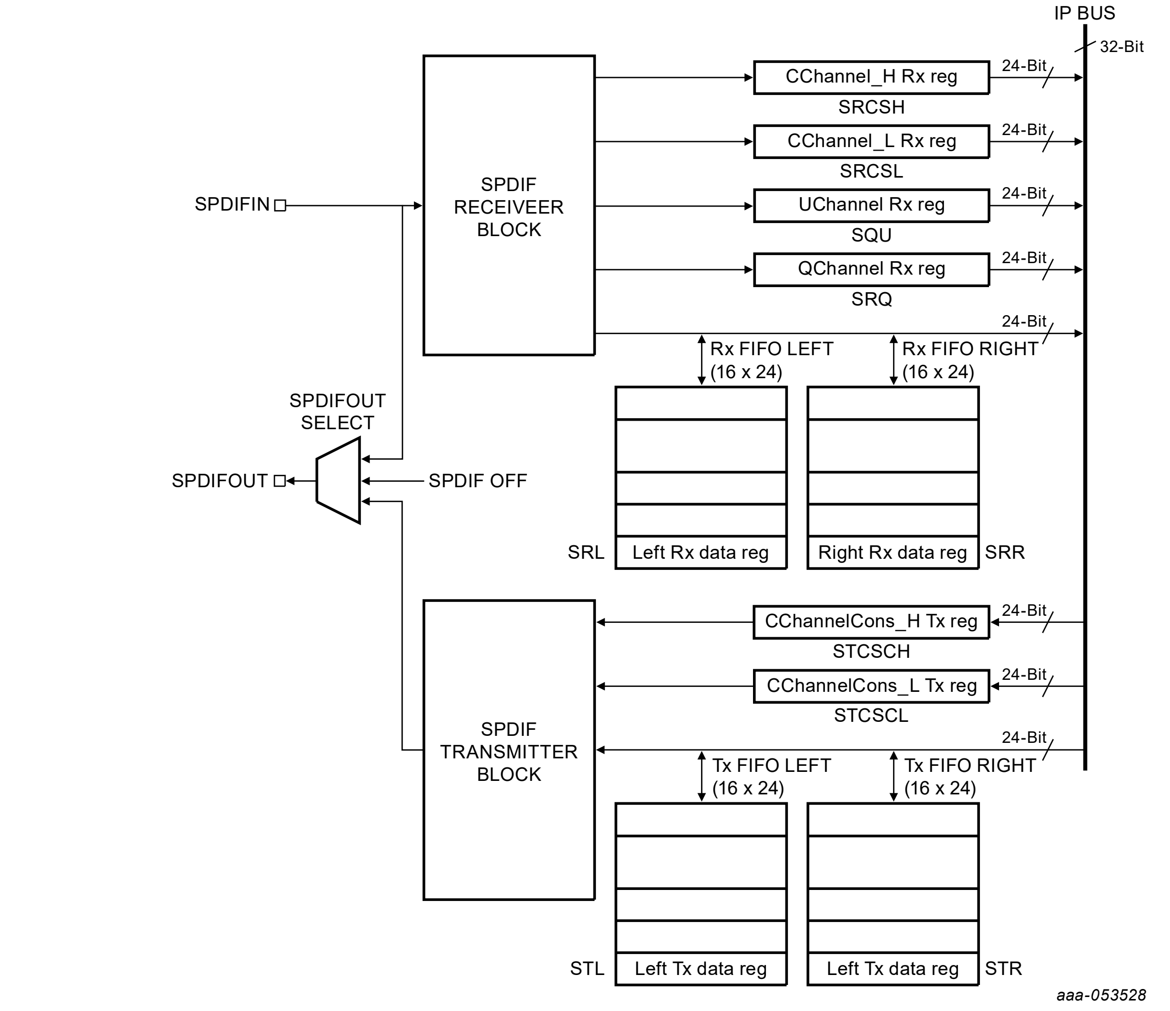

Figure 32. S/PDIF transceiver data interface block diagram

7.4.1.2 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 an 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 the S/PDIF block. The S/PDIF receiver can recover the S/PDIF RX clock from the S/PDIF stream. Figure 32 shows the clock structure of the S/PDIF transceiver.

7.4.1.3 Software overview

The S/PDIF driver is designed under 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 KHz, 48 KHz, or 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 KHz to 96 KHz. The driver provides the same interface for PCM and compressed data transmission.

7.4.1.4 ASoC layer

The ASoC layer divides audio drivers for embedded platforms into separate 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`. For the S/PDIF driver, we can reuse the platform layer (`imx-pcm-dma.c`) that is used by the SSI stereo codec driver and also the generic dummy codec driver useful for DAI links creation without a real codec.

7.4.2 S/PDIF TX Driver

The S/PDIF TX driver supports the following features.

- 32 KHz, 44.1 KHz, and 48 KHz sample rates.
- Signed 16-bit 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.
- Stereo playback.
- Information query through `iecset` or `amixer`.
- The device ID can be determined by using the `aplay -l` utility to list out the playback audio devices. For example:

```
root@ ~$ aplay -l
**** List of PLAYBACK Hardware Devices ****
card 0: imxspdif [imx-spdif], device 0: S/PDIF PCM snd-soc-dummy-dai-0 []
Subdevices: 1/1
Subdevice #0: subdevice #0
```

Note: *The number at the beginning of the `IMX_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.

```
#iecset -c 0
```

7.4.2.1 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 whether the left and right FIFOs are synchronized. 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.

7.4.2.2 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 */
};
```

7.4.3 S/PDIF RX Driver

The S/PDIF RX driver supports the following features:

- 16, 32, 44.1, 48, 64, and 96 kHz receiving sample rate.
- Signed 24-bit little endian sample format. Due to the S/PDIF SDMA feature, each channel bit length in PCM recorded frame is 32 bits, and only the 24 LSBs are valid.
  In the ALSA subsystem, the supported format is defined as S24_LE.
- Stereo record.
- The device ID can be determined by using the `arecord -l` to list the out record devices.
  For example:

```
root@ ~$ 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: S/PDIF PCM snd-soc-dummy-dai-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 record 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
```

7.4.3.1 Driver design

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

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

For user bits reception, there are two modes for User Channel reception: CD and non-CD. The mode is determined by the `USyncMode` (bit 1 of `CDText_Control` register). User can call the sound control interface to set the mode (see Table 89), 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, the 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.

7.4.3.2 Provided user interface

The S/PDIF RX driver provides interfaces for user application as shown in the table below.

Table 89. 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 obtained 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 33 to use the S/PDIF RX driver. First, the application opens the S/PDIF RX PCM device, waits for the DPLL to lock the input bit stream, and gets the input sample rate. If the USyncMode needs to be set, set it before reading the U/Q bits. Next, set the hardware parameters, including channel number, format and capture sample rate, which is obtained from the driver. Then, call prepare and trigger to startup S/PDIF RX stream read. Finally, call the read function to get the data. During the reading process, applications can read the U/Q bits and channel status from the driver and validate the no good bit.

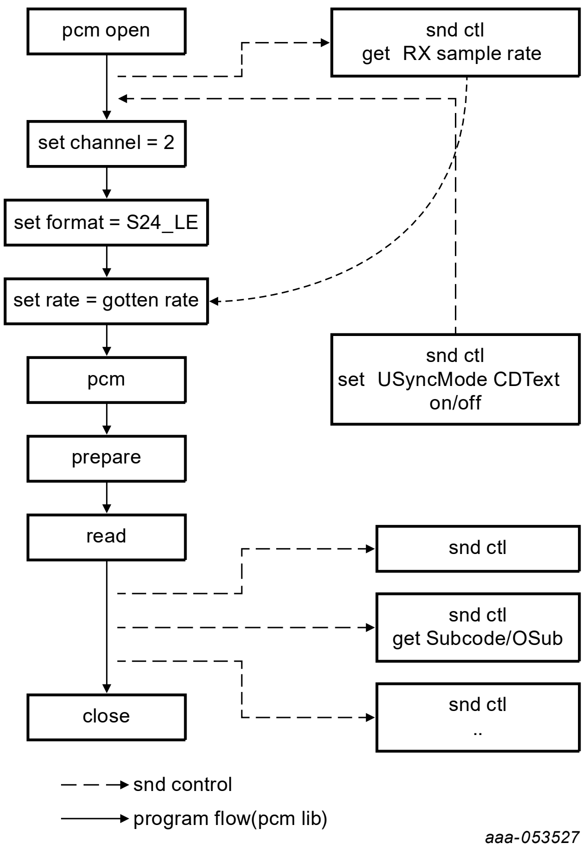

Figure 33. S/PDIF RX application program flow

7.4.4 Source code structure

The table below lists the source files for the driver.

Table 90. S/PDIF driver files

File	Description
`sound/soc/soc-utils.c`	Dummy ALSA SOC codec driver
`sound/soc/fsl/imx-spdif.c`	S/PDIF ALSA SOC machine layer
`sound/soc/fsl/fsl_spdif.c`	S/PDIF ALSA SOC DAI layer
`sound/soc/fsl/imx-pcm-dma.c`	ALSA SOC platform layer
`sound/soc/fsl/imx-pcm.h`	ALSA SOC platform layer header

7.4.4.1 Menu configuration options

The following Linux kernel configurations are provided for this module:

In menu configuration, enable the following module:

CONFIG_SND_IMX_SPDIF: Configuration option for the S/PDIF driver:

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 i.MX boards with S/PDIF**

7.4.4.2 Device tree bindings

See the following documents:

- `Documentation/devicetree/bindings/sound/fsl,spdif.txt`
- `Documentation/devicetree/bindings/sound/imx-audio-spdif.txt`

7.4.4.3 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.

7.4.5 Unit test preparation

To prepare to run a unit test, perform the following actions:

- Setup M-Audio Transit USB sound card by installing M-Audio Transit driver on your PC.
- Install WaveLab tools on your PC.

7.4.5.1 TX test steps

- Plug optical line into the `[line|optical]` port of M-Audio transit.
  Note: *Make sure the `[optical out]` port of M-Audio transit has no output (red light off) after plugging the optical line.*
- Start up WaveLab, press the record button on the toolbar, set up the record file name, sample rate, and channel number, and then record.
- Meanwhile, use the following command to play one wave file on board:

```
#aplay -D hw:[card id],[pcm id] audioXXkYYS.wav
```

- After `aplay` finishes, stop recording in WaveLab.
- Play the recorded WAV file in WaveLab to check.

7.4.5.2 RX test steps

1. Plug optical line into `[optical port]` of M-Audio transit.
2. Start up WaveLab, open a test WAV file: `audioXXkYYS.wav` to play in loop.
3. Meanwhile, use the following command to record one WAV file on board. After finishing the recording, you may play back the record 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 record command must be consistent with the WAV file playing on WaveLab.*

7.5 Audio Mixer (AUDMIX)

7.5.1 Introduction

Many applications require mixing of two or more audios to take different effects. Mixing of two audio streams into a single stream can be done with Audio Mixer. Audio Mixer has two input serial audio interfaces. These are driven by two Synchronous Audio Interface (SAI) modules. Each input serial interface carries 8 audio channels in its frame in TDM manner. Mixer mixes audio samples of corresponding channels from two interfaces into a single sample. Before mixing, audio samples of two inputs can be attenuated based on configuration. The output of the Audio Mixer is also a serial audio interface. Like input interfaces, it has the same TDM frame format. This output is used to drive the serial DAC TDM interface of the audio codec and also sent to the external pins along with the received path of normal audio SAI module for readback by the CPU.

The output of Audio mixer can be selected from any of the three streams:

- Serial audio input 1
- Serial audio input 2
- Mixed audio

Mixing operation is independent of audio sample rate, but the two audio input streams must have the same audio sample rate with the same number of channels in TDM frame to be eligible for mixing.

7.5.2 Block diagram

The following figure shows the high-level view of the Audio Mixer block.

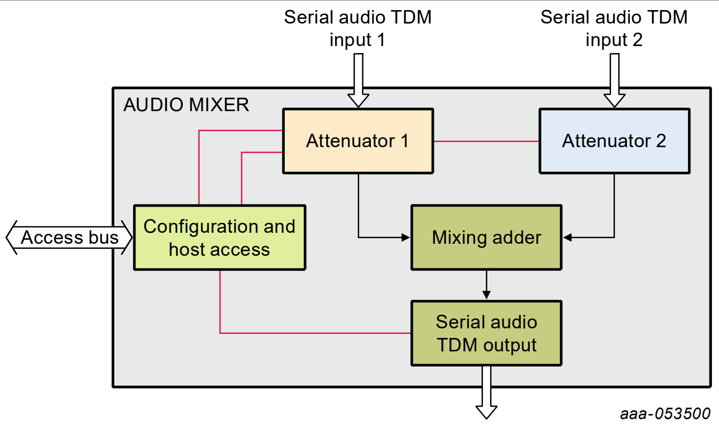

Figure 34. Audio Mixer block diagram

7.5.3 Hardware overview

The Mixer block has two serial audio input interfaces for two audio streams. One of them is used for normal audio and the other is for safety tone. The serial audio TDM frame can contain eight samples of 32 bit each. The first six samples are for three stereo DACs. Each DAC takes two samples for left and right channels. The last two samples are extra and kept for future use. In audio mixing application, the two audio input streams must have the same number of channels and frame rate. The frame format is shown in the following figure.

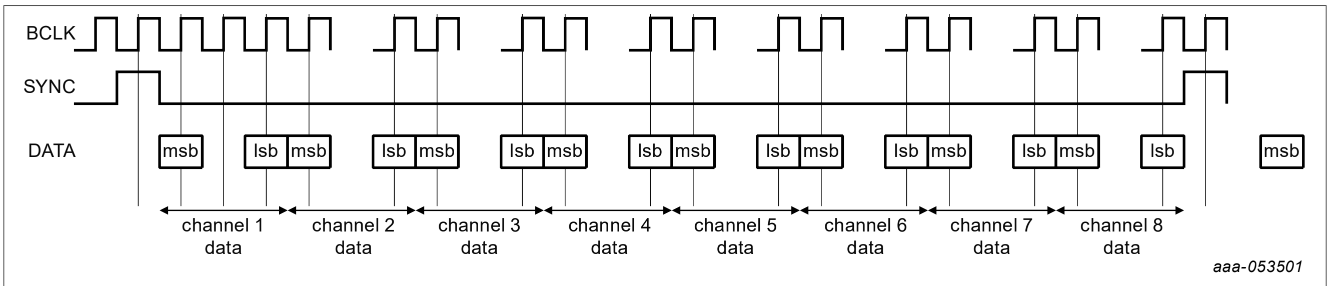

Figure 35. Audio TDM serial interface frame

Input TDM frame is de-serialized as 32-bit samples starting from frame pulse in its own interface bit clock. Each sample passes through the attenuator. Attenuator reduces the level of audio signal. This process is called attenuation. Attenuation of the signal is done by multiplying the audio sample with an attenuation value. The attenuation value defines the level of audio signal at the output of the attenuator. Attenuation can be enabled or disabled. If disabled, the audio sample is passed without modification. If enabled, attenuation is done as per the configuration that defines the attenuation value at different time (called as attenuation profile).

There are two independent attenuators for two audio streams. Output of two attenuators are used for mixing. Mixing is done by adding samples of corresponding channels from two attenuators. The result gives the mixed sample value. It is then quantized to get the desired width of audio sample. The quantized sample is rounded to form the output sample. Rounding is done on LSB of quantized sample. The final sample is then serialized and transmitted in the same frame format as input interfaces with the selected bit clock.

7.5.4 Software overview

The Audio Mixer driver is designed under ALSA System on Chip (ASoC) layer. The ASoC driver for Audio Mixer provides two playback devices for AudioMixer inputs and one capture device to capture the Audio Mixer output. The playback audio format is linear PCM 16-bit, 24-bit, or 32-bit wide audio. The captured audio format is linear PCM audio data, 16-bit, 18-bit, 20-bit, 24-bit, or 32-bit wide.

7.5.4.1 User interface

Audio Mixer interface is accessible from user space by using the `amixer -c <audio mixer card>` tool. The following Audio Mixer controls are exposed to user space.

Table 91. Audio Mixer controls

ID	Name	Type	Access	Value	Default
1	Mixing Clock Source	enum	r/w	#0 'TDM1', #1 'TDM2'	#0 'TDM1'
2	Output Source	enum	r/w	#0 'Disabled', #1 'TDM1', #2 'TDM2', #3 'Mixed'	#0 'Disabled'
3	Output Width	enum	r/w	#0 '16b', #1 '18b', #2 '20b', #3 '24b', #4 '32b'	#4 '32b'
4	Output Clock Polarity	enum	r/w	#0 'Positive edge', #1 'Negative edge'	#1 'Negative edge'
5	Frame Rate Diff Error	enum	r/w	#0 'Unmask', #1 'Mask'	#0 'Unmask'
6	Clock Freq Diff Error	enum	r/w	#0 'Unmask', #1 'Mask'	#0 'Unmask'
7	Sync Mode Config	enum	r/w	#0 'Disabled', #1 'Enabled'	#0 'Disabled'
8	Sync Mode Clk Source	enum	r/w	#0 'TDM1', #1 'TDM2'	#0 'TDM1'
9	TDM1 Attenuation	enum	r/w	#0 'Disabled', #1 'Enabled'	#0 'Disabled'

Table 91. Audio Mixer controls*...continued*

ID	Name	Type	Access	Value	Default
10	TDM1 Attenuation Direction	enum	r/w	#0 'Downward', #1 'Upward'	#0 'Downward'
11	TDM1 Attenuation Step Divider	int	r/w	min=0, max=4095	0
12	TDM1 Attenuation Initial Value	int	r/w	min=0, max=262143	262143
13	TDM1 Attenuation Step Up Factor	int	r/w	min=0, max=262143	174762
14	TDM1 Attenuation Step Down Factor	int	r/w	min=0, max=262143	196608
15	TDM1 Attenuation Step Target	int	r/w	min=0, max=262143	16
16	TDM2 Attenuation	enum	r/w	#0 'Disabled', #1 'Enabled'	#0 'Disabled'
17	TDM2 Attenuation Direction	enum	r/w	#0 'Downward', #1 'Upward'	#0 'Downward'
18	TDM2 Attenuation Step Divider	int	r/w	min=0, max=4095	0
19	TDM2 Attenuation Initial Value	int	r/w	min=0, max=262143	262143
20	TDM2 Attenuation Step Up Factor	int	r/w	min=0, max=262143	174762
21	TDM2 Attenuation Step Down Factor	int	r/w	min=0, max=262143	196608
22	TDM2 Attenuation Step Target	int	r/w	min=0, max=262143	16

7.5.4.2 Source code structure

The following table lists the source files for the driver.

Table 92. Audio Mixer driver files

File	Description
`sound/soc/fsl/fsl_amix.h`	Includes file with common defines
`sound/soc/fsl/fsl_amix.c`	Audio Mixer DAI Driver
`sound/soc/fsl/imx-amix.c`	Audio Mixer Machine Driver
`Documentation/devicetree/bindings/sound/fsl,amix.txt`	Audio Mixer device tree bindings documentation

7.5.4.3 Menu configuration options

The following Linux kernel configurations are provided for this module:

CONFIG_SND_IMX_AMIX: Configuration option for the Audio Mixer Driver

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 i.MX boards with AMIX**

7.6 PDM Microphone Interface (MICFIL)

7.6.1 Introduction

PDM is a popular way to deliver audio from microphones to the processor in several applications, such as mobile telephones. However, current digital-audio systems use multibit audio signal (also known as multibit PCM) to represent the signal. For this purpose, a set of FIR, CIC or/and Half Band filters are usually implemented on DSPs or software. This module implements the required digital interface to provide a 16-bit audio signal from a PDM microphone bitstream with a configurable output sampling rate.

7.6.2 Block diagram

The following figure shows the high-level view of the PDM Microphone Interface block.

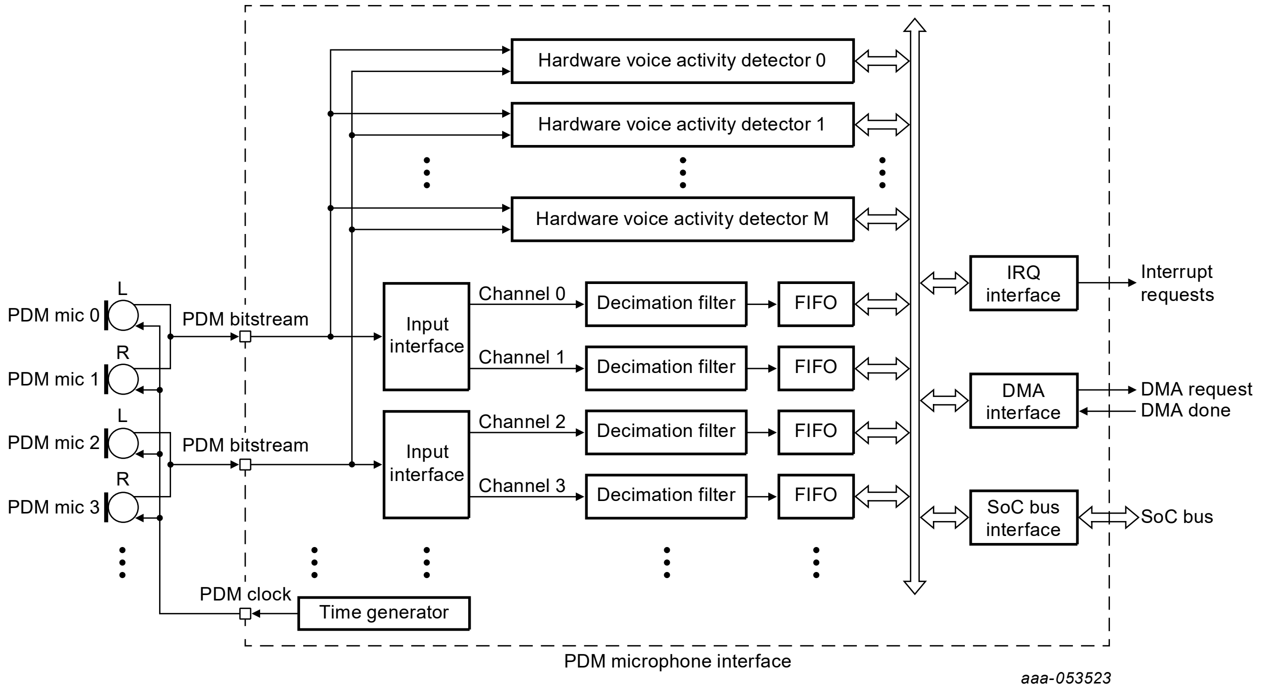

Figure 36. PDM Microphone Interface block

7.6.3 Hardware overview

The implementation of this module is based on the application of digital signal processing techniques in hardware. The PDM Microphone Interface architecture was designed to gate saving and minimal power consumption. It implements a bunch of filters to transform a 1-bit PDM bitstream to a 16-bit PCM signal in the audio band.

To avoid aliasing frequencies in the passband, the overall filter has 80 dB stopband attenuation and passband ripple less than 0.2vdB. The whole module is implemented to work in a multichannel mode. All channels have the same configuration but each input channel could be turned on/off independently.

The PDM Microphone Interface module is composed by the following:

- An input interface for each pair of PDM microphones
- A decimation filter by channel
- A FIFO by channel

- A time generation unit
- Shared interfaces to DMA, interrupts, and SoC
- One or more Hardware Voice Activity Detectors (HWVAD).

The Decimation Filter implements a low-pass filter in the audio band (20Hz-22.5 KHz @ 48 KHz output sampling rate by default) with a configurable decimation rate. It is implemented using an arrange of a CIC, a Half Band, a FIR, and a DC remover filter.

The Time Generator unit generates the PDM clock to the microphones. This clock is the same for all the PDM microphones and it is active for all the microphones, that is, there is not possibility to turn off the PDM clock for one microphone only. It also generates the timing signals and controls for all the filter blocks. The decimation in the filters is also controlled by this block. It activates each block and channel and gives the start signal to FIR FSM and Half Band FSM.

Finally, the output of each Decimation Filter is stored in a FIFO buffer. Each FIFO is mapped in the DATACHn registers. It is possible to generate either an interrupt or a DMA request, when in each FIFO of all enabled channels, the number of data stored surpasses a configured watermark.

7.6.4 Software overview

The PDM Microphone driver is designed under the ALSA System on Chip (ASoC) layer. The ASoC driver for PDM Microphone provides one capture device to capture the PDM Microphone output. The captured audio format is 8-channels 32-bit wide linear PCM audio data @ 48 kHz or 44.1 kHz rate.

7.6.4.1 User interface

PDM Microphone interface is accessible from user space by using the `amixer -c <pdm mic card>` tool. Controls are listed in the following table.

Table 93. PDM Microphone controls

ID	Name	Type	Access	Value	Default
1	CH0 Gain	int	r/w	min=0, max=15	15
2	CH1 Gain	int	r/w	min=0, max=15	15
3	CH2 Gain	int	r/w	min=0, max=15	15
4	CH3 Gain	int	r/w	min=0, max=15	15
5	CH4 Gain	int	r/w	min=0, max=15	15
6	CH5 Gain	int	r/w	min=0, max=15	15
7	CH6 Gain	int	r/w	min=0, max=15	15
8	CH7 Gain	int	r/w	min=0, max=15	15
9	MICFIL Quality Select	enum	r/w	#0 'Medium', #1 'High', #2 'N/A', #3 'N/A', #4 'VLow2', #5 'VLow1', #6 'VLow0', #7 'Low'	#0 'Medium'
10	HWVAD Initialization Mode	enum	r/w	#0 'Envelope mode', #1 'Energy mode'	#0 'Envelope mode'
11	HWVAD High-Pass Filter	enum	r/w	#0 'Filter bypass', #1 'Cut-off @1750Hz', #2 'Cut-off @215 Hz', #3 'Cut-off @102Hz'	#0 'Filter bypass'
12	HWVAD Zero-Crossing Detector Enable	enum	r/w	#0 'OFF', #1 'ON'	#0 'OFF'

Table 93. PDM Microphone controls*...continued*

ID	Name	Type	Access	Value	Default
13	HWVAD Zero-Crossing Detector Auto Threshold	enum	r/w	#0 'OFF', #1 'ON'	#0 'OFF'
14	HWVAD Noise OR Enable	enum	r/w	#0 'Disabled', #1 'Enabled'	#0 'Disabled'
15	HWVAD Sampling Rate	enum	r/w	#0 '48KHz', #1 '44.1KHz'	#0 '48KHz'
16	Clock Source	enum	r/w	#0 'Auto', #1 'AudioPLL1', #2 'AudioPLL2', #3 'ExtClk3'	#0 'Auto'
17	HWVAD Input Gain	int	r/w	min=0, max=15	0
18	HWVAD Sound Gain	int	r/w	min=0, max=15	0
19	HWVAD Noise Gain	int	r/w	min=0, max=15	0
20	HWVAD Detector Frame Time	int	r/w	min=1, max=64	1
21	HWVAD Detector Initialization Time	int	r/w	min=1, max=32	1
22	HWVAD Noise Filter Adjustment	int	r/w	min=1, max=32	1
23	HWVAD Zero-Crossing Detector Threshold	int	r/w	min=1, max=1024	1
24	HWVAD Zero-Crossing Detector Adjustment	int	r/w	min=1, max=16	1

7.6.4.2 Source code structure

The following table lists the source files for the driver.

Table 94. Audio Mixer driver files

File	Description
`sound/soc/fsl/fsl_micfil.h`	Includes file with common defines
`sound/soc/fsl/fsl_micfil.c`	PDM Microphone DAI Driver
`sound/soc/fsl/imx-micfil.c`	PDM Microphone Machine Driver
`Documentation/devicetree/bindings/sound/fsl,micfil.txt`	PDM Microphone device tree bindings documentation

7.6.4.3 Menu configuration options

The following Linux kernel configurations are provided for this module:

CONFIG_SND_IMX_MICFIL: Configuration option for PDM Microphone Driver

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 i.MX boards with micfil**

7.7 Asynchronous Sample Rate Converter (ASRC) on i.MX 8M Nano/i.MX 8M Plus

7.7.1 Introduction

The Asynchronous Sample Rate Converter (ASRC) is a digital module that converts audio from a source sample rate to a destination sample rate.

7.7.2 Hardware operation

The primary features for the ASRC are as follows:

- 4 Contexts: groups of channels with an independent time base
- Fully independent and concurrent context control
- Simultaneous processing of up to 32 audio channels
- Programmable filter characteristics for each context
- 32, 24, 20, and 16-bit fixed point audio sample support
- 32-bit floating point audio sample support
- 8 kHz to 384 kHz sample rate
- 1/16 to 16x sample rate conversion ratio
- Software control of fine conversion ratio

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

For more information, see the ASRC chapter in the Applications Processor documentation associated with the SoC.

7.7.3 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 two scenarios:

- **Memory** -> **ASRC** -> **Memory**: ASRC is controlled by the user application or ALSA plug-in.
- **Memory** -> **ASRC** -> **peripheral**: ASRC is controlled directly by other ALSA drivers.

Note: *The audio driver interaction diagram is the same as Figure 31.*

The ASRC stream interface provides the interface for the user space. The ASRC registers itself under `/dev/mxc_asrc`.

7.7.3.1 Sequence for Memory to ASRC to Memory

1. Open the `/dev/mxc_asrc` device.
2. Request ASRC pair: ASRC_REQ_PAIR.
3. Configure ASRC pair: ASRC_CONIFG_PAIR.
4. Start ASRC: ASRC_START_CONV.
5. 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.
6. Stop ASRC conversion: ASRC_STOP_CONV.
7. Release ASRC pair: ASRC_RELEASE_PAIR.
8. Close the `/dev/mxc_asrc` device.

7.7.3.2 Sequence for Memory to ASRC to Peripheral

Memory to ASRC to peripheral audio path is involved in audio codec driver. In audio sound card, a new device with the name `wm8960audio [cs42888-audio], device 1: HiFi-ASRC-FE (*)` is specified for playback and capture with ASRC. The steps below show the flow of calling ASRC to memory to peripheral:

1. The sound device (PCM) has been registered and starts to enable the DMA channel in the ALSA driver.
2. Request ASRC context: `fsl_easrc_request_context`.
3. Configure ASRC context: `fsl_easrc_config_context`.
4. Enable the DMA channel from Memory to ASRC and from ASRC to Memory.
5. Start DMA channel and start ASRC conversion: `fsl_easrc_start_context`.
6. When audio data playback complete, stop DMA channel and ASRC: `fsl_easrc_stop_context`.
7. Release ASRC pair: `fsl_easrc_release_context`.

7.7.3.3 Source code structure

The table below lists the source files available in `sound/soc/fsl`.

Table 95. ASRC Source File List

File	Description
`sound/soc/fsl/fsl_easrc_m2m.c`	ASRC M2M driver implementation codes
`sound/soc/fsl/fsl_easrc.c`	ALSA CPU DAI driver of ASRC M2P
`sound/soc/fsl/fsl_easrc.h`	Header file for ALSA CPU DAI driver of ASRC P2P
`Sound/soc/fsl/fsl_asrc_dma.c`	ALSA CPU DAI driver of ASRC P2P

7.7.3.4 Menu configuration options

The menu configuration options are as follows:

- Device Drivers
- Sound card support
- Advanced Linux Sound Architecture
- ALSA for SoC audio support
- SoC Audio for Freescale i.MX CPUs
- Enhanced Asynchronous Sample Rate Converter (EASRC) module support. Then the ASRC driver can only be configured with the build-in module.

7.7.3.5 Device tree binding

The functions of device tree bindings for ASRC M2M are as follows:

- `compatible`: Compatible list, which must contain `fsl,imx8mn-easrc`.
- `reg`: Offset and length of the register set for the device.
- `interrupts`: Contains the asrc interrupt.
- `clocks`: Contains an entry for each entry in clock-names.

- `clock`-names: Must contain `mem`.
- `dmas`: Generic DMA devicetree binding as described in `Documentation/devicetree/bindings/dma/dma.txt`.
- `dma-names`: Eight dmas have to be defined, `ctx0_rx`, `ctx0_tx`, `ctx1_rx`, `ctx1_tx`, `ctx2_rx`, `ctx2_tx`, `ctx3_rx`, `ctx3_tx`.
- `fsl,asrc-rate`: A valid sample rate for Back-End (I2S) playback and record.
- `fsl,asrc-format`: A valid sample width for Back-End (I2S) playback and record.

7.7.3.6 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 listed below:

- `pair`: ASRC pair allocated by the IOCTL (`ASRC_REQ_PAIR`).
- `channel_num`: channel number.
- `dma_buffer_size`: buffer size for input and output buffers. The buffer size should be in the unit of page size. Usually, 4k bytes is used.
- `input_sample_rate`: input sampling rate. Input sample rate should be in 5.512k, 8k, 11.025k, 16k, 22k, 32k, 44.1k, 48k, 64k, 88.2k 96k, 176.4k, 192k.
- `output_sample_rate`: output sampling rate. Output sampling rate should be in 32k, 44.1k, 48k, 64k, 88.2k, 96k, 176.4k 192k.
- `input_format`: word format of input audio data. The input data word width can be 16 bit or 24 bit.
- `output_format`: word width of output audio data. The output data word width can be 16 bit or 24 bit.
- `inclk`: none
- `outclk`: none

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 convertion, the driver fills the `output_buffer_length` according to the data number generated by ASRC and copies `output_buffer_length` to `output_buffer_vaddr`. However, before calling `ASRC_CONVERT`, the user is responsible for filling 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 the user-filled `output_buffer_size` to `output_buffer_vaddr`. If the generated buffer size is smaller than user-filled `output_buffer_size` (the difference should be 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.

8 Security

8.1 Cryptographic Acceleration and Assurance Module (CAAM)

8.1.1 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 Sctterlist 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`.

8.1.2 Configuration and Job execution level

This section has two parts:

- Control/Configuration driver
- Job Ring driver

8.1.2.1 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, its 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.

8.1.2.2 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 cannot 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 requests 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 the driver cannot map the job descriptor

8.1.3 API interface level

CAAM 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.

Table 96. Authentication-encryption algorithms

Name	Driver name	Purpose
authenc(hmac(md5),cbc(aes))	authenc-hmac-md5-cbc-aes-caam	Single-pass authentication/encryption using MD5 and AES-CBC
authenc(hmac(sha1),cbc(aes))	authenc-hmac-sha1-cbc-aes-caam	Single-pass authentication/encryption using SHA1 and AES-CBC
authenc(hmac(sha224),cbc(aes))	authenc-hmac-sha224-cbc-aes-caam	Single-pass authentication/encryption using SHA224 and AES-CBC
authenc(hmac(sha256),cbc(aes))	authenc-hmac-sha256-cbc-aes-caam	Single-pass authentication/encryption using SHA256 and AES-CBC
authenc(hmac(sha384),cbc(aes))	authenc-hmac-sha384-cbc-aes-caam	Single-pass authentication/encryption using SHA384 and AES-CBC
authenc(hmac(sha512),cbc(aes))	authenc-hmac-sha512-cbc-aes-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-cbc-des3_ede-caam	Single-pass authentication/encryption using SHA1 and Triple-DES-CBC
authenc(hmac(sha224),cbc(des3_ede))	authenc-hmac-sha224-cbc-des3_ede-caam	Single-pass authentication/encryption using SHA224 and Triple-DES-CBC
authenc(hmac(sha256),cbc(des3_ede))	authenc-hmac-sha256-cbc-des3_ede-caam	Single-pass authentication/encryption using SHA256 and Triple-DES-CBC
authenc(hmac(sha384),cbc(des3_ede))	authenc-hmac-sha384-cbc-des3_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-des-caam	Single-pass authentication/encryption using MD5 and Single-DES-CBC
authenc(hmac(sha1),cbc(des))	authenc-hmac-sha1-cbc-des-caam	Single-pass authentication/encryption using SHA1 and Single-DES-CBC
authenc(hmac(sha224),cbc(des))	authenc-hmac-sha224-cbc-des-caam	Single-pass authentication/encryption using SHA224 and Single-DES-CBC
authenc(hmac(sha256),cbc(des))	authenc-hmac-sha256-cbc-des-caam	Single-pass authentication/encryption using SHA256 and Single-DES-CBC
authenc(hmac(sha384),cbc(des))	authenc-hmac-sha384-cbc-des-caam	Single-pass authentication/encryption using SHA384 and Single-DES-CBC
authenc(hmac(sha512),cbc(des))	authenc-hmac-sha512-cbc-des-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.

Table 97. Symmetric key blockcipher algorithms

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

Table 97. Symmetric key blockcipher algorithms*...continued*

Name	Driver name	Purpose
cbc(des)	cbc-des-caam	Single DES with a CBC mode wrapper
ecb(aes)	ecb-aes-caam	AES with a ECB mode wrapper
ecb(des3_ede)	ecb-3des-caam	Triples DES with a ECB mode wrapper
ecb(des)	ecb-des-caam	Single DES with a ECB mode wrapper
ecb(arc4)	ecb-arc4-caam	ARC4 with a ECB mode wrapper
ctr(aes)	ctr-aes-caam	AES with a CTR 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.

Table 98. Asynchronous hashes

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/random`.

`/dev/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 sourceforge.net/projects/gkernel/files/rng-tools.

8.1.4 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 forces 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 enabled, 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.

8.1.5 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 its 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 architecture-specific port is not upstreamed currently, 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.
- The full CAAM function is exclusive with the Mega/Fast mix-off feature in DSM. If CAAM is enabled, the Mega/Fast mix off feature needs to be disabled, and the user should run `echo enabled > /sys/bus/platform/devices/2100000.aips-bus/2100000.caam/2101000.jr0/power/wakeup` after the kernel boots up, and then Mega/Fast mix will keep the power on in DSM.

8.1.6 Limitations in the existing implementation overview

This section 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 its 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.

8.1.7 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.

8.1.8 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.

8.1.9 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 is 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.

8.1.10 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()`.

8.1.11 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.

8.1.12 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.

8.1.13 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

8.1.14 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

8.1.15 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.

8.1.16 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 depends 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 its 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.
- Encrypts 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 currently.
- Implementation is presently limited to kernel-mode operations.
- One instance is possible currently. 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`

8.1.17 CAAM/SNVS - Security Violation Handling Interface overview

This section 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 (for example, 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 allows the system designer to select how to respond to specific security violation causes using a simple function call written to the system-specific requirements.

8.1.18 Operation

For existing platforms, six security violation interrupt causes are possible within SNVS. Five 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 bootloader 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, a default handler is called. This handler only identifies the interrupt cause to the system console.

8.1.19 Configuration interface

The following interface can be used to define or remove application-defined violation handlers from the driver's dispatch table.

8.1.20 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. If left NULL, the default description string is used.

Returns:

- `Zero` on success.
- `-EINVAL` if an argument was invalid or unusable.

8.1.21 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.

8.1.22 Driver configuration CAAM/SNVS

```
CRYPTO_DEV_FSL_CAAM_SECVIO
```

Enables inclusion of Security Violation driver and configuration interface as part of the build configuration. The driver is not buildable as a module in its present form.

8.2 Display Content Integrity Checker (DCIC)

8.2.1 Introduction

The goal of the DCIC is to verify that a safety-critical information sent to a display is not corrupted.

The DCIC has the following features:

- Pixel clock up to 148.5 MHz
- Configurable polarity of Display Interface control signals
- 24-bit pixel data bus
- Up to 16 rectangular ROIs with a configurable location and size
- Independent CRC32 signature calculation for each ROI

- External controller mismatch indication signal

8.2.2 Source code structure

Table 99. DCIC driver files

File	Description
`drivers/video/fbdev/mxc/mxc_dcic.c`	DCIC driver source code
`include/uapi/linux/mxc_dcic.h`	DCIC User Space Header

8.2.3 Menu configuration options

In menu configuration, enable the following module:

Device Drivers -> **Graphics support** -> **MXC DCIC**

8.2.4 DTS configuration

```
dcic_id = <0>;  /* DCIC device index 0-dcic1, i-dcic2 */
dcic_mux = "dcic-lcdif1"; /* DCIC input select */
```

Table 100. DCIC input select

Module	i.MX 6SoloX	i.MX 6Dual/6Quad

DCIC1	`dcic_lvds`
	`dcic_lcdif1`
		`dcic-ipu0-di1`
		`dcic-lvds0`
		`dcic-lvds1`
		`dcic-hdmi`

DCIC2	`dcic_lvds`
	`dcic_lcdif2`
		`dcic-ipu0-di0/dcic-ipu1-di0`
		`dcic-lvds0`
		`dcic-lvds1`
		`dcic-mipi_dpi`

8.2.5 IOCTLs functions

The DCIC driver supports the following IOCTLs:

- `DCIC_IOC_CONFIG_DCIC`: Configures the DCIC input CLK, VSYNC, HSYNC, and data signal polarity.
- `DCIC_IOC_CONFIG_ROI`: Configures the ROI block size and reference signature.
- `DCIC_IOC_GET_RESULT`: Gets the result of the ROI calculated signature.

8.2.6 Structures

```
struct roi_params {
    unsigned int roi_n;       /* ROI index */
    unsigned int ref_sig;    /* Reference CRC32 */
    unsigned int start_y;    /* start vertical lines of ROI */
    unsigned int start_x;   /* start horizon lines of ROI */
    unsigned int end_y;    /* end vertical lines of ROI */
    unsigned int end_x;    /* end horizon lines of ROI */
    char freeze;                 /*  state of ROI */
};
```

8.2.7 DCIC CRC calculation functions

There are four functions in this unit test to calculate the reference signature:

```
crc32_calc_18of24bit()    /* CRC calculate 18 bit of 24  */
crc32_calc_24bit()        /* CRC calculate 24  */
crc32_calc_24of16bit()    /* CRC calculate 24 bit of 16  */
crc32_calc_18of16bit()    /* CRC calculate 18 bit of 16  */
```

DCIC calculates CRC according to the display bus width, but the display bus width does not always align with bytes per pixel (bpp), and the four functions above can cover different display bus widths and bpps.

8.3 Smart Card Interface - Subscriber Identification Module (SIM)

8.3.1 Introduction

The Subscriber Identification Module (SIM) is designed to facilitate communication to SIM cards or Eurochip prepaid phone cards, and compatible with ISO/IEC 7816-3 standards. The SIM module has one port that can be used to interface with various cards. The interface with the Micro Controller Unit (MCU) is a 32-bit connection as described in the reference document IP Bus Specification.

8.3.2 Modes of operation

The SIM module I/O interface can be operated in one of the three modes of operation summarized below:

- Two-wire interface: Both the IC pin RX and TX are used to interface to the SmartCard.
- External one-wire interface: The IC pins RX and TX are tied together externally to the IC and routed to the SmartCard.
- Internal one-wire interface: The IC pin TX is routed to the SmartCard. The receive pin RX is connected to the TX pin internally to the IC.

8.3.3 External signal description

- `SIM_CLK`: clock that the SIM module provides for the SmartCard. Typical frequencies are 1 MHz to 5 MHz. This clock is 372 times the data rate that is on pin `SIM_TRXD`.
- `SIM_RST_B`: reset the signal from the SIM to the SmartCard.
- `SIM_SVEN`: SmartCard power supply enable control signal.
- `SIM_TRXD`: transmitted/received date from SIM module to SmartCard.
- `SIM_PD`: SmartCard insertion detect.

8.3.4 Source code structure

Table 101. SIM source

File	Description
`drivers/mxc/sim/imx_sim.c`	SIM Driver
`drivers/mxc/sim/imx_envsim.c`	SIM Env

8.3.5 Menu configuration options

Configure the kernel option to enable the module by `menuconfig`:

Device Drivers -> **MXC support drivers** -> **MXC SIM Support**

8.3.6 Software framework

The following figures show the SIM TX and RX software flows.

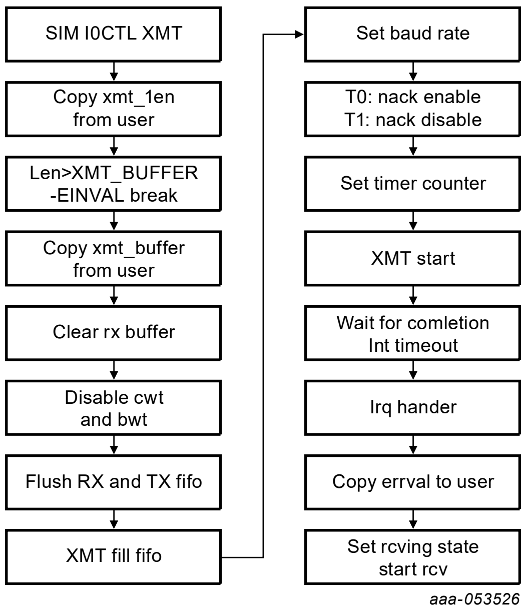

Figure 37. SIM transmitting flow

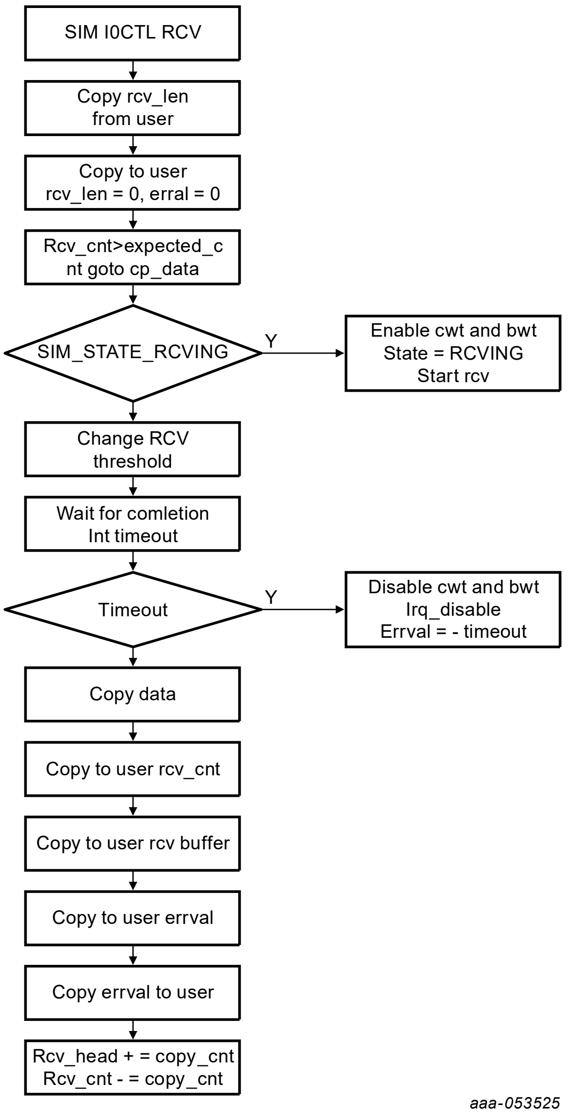
SIM I0CTL RCV
Copy rcv_len from user
Copy to user rcv_len = 0, erral = 0
Rcv_cnt>expected_cnt goto cp_data
SIM_STATE_RCVING
Y
Enable cwt and bwt State = RCVING Start rcv
Change RCV threshold
Wait for comletion Int timeout
Timeout
Y
Disable cwt and bwt Irq_disable Errval = - timeout
Copy data
Copy to user rcv_cnt
Copy to user rcv buffer
Copy to user errval
Copy errval to user
Rcv_head + = copy_cnt Rcv_cnt - = copy_cnt
aaa-053525

Figure 38. SIM receiving flow

8.4 Secure Non-Volatile Storage (SNVS)

8.4.1 Introduction

For more information on Secure Non-Volatile Storage (SNVS), see the i.MX Security Manual for the associated SoC.

SNVS is a block that interfaces with CAAM and SRTC.

For SNVS services related to CAAM, see Driver Configuration CAAM/SNVS.

For SNVS services related to SRTC, see SRTC Introduction

8.5 SNVS Real Time Clock (SRTC)

8.5.1 Introduction

The Real Time Clock (RTC) 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.

8.5.2 Hardware operation

The RTC is a fake timer provided by the system controller firmware. It only supports basic function of read/set time, read/set alarm.

8.5.3 Software operation

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

8.5.4 Driver features

The RTC driver includes the following features:

- Implements the functions required by Linux OS to provide the real time clock and alarm interrupt.
- Alarm wakes up the system from low power modes.

8.5.5 Source code structure

The RTC module is implemented in `drivers/rtc`.

Table 102. RTC driver files

File	Description
`drivers/rtc/rtc-imxdi.c`	i.MX 6 RTC driver
`drivers/rtc/rtc-imx-sc.c`	i.MX 8 RTC System Controller driver
`drivers/rtc/rtc-imx-rpmsg.c`	RPMSG RTC driver

8.5.6 Menu configuration options

In menu configuration, enable the following module:

For i.MX 6, select **Device Drivers** -> **Real Time Clock** -> **Freescale IMX DryIce Real Time Clock**.

For i.MX 8 with SC, select **Device Drivers** -> **Real Time Clock** -> **NXP SC RTC support**.

For RPMSG, select **Device Drivers** -> **Real Time Clock** -> **NXP RPMSG RTC support**.

9 NXP eIQ Machine Learning

9.1 Overview of NXP eIQ Machine Learning

9.1.1 Introduction

Machine learning (ML) is a computer science domain having its roots in the 1960s. ML provides algorithms capable of finding patterns and rules in data. ML is a category of algorithm that allows software applications to become more accurate in predicting outcomes without being explicitly programmed. The basic premise of ML is to build algorithms that can receive input data and use statistical analysis to predict an output while updating outputs as new data becomes available. In 2010, a huge boom started called Deep Learning is a fast-growing subdomain of ML, based on Neural Networks (NN). Inspired by the human brain, Deep Learning has achieved state of the art results in various tasks (for example, computer vision (CV), Natural Language Processing (NLP). Neural Nets are capable of learning complex patterns from millions of examples. Huge adaptation in the embedded world is expected – an area where NXP is a leader. Continuing the effort of enabling its customers, NXP has created NXP eIQ for i.MX, a set of ML tools, which allows developing and deploying ML applications on i.MX 8 QuadMax devices. This section provides an overview of specific areas of NXP eIQ machine learning technology. For detailed execution of machine learning commands, see the *i.MX Machine Learning User's Guide* (UG10166).

9.1.2 OpenCV

OpenCV is an open source computer vision library and one of its modules, called ML, provides traditional machine learning algorithms. Another important module in OpenCV is DNN. It supports neural network algorithms.

OpenCV offers a unified solution for both neural network inference (DNN module) and classic machine learning algorithms (ML module). By including many computer vision functions, OpenCV makes it easier to build complex machine learning applications in a short amount of time and without having dependencies on other libraries.

OpenCV has wide adoption in the Computer Vision field and is supported by a strong and very active community. Key algorithms are specifically optimized for various devices and instructions sets. For i.MX, OpenCV uses Arm NEON acceleration. Arm Neon technology is an advanced SIMD (single instruction multiple data) architecture extension for the Arm Cortex-A series. Neon technology is intended to improve the multimedia user experience by accelerating audio and video encoding/decoding, user interface, 2D/3D graphics or gaming. Neon can also accelerate signal processing algorithms and functions to speed up applications such as audio and video processing, voice and facial recognition, computer vision and deep learning.

At its core, the OpenCV DNN module implements an inference engine and does not provide any functionalities for neural network training. For more details about supported models and supported layers, check the official OpenCV Deep Learning page.

The OpenCV ML module contains classes and functions for solving machine learning problems, such as classification, regression or clustering. It involves algorithms such as support vector machine (SVM), decision trees, random trees, expectation maximization, k-nearest neighbors, classic Bayes classifier, logistic regression, and boosted trees.

9.1.3 Arm Compute

The Arm Compute Library is a collection of low-level functions optimized for Arm CPU and GPU architectures targeted at image processing, computer vision, and machine learning. Arm computer is a convenient repository of optimized functions that developers can source individually or use as part of complex pipelines to accelerate algorithms and applications. Arm compute library also supports NEON acceleration. Arm computer can be shown with examples using DNN models with random weights and inputs and AlexNet using the graph API.

9.1.4 TensorFlow Lite

TensorFlow Lite is a light-weight version of and a next step from TensorFlow. TensorFlow Lite is an open source software library focused on running machine learning models on mobile and embedded devices (available at www.tensorflow.org/lite). It enables on-device machine learning inference with low latency and small binary size. TensorFlow Lite also supports hardware acceleration using the Android OS Neural Networks API. TensorFlow Lite supports a set of core operators (both quantized and floating point) tuned for mobile platforms. They incorporate pre-fused activations and biases to further enhance the performance and quantized accuracy. Additionally, TensorFlow Lite also supports the use of custom operations in models.

TensorFlow Lite defines a new model file format, based on FlatBuffers. FlatBuffers is an open source, efficient, cross-platform serialization library. It is similar to protocol buffers, but the primary difference is that FlatBuffers do not need a parsing/unpacking step for a secondary representation before you can access the data, often coupled with per-object memory allocation. Also, the code footprint of FlatBuffers is an order of magnitude smaller than protocol buffers.

TensorFlow Lite has a new mobile-optimized interpreter, which has the key goal to keep apps lean and fast. The interpreter uses static graph ordering and a custom (less-dynamic) memory allocator to ensure minimal load, initialization, and execution latency.

9.1.5 ONNX Runtime

ONNX Runtime is an open source inference engine framework developed by Microsoft supporting ONNX model format. ONNX Runtime runs on CPU with NEON and it also supports GPU/NPU hardware accelerators using the execution providers. For more details about ONNX Runtime, see the official ONNX Runtime project webpage.

9.1.6 PyTorch

PyTorch is an open source machine learning library used for applications, such as computer vision and natural language processing. It is free and open-source software. PyTorch provides a Python package for high-level features like tensor computation. For more details about PyTorch, see the official webpage www.pytorch.org.

9.1.7 DeepViewRT

DeepViewRT is a proprietary neural network inference engine optimized for NXP microprocessors and microcontrollers. It not only implements its own compute engine, but is also able to use popular 3rd-party ones.

9.1.8 TVM

TVM is an open deep learning compiler stack for CPUs, GPUs, and specialized accelerators. It aims to close the gap between the productivity-focused deep learning frameworks, and the performance-oriented or efficiency-oriented hardware backends.

10 Data Plane Development Kit (DPDK)

10.1 Introduction

Data Plane Development Kit (DPDK) is a user space packet processing framework.

The following sections provide instructions for installing and configuring the user space DPDK v22.11 software. Besides highlighting the applicable platforms, this guide provides steps for compiling and executing sample DPDK applications in a Linux application (`linuxapp`) environment over i.MX boards.

10.1.1 Supported platforms and platform-specific details

DPDK supports i.MX 8M Mini, i.MX 8M Plus, i.MX 8DXL, i.MX 91, i.MX 93, i.MX 95, i.MX 943, and various Layerscape family of SoCs. This section describes the port layout of their Design Boards. Port layout information is especially relevant while executing DPDK applications to map DPDK port number to physical ports.

The following provides the i.MX board-specific information.

10.1.1.1 i.MX 8M Mini EVK (i.MX 8MM)

8MM refers to the i.MX 8M Mini platform. For more information on i.MX 8MM, see i.MX 8 Series Applications Processors and i.MX 8M Mini.

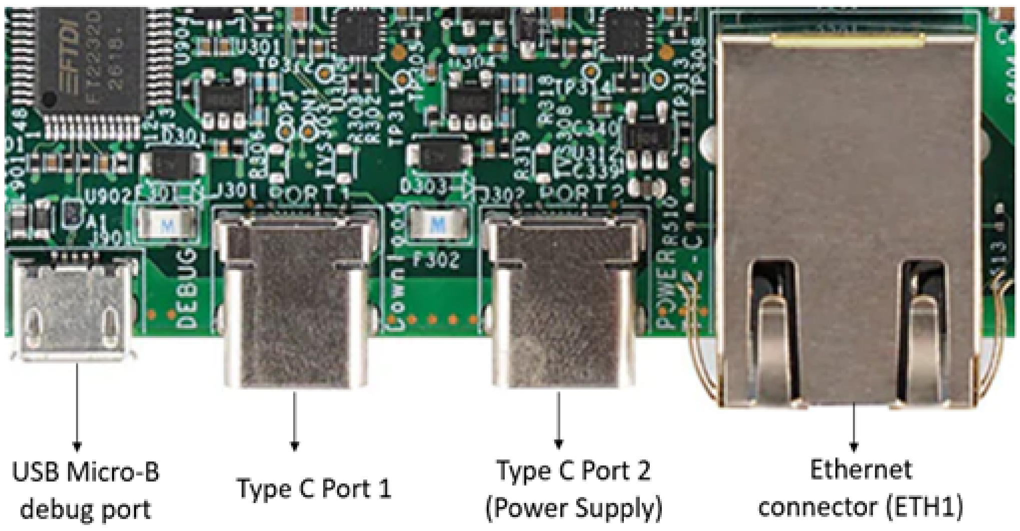

Figure 39. i.MX 8M Mini port layout

Table 103. Label on case

Label on case	DPDK vdev port name
Eth1	`net_enetfec`

10.1.1.2 i.MX 8M Plus EVK (i.MX 8MP)

8MP refers to the i.MX 8M Plus platform. For more information on i.MX 8MP, see i.MX 8 Series Applications Processors and i.MX 8M Plus.

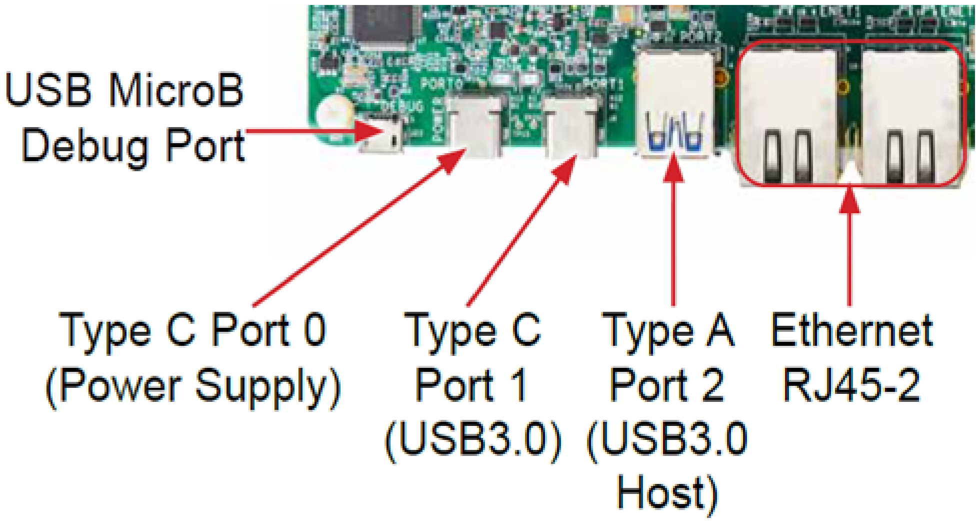

Figure 40. i.MX 8M Plus port layout

Table 104. Label on case

Label on case	DPDK vdev port name
Eth1	`net_enetfec`
Eth2	`net_enetqos`

10.1.1.3 i.MX 8DXL EVK (i.MX 8DXL)

i.MX 8DXL refers to the NXP i.MX 8DXL EVK platform. For more information, see i.MX 8 Series Applications Processors and i.MX 8XLite Evaluation Kit.

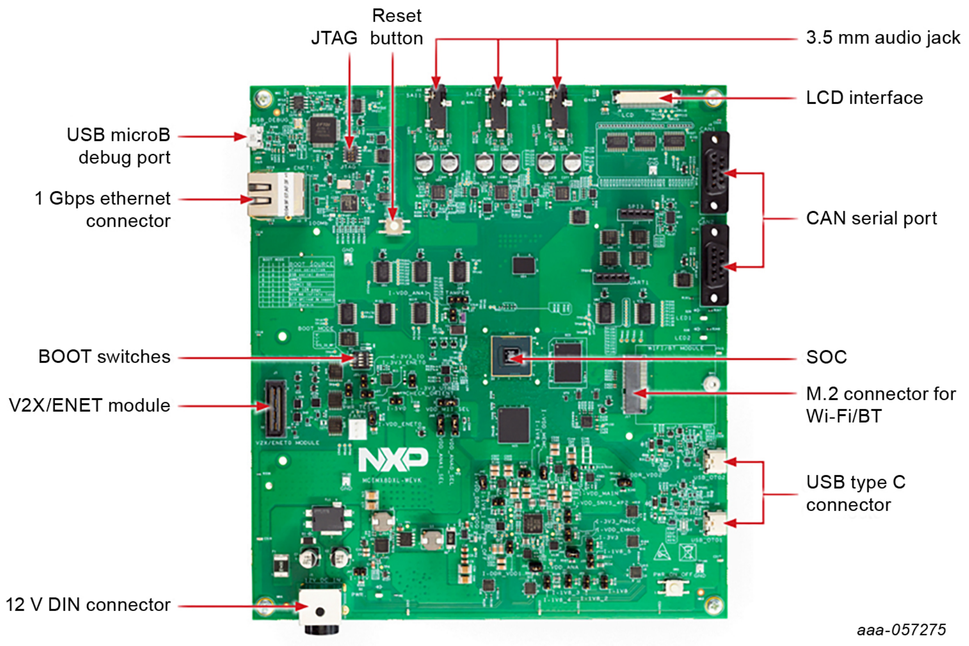

Figure 41. i.MX 8DXL EVK port layout

Table 105. Label on case

Label on case	DPDK vdev port name
ENET1	`net_enetqos`

10.1.1.4 i.MX 91 EVK (i.MX 91)

i.MX 91 refers to the NXP i.MX 91 EVK platform. For more information, see i.MX 9 Applications Processors and i.MX 91.

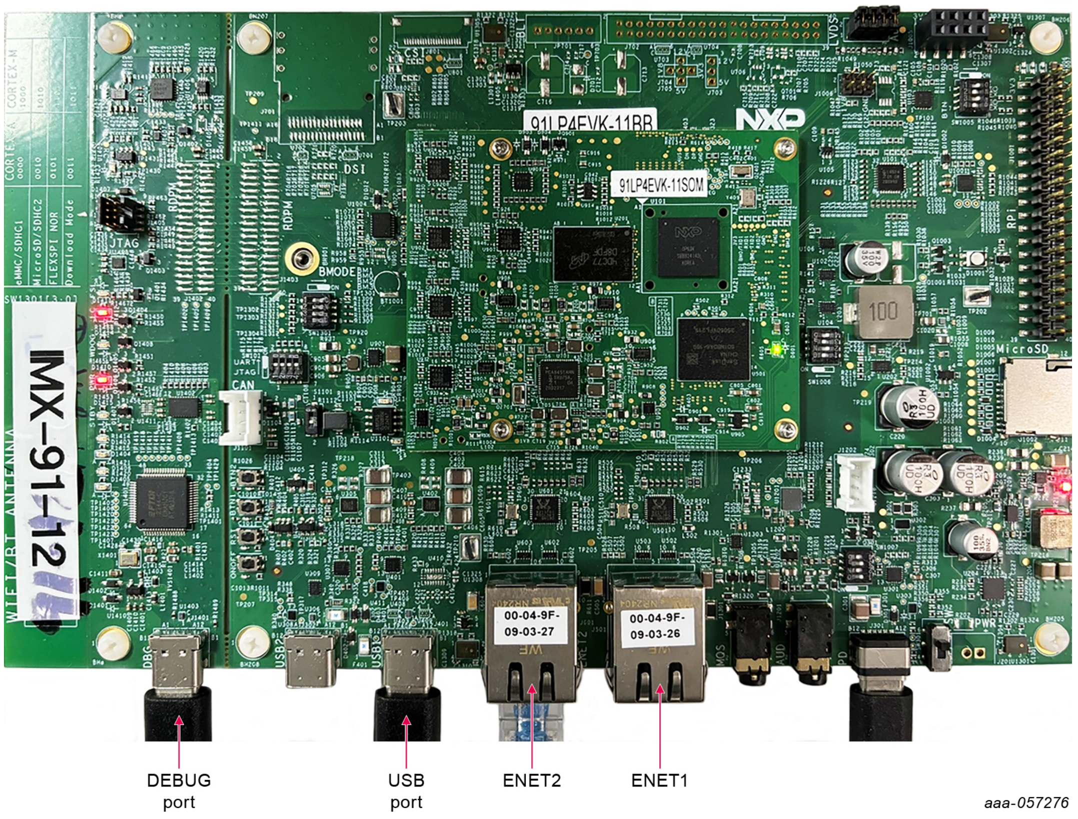

Figure 42. i.MX 91 EVK port layout

Table 106. Label on case

Label on case	DPDK vdev port name
ENET1	`net_enetqos`
ENET2	`net_enetfec`

10.1.1.5 i.MX 93 EVK (i.MX 93)

i.MX 93 refers to the NXP i.MX 93 EVK platform. For more information, see i.MX 93 Evaluation Kit.

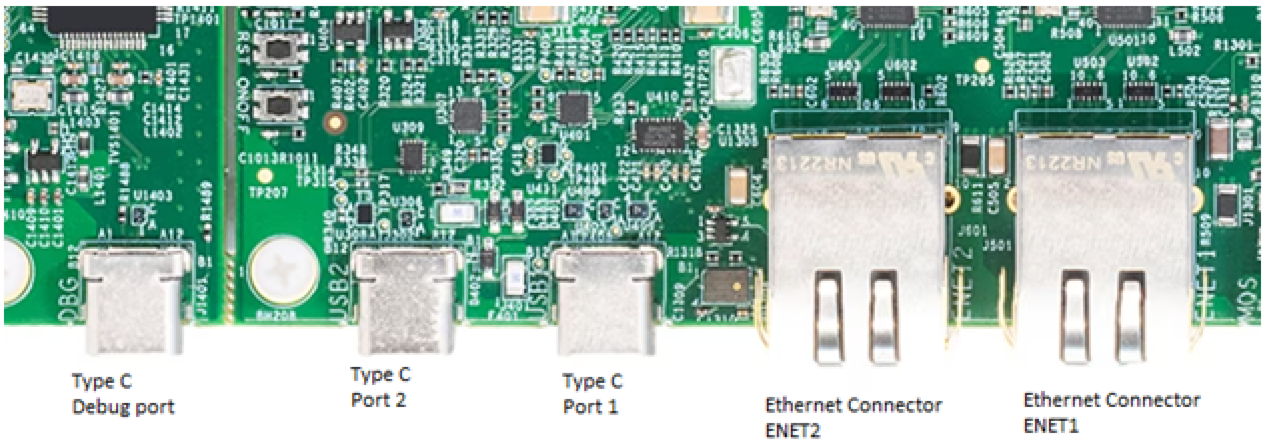

Figure 43. i.MX 93 port layout

Table 107. Label on case

Label on case	DPDK vdev port name
ENET1	`net_enetqos`
ENET2	`net_enetfec`

10.1.1.6 i.MX 95 EVK (i.MX 95)

i.MX 95 refers to the NXP i.MX 95 EVK platform. For more information, see i.MX 95 Applications Processor Family.

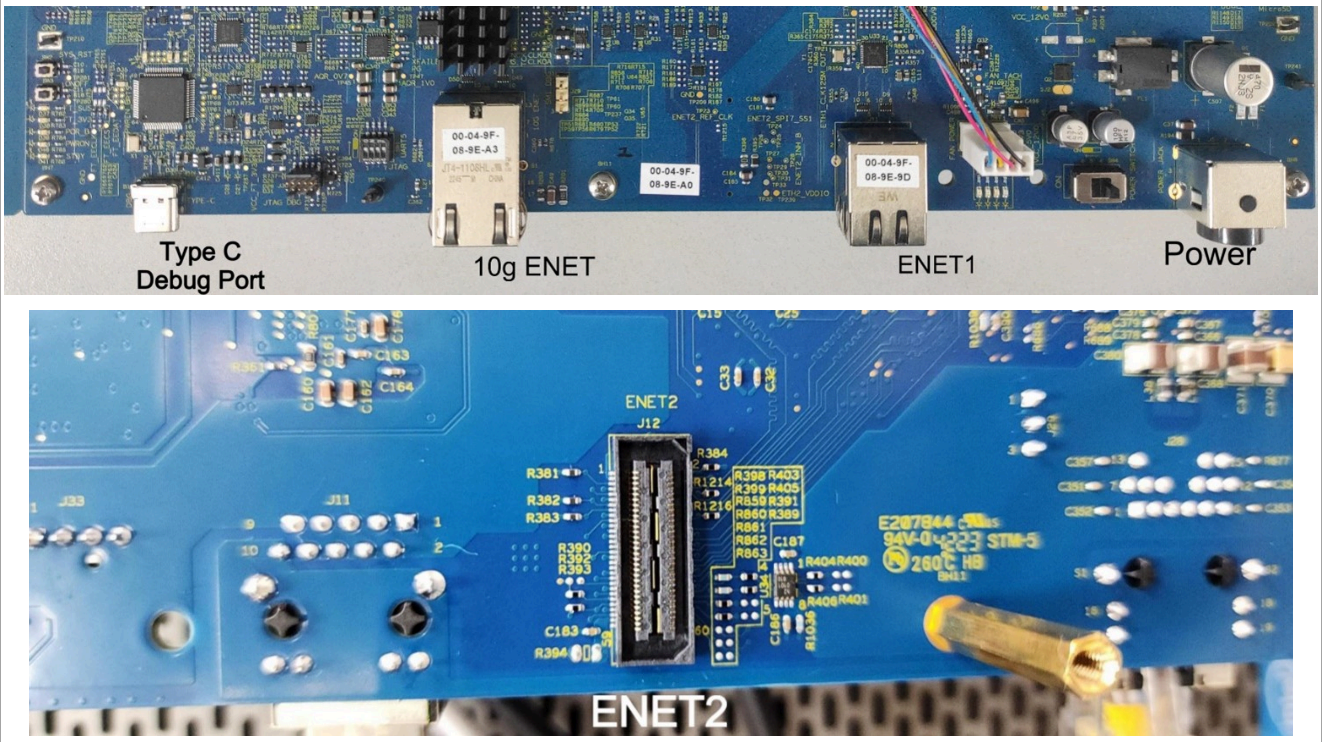

Figure 44. i.MX 95 port layout

Table 108. Label on case

Label on case	DPDK port name
ENET1	0002:00:02.0
10g ENET	0002:00:12.0
ENET2	0002:00:0a.0

Note:

- *DPDK port names are the PCI addresses for the Ethernet ports and users might see a different name on the board.*
- *Users need to use the `lspci` command to know the PCI addresses on the respective boards.*
- *TJA1103 or TJA1120 daughter card is required for enablement of the ENET2 port.*

10.1.1.7 i.MX 943 EVK (i.MX 943)

i.MX 943 refers to the NXP i.MX 943 EVK platform. For more information, see i.MX 9 Applications Processors and i.MX 94 Applications Processor.

Table 109. Label on case

Label on case	DPDK port name
2.5G SGMII5/6	0000:00:00.0
1G RGMII 3	0001:01:08.0
1G RGMII 4	0001:01:10.0

10.1.2 References

Table 110. DPDK application references

Sample application	DPDK Web manual link	Description
Layer-2 Forwarding (l2fwd)	l2fwd usage	Layer 2 Forwarding sample application setup and usage guide.
Layer-3 Forwarding (l3fwd)	l3fwd usage	Layer 3 Forwarding sample application setup and usage guide.
PMD Test Application (testpmd)	testpmd_usage	Guide for test application which can be used to test all PMD supported features.
DPDK Web Guide	DPDK documentation	Link to DPDK Web Manual containing information about all supported PMD and Applications.

Table 111. Release references

Component	Base upstream release version
DPDK	22.11

10.2 DPDK overview

The key goal of the DPDK is to provide a simple, complete framework for fast packet processing in data plane applications. Using the APIs provided as part of the framework, applications can use the capabilities of the underlying network infrastructure.

The framework creates a set of libraries for target environments, layered through an Environment Abstraction Layer (EAL), which hides all the device glue logic beneath a set of consistent APIs. These environments are created by using configuration files. Once the EAL library is created, the user may link with the library to create their own applications. Various other libraries, outside EAL, including the Hash, Longest Prefix Match (LPM) and rings libraries are also available for performing specific operations. Sample applications are also provided to help understand various features and uses of the DPDK framework.

DPDK implements a run-to-completion model for packet processing where all resources must be allocated prior to calling data plane applications, running as execution units on logical processing cores. In addition, a pipeline model may also be used by passing packets or messages between cores via rings. This allows work to be performed in stages, resulting in more efficient use of code on cores.

Data Plane Development Kit (DPDK) is a user space packet processing framework (under the Linux Foundation), comprised of various user space libraries and drivers for fast packet processing. DPDK uses various techniques to optimize the packet throughput. How it works and the key to its performance are based on the Fast-Path and Poll Mode Driver (PMD).

Fast-Path (Kernel bypass): A fast-path is created from the NIC to the application within user space, in turn, bypassing the kernel. This eliminates context switching when moving the frame between user space/ kernel space. Additionally, further gains are also obtained by negating the kernel stack/network driver and the performance penalties they introduce.

Poll Mode Driver: Instead of the NIC raising an interrupt to the CPU when a frame is received, the CPU runs a poll mode driver to constantly poll the NIC for new packets. However, this does mean that a CPU core must be dedicated and assigned to running PMD.

More information on general working of DPDK can be found through DPDK website.

10.2.1 DPDK platform support

This section describes the NXP Data Path Acceleration architecture. See the diagram below.

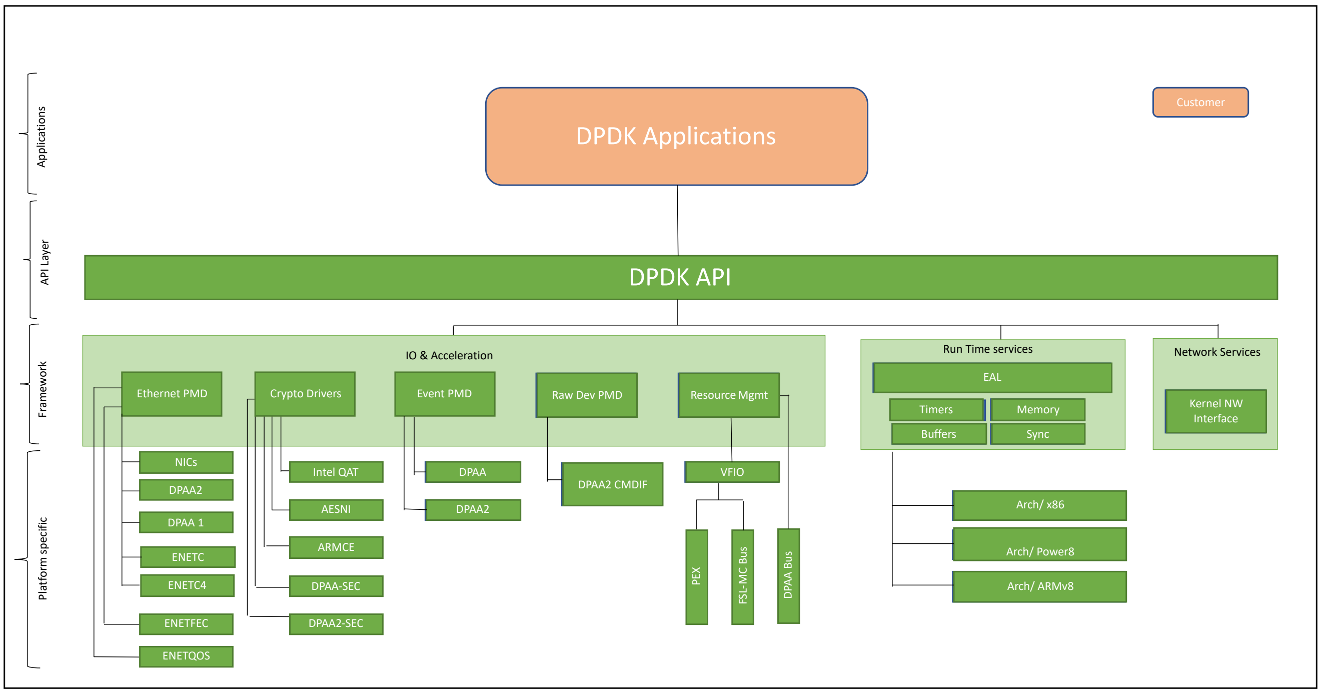

Figure 45. DPDK architecture with NXP components

10.2.2 Supported DPDK features

The following is the list of DPDK NIC features on i.MX 8M Mini, i.MX 8M Plus, i.MX 8DXL, i.MX 91, i.MX 93, and i.MX 943:

- Basic statistics
- Packet type parsing
- Promiscuous mode
- L3/L4 checksum offload
- Linux
- Arm v8

Applications:

- `dpdk-l2fwd`
- `dpdk-l3fwd`

- `dpdk-testpmd` (RX only, TX only and forward modes)
- `dpdk-fpr` (only for i.MX 93)
- `dpdk-pdump` (only for i.MX 943)
- `dpdk-proc-info` (only for i.MX 943)

Limitations:

- Multi-queue is not supported.

10.2.3 Supported DPDK features on ENETC (i.MX 95)

The following is the list of DPDK NIC features:

- Basic stats
- Packet type parsing
- Promiscuous mode
- Multicast Promiscuous mode
- L3/L4 checksum offload
- Link speed/ status support
- Linux
- Arm v8
- MAC exact filter table filtering
- VLAN exact filter table filtering
- VLAN promiscuous
- Multi-queue supported
- Link status interrupt
- MTCP stack supported
- TSN QBV supported

Applications:

- `dpdk-l2fwd`
- `dpdk-l3fwd`
- `dpdk-testpmd` (RX only, TX only and forward modes)
- `dpdk-pdump`
- `dpdk-proc-info`
- `dpdk-ip_fragmentation`
- `dpdk-ip_reassembly`
- `dpdk-fpr`
- `dpdk-test`
- `dpdk-ipsec-secgw`
- `dpdk-test-crypto-perf`

Other Supported Network interfaces:

- PCI 1g card
- Tap and Virtio interfaces

10.3 Build DPDK

This section includes two subsections, which detail:

- Building DPDK binaries (libraries and sample applications) using the Yocto build system.

- Building DPDK binaries as standalone package, through DPDK's own build system.

10.3.1 Build DPDK using Yocto

DPDK is one of the application packages of the Yocto build system. This section describes a method to build DPDK as a standalone package within the Yocto environment. It is assumed that the Yocto environment has already been configured before executing the commands below. See Download Yocto layers for complete details of using the Yocto build system.

After the Yocto environment has been set up, the following commands can be used to build DPDK applications and libraries. Generated files (libraries and binaries) would be available in the `<yocto_sdk>/ bld-<Name>/ tmp/work/<Machine>-poky-linux/dpdk/` folder. After the `rootfs` (root filesystem) is generated, the binaries would be merged into it.

`bitbake dpdk`: It is assumed `setup-environment` was run before running this command.

See Build Yocto images for packing these binaries into the target `rootfs` using the Yocto build system. Yocto environment by default compiles DPDK and place it in the rootfs.

Single image of DPDK binary supports Layerscape and i.MX platforms. Once the DPDK package has been installed, binaries would be available in the `/usr/bin` folder in the rootfs.

`/usr/bin` contains the sample applications listed in DPDK Application References.

At various places in this document, the binaries above would be referred for representing execution and other information. It is assumed that execution is being done either using the PATH variable set, as explained above, or with the absolute path to the binaries.

The following table lists various DPDK example applications, which are available in the Yocto generated rootfs.

Table 112. DPDK example applications

File/Image name related to DPDK	Description

`/usr/bin/dpdk-l2fwd`
`/usr/bin/dpdk-l3fwd`
`/usr/bin/dpdk-testpmd`
`/usr/bin/dpdk-pdump`
`/usr/bin/dpdk-proc-info`
	DPDK Example applications and PMD test application.

`/usr/bin/dpdk-ip_fragmentation`
`/usr/bin/dpdk-ip_reassembly`
`/usr/bin/dpdk-fpr`
`/usr/bin/dpdk-ipsec-secgw`
`/usr/bin/dpdk-test`
`/usr/bin/dpdk-test-crypto-perf`
	Supported for i.MX 95 only.
	`dpdk-fpr` for i.MX 93 and i.MX 95.

10.3.2 Standalone build of DPDK libraries and applications

This section describes the steps required to build DPDK binaries (libraries and example applications) in a standalone environment. This environment can either be on a host enabled for cross building for Layerscape/ i.MX boards or directly on the Layerscape/i.MX target board.

Note: *This section primarily focuses on standalone building of DPDK on a host machine using cross compilation for Layerscape/i.MX boards as target. Though, necessary notes have been added to enable compilation directly on target boards. See Download Yocto layers for creating an environment suitable for building DPDK on Layerscape/i.MX boards.*

For instructions on how to build DPDK using Yocto system, see Section 10.3.1.

Obtaining the DPDK source code

The DPDK source code contains all the necessary libraries for build example applications as well as test applications. The source code also includes various configuration and scripts for supporting build and execution. Obtain the DPDK source code using the link below:

```
git clone https://github.com/NXPmicro/dpdk.git
```

Once the above repository has been cloned, DPDK source code is available for compilation. This source is common for the Layerscape and i.MX platforms.

Prerequisites before compiling DPDK

Before compiling DPDK as a standalone build, following dependencies need to be resolved independently:

- Platform compliant and compiled Linux Kernel source code so that KNI modules can be built.
  - This is optional and if KNI module support is not required, this can be ignored.
  - For details of compiling platform-compliant Linux kernel, see Download Yocto layers and Build Yocto images
  - For disabling KNI module, see the notes below.
- OpenSSL libraries required for building software crypto driver (OpenSSL PMD).
  - This is optional and if the software crypto driver support is not required, this dependency can be ignored.
  - OpenSSL package needs to be separately compiled and libraries installed at a known path before DPDK build can be done.

Follow the steps below to build OpenSSL as a standalone package:

```
git clone https://github.com/nxp-qoriq/qoriq-components_openssl -b integration
# Clone the OpenSSL source code
cd openssl
# Change into cloned directory
git checkout LSDK-21.08
# Checkout the specific Tag supported by DPDK
```

Export the Cross Compilation tool chain for building OpenSSL for target. The following step for exporting cross compilation toolchain is required only when compiling on Host. On a target board, it is assumed default build toolchain would be used.

```
export CROSS_COMPILE=<path to uncompressed toolchain archive>/bin/aarch64-linux-
gnu-
```

Configure the OpenSSL build system with the following command. The --prefix argument specifies a path where OpenSSL libraries would be deployed after build completes. This is also a path, which would be provided to DPDK build system for accessing the compiled OpenSSL libraries.

```
./Configure linux-aarch64 --prefix=<OpenSSL library path> shared
make depend
make
make install
export OPENSSL_PATH=<OpenSSL library path>
```

Note:

When building DPDK on target board, it is possible that OpenSSL libraries required by DPDK are already available as part of the rootfs, in which case external compilation of OpenSSL package would not be required.

To disable OpenSSL PMD support, see the notes below.

Compiling DPDK using Meson

Follow the steps below to compile DPDK once the above prerequisites are resolved. These steps are common for all platforms and are needed only when cross compiling on a host for all boards as target. In case of direct compilation on target boards, it is assumed that prerequisites would be satisfied using the root filesystem.

1. Set up the environment for compilation:
   a. This step is required only in the host environment where default toolchain is not for target boards. When compiling on a target board, this step can be skipped.
   ```
   export CROSS_PATH=<path to cross-compile toolchain>
   export PATH=$PATH:$CROSS_PATH
   ```
   b. Set up OpenSSL path for software crypto drivers (OpenSSL PMD). This is optional and can be skipped in case software crypto driver (OpenSSL PMD) support is not required. These external variables can also be used to pass other required libraries for example `libpcap`.
2. Use DPDK build system for compiling DPDK.
   Note: *DPDK binaries generated using below steps are compatible for Layerscape and i.MX platforms. This is also valid when DPDK is build through Yocto build system.*
   a. Execute the following command:
   ```
   meson arm64-build --cross-file config/arm/<config_file> -Dexamples =
    <list of example applications to be compiled separated by commas> -
   Dprefix=<location to install DPDK>
   ninja -C arm64-build
   ```
   Here, `-Dprefix` and `-Dexamples` are optional parameters. `Dprefix` parameter is used to deploy all the DPDK binaries (libraries and example applications) to a standard Linux package-specific layout within a directory represented by this parameter. Alternatively, a directory `dpdk/arm64-build/` is also created, and binaries and libraries are also available in it. `install` parameter is also not required in the `ninja` command, if installation is not required. Example is used to compile required examples. To compile only drivers, this parameter is not needed.
   Note: *The `config_file` here should be `arm64_imx_linux_gnu_gcc` for i.MX 93, i.MX 91, and i.MX 95 boards (Cortex-A55 platforms).*
   For other platforms: `arm64_dpaa_linux_gnu_gcc` should be used.
   b. Once the example applications are compiled, the binaries are available in the DPDK build directory with prefix `dpdk`:
   ```
   dpdk/arm64-build/examples/*
   ```
   Besides the example application above, DPDK also provides some sample test applications, which can be used for comprehensive verification of the DPDK driver (PMD) features for available and compatible devices. These sample test applications are compiled by default during the DPDK source compilation. These are available in the `dpdk/arm64-build/app/` directory.

10.4 Flashing the target board

The Universal Update Utility (UUU) runs on a Windows or Linux OS host and is used to download images to different devices on an i.MX board.

For how to flash the target board, see the section “Universal update utility” in the *i.MX Linux User's Guide* (UG10163).

10.5 Running DPDK

10.5.1 Test environment setup

Various sample application execution steps are described in the following sections. The following figure shows the setup containing the DUT (Device Under Test) and the Packet Generator (Spirent packet generator). This is applicable for the commands provided in following section. The setup includes a one-to-one link between DUT and Packet generator unit. DPDK application running on the DUT is expected to forward the traffic.

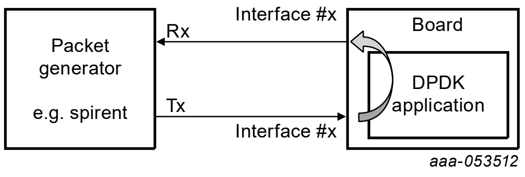

Figure 46. DPDK application setup

10.5.2 Prerequisites to boot the board

The following steps must be performed on the U-Boot for all the i.MX platforms.

10.5.2.1 Update bootargs

Bring up the board with proper images and add the following parameters in `mmcargs`:

```
U-Boot> edit mmcargs
edit: <default_board_specific_args> <hugepages> <imx-95-params> <imx-mem-
params>;
U-Boot > saveenv
```

- `default_board_specific_args` is the default boot argument. For example, it can be like:

  ```
  mmcargs=setenv bootargs ${jh_clk} console=${console} root=${mmcroot}
  ```

- `hugepages` can be used to add the hugepages:

  ```
  default_hugepagesz=2m hugepagesz=2m hugepages=256
  ```

  Note:
  The number of hugepages can be increased/decreased, which depends on the system memory availability and application requirements.
  Only 2MB size hugepages are supported on all i.MX SoCs except for i.MX 95. The i.MX 95 also supports 1G size hugepages along with 2MB size pages. It is mandatory to have a few 2MB size hugepages even if the application needs only 1G size hugepages.
  This is because Ethernet RX/TX rings only use 2MB size hugepage memory. Therefore, it needs to reserve the number of RX and TX rings + addtional a few (at least 4) 2MB size hugepages as well as along with 1G size hugepages. Addtional 2MB hugepages are required for the DPDK real library to reserve the memory from 2MB size hugepages.
  For example, if the application needs to use one 1G hugepage and 6 RX + 6 TX queues, at least one 1G and 162MB size pages must be reserved:

  ```
  default_hugepagesz=1024m hugepagesz=1024m hugepages=1 hugepagesz=2m
   hugepages=16
  ```

  On the kernel console, to use both hugepages, mount the hugepages as follows:

  ```
  mkdir /mnt/huge_1G
  mkdir /mnt/huge_2M
  mount -t hugetlbfs -o pagesize=1G none /mnt/huge_1G
  mount -t hugetlbfs -o pagesize=2M none /mnt/huge_2M
  ```

- `imx-95-params` is only for the i.MX 95 platform:

  ```
  iommu.passthrough=1
  ```

- `imx-mem-params` is used to reserve the initial DDR memory from U-Boot through the `mem` environment variable. It needs to be set on platforms with FEC/eQoS Ethernet interfaces as the BD ring of FEC/eQoS interfaces can only accept 32-bit addresses.

```
mem=2096M
```

For optimized performance on the i.MX platforms, perform the following steps:

1. Add `isolcpus=<core_list>` in `mmcargs` to add the CPUs planned to be used for DPDK applications.

   ```
   Where <core_list> is:
   For i.MX 8MM, 8MP: 1-3
   For i.MX 8DXL: 1
   For i.MX 93: 1
   For i.MX 95: 1-5
   ```

   Note:
   - *i.MX 8DXL and i.MX 93 have only 2 cores, but to reserve the core 1 for DPDK, we can provide `core_list=1` for both platforms.*
   - *i.MX 91 has only core 0, so the option `isolcpus` cannot be used.*
2. Add the following parameters in `mmcargs` to set the CPU not to enter idle mode:

   ```
   `cpuidle.off=1 cpufreq.off=1`
   ```

3. Set the system in performance mode (applicable only if the system does not support the option `cpufreq.off=1` in bootargs):

   ```
   echo performance | sudo tee /sys/devices/system/cpu/cpu*/cpufreq/
   scaling_governor
   grep . /sys/devices/system/cpu/cpu*/cpufreq/scaling_governor
   ```

Note:

If the governor is not supported, check the `lpm-mode`. For example, for the i.MX 93 board:

```
root@imx93evk:~# cat /sys/devices/platform/imx93-lpm/mode
System is in OD mode with DDR 3733 MTS!

You can set it as :
root@:~# echo 0 > /sys/devices/platform/imx93-lpm/mode
System switching to OD mode...
root@:~# cat /sys/devices/platform/imx93-lpm/mode
System is in OD mode with DDR 3733 MTS!
```

For ENET-FEC and ENET-QOS on all i.MX platforms:

Check the properties such as `reset-gpios`, `reset-assert-us`, and `reset-deassert-us` for the Ethernet node on U-Boot using the following command:

```
fdt print
```

For details of the `fdt` command, see Section 10.5.2.2.

If these properties are present in the Ethernet node, remove it as it may cause issues in the forwarding. Use the following command on the U-Boot:

```
=> fdt rm /<soc>/<bus>/<ethernet>/mdio/<ethernet-phy>/ "property";
        eg. fdt rm /soc@0/bus@42800000/ethernet@42890000/mdio/ethernet-phy@2
 "reset-gpios";
```

10.5.2.2 Device Tree file

For the i.MX platforms, the DPDK-specific Device Tree file should be used to boot up the board. This Device Tree file is configured to provide user space applications with network interfaces. Once the Device Tree configuration mentioned above is used, all the Ethernet ports would be available in the user space only except for the i.MX 95 platform. Changes to the Device Tree file would be required to assign some of the ports to Linux Kernel.

Users can use the following method to replace the default `dtb` to support DPDK on the i.MX platforms.

Execute these commands on U-Boot for i.MX 8M Mini:

```
U-Boot > setenv fdtfile imx8mm-evk-dpdk.dtb
U-Boot > saveenv
U-boot > boot
```

Execute these commands on U-Boot for i.MX 8M Plus:

```
U-Boot > setenv fdtfile imx8mp-evk-dpdk.dtb
U-Boot > saveenv
U-boot > boot
```

Execute these commands on U-Boot for i.MX 8DXL:

```
U-Boot > setenv fdtfile imx8dxl-evk.dtb; setenv loadkernel 'fatload mmc
 ${mmcdev}:${mmcpart} ${loadaddr} Image';
U-Boot > setenv loadfdt 'fatload mmc ${mmcdev}:${mmcpart} ${fdt_addr}
 ${fdt_file}';
U-Boot > run loadkernel; run loadfdt; fdt addr ${fdt_addr};
U-Boot > fdt set /bus@5b000000/ethernet@5b050000 compatible "fsl,imx-enet-qos"
 "snps,dwmac-5.10a";
U-Boot > setenv bootcmd 'mmc dev ${mmcdev};run mmcargs; booti ${loadaddr} -
 ${fdt_addr};'
```

Execute these commands on U-Boot for i.MX 91:

```
U-Boot > setenv fdtfile imx91-11x11-evk.dtb
U-Boot > setenv loadkernel 'fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} Image'
U-Boot > setenv loadfdt 'fatload mmc ${mmcdev}:${mmcpart} ${fdt_addr}
 ${fdtfile}';
U-Boot > run loadkernel; run loadfdt; fdt addr ${fdt_addr};
U-Boot > fdt set /soc@0/bus@42800000/ethernet@42890000 compatible "fsl,imx8mm-
fec-uio";
U-Boot > fdt set /soc@0/bus@42800000/ethernet@428a0000 compatible "fsl,imx-enet-
qos" "snps,dwmac-5.10a";
U-Boot > fdt resize 4096;
U-Boot > fdt set /aliases "ethernet1" "/soc@0/bus@42800000/ethernet@428a0000";
U-Boot > setenv bootcmd 'mmc dev ${mmcdev}; run mmcargs; booti ${loadaddr} -
 ${fdt_addr};'
```

Execute these commands on U-Boot for i.MX 93:

```
U-Boot > setenv fdtfile imx93-11x11-evk.dtb
U-Boot > setenv loadkernel fatload mmc ${mmcdev}:${mmcpart} ${loadaddr} Image;
U-Boot > setenv loadfdt 'fatload mmc ${mmcdev}:${mmcpart} ${fdt_addr_r}
 ${fdtfile}';
U-Boot > run loadkernel; run loadfdt; fdt addr ${fdt_addr_r};
```

```
U-Boot > fdt set /soc@0/bus@42800000/ethernet@42890000 compatible "fsl,imx8mm-
fec-uio";
U-Boot > fdt rm /soc@0/bus@42800000/ethernet@42890000/mdio/ethernet-phy@2
 "reset-gpios";
U-Boot > fdt rm /soc@0/bus@42800000/ethernet@42890000/mdio/ethernet-phy@2
 "reset-assert-us";
U-Boot > fdt rm /soc@0/bus@42800000/ethernet@42890000/mdio/ethernet-phy@2
 "reset-deassert-us";
U-Boot > fdt set /soc@0/bus@42800000/ethernet@428a0000 compatible "fsl,imx-enet-
qos" "snps,dwmac-5.10a";
U-Boot > fdt resize 4096;
U-Boot > fdt set /aliases "ethernet1" "/soc@0/bus@42800000/ethernet@428a0000";
U-Boot > fdt rm /soc@0/bus@42800000/ethernet@428a0000/mdio/ethernet-phy@1
 "reset-gpios";
U-Boot > fdt rm /soc@0/bus@42800000/ethernet@428a0000/mdio/ethernet-phy@1
 "reset-assert-us";
U-Boot > fdt rm /soc@0/bus@42800000/ethernet@428a0000/mdio/ethernet-phy@1
 "reset-deassert-us";
U-Boot > run mmcargs; sleep 1; booti ${loadaddr} - ${fdt_addr};
```

Execute these commands on U-Boot for i.MX 95 (Cortex-A core):

```
U-Boot > setenv fdtfile imx95-19x19-evk.dtb
U-Boot > saveenv
U-boot > boot
```

Execute these commands on U-Boot for i.MX 95 (Cortex-M7 core):

```
U-Boot > setenv fdtfile imx95-19x19-evk-netc-rpmsg.dtb
U-Boot > saveenv
U-boot > boot
```

Execute these commands on U-Boot for i.MX 943 (SGMII ports):

```
U-Boot > setenv fdtfile imx943-evk-sgmii.dtb
U-Boot > saveenv
U-Boot > boot
```

10.5.3 Prerequisites for running DPDK applications on Linux OS

When the Linux OS is up, perform the following steps before running the DPDK applications:

1. Load the non-cacheable module.

   ```
   modprobe kpage_ncache.ko
   ```

2. Manually allocate hugepages if required.

   ```
   echo 448 > /sys/kernel/mm/hugepages/hugepages-2048kB/nr_hugepages
   ```

   Note: *Users may not be required to allocate hugepages if they are already added from the kernel bootargs. It can be checked using:*

   ```
   cat /proc/cmdline
   ```

10.5.4 Executing DPDK applications

This section describes how to execute DPDK sample applications on i.MX platforms.

10.5.4.1 Executing DPDK applications on i.MX 8M Mini, i.MX 8M Plus, i.MX 8DXL, i.MX 91, and i.MX 93 platforms

Note:

- *The following command snippets assume that the commands are executed while being present in `/usr/bin` or appropriate PATH variable has been set.*
- *i.MX 8M Mini supports ENETFEC Ethernet interface.*
- *i.MX 8DXL supports ENETQOS Ethernet interface.*
- *i.MX 8M Plus, i.MX 93, and i.MX 91 support ENETFEC and ENETQOS Ethernet interfaces.*
- *User must provide the `--vdev` argument with the value `net_enetfec` to enable ENETFEC Ethernet device and/or `--vdev` argument with the value `net_enetqos` to enable ENETQOS Ethernet device for DPDK applications.*

l2fwd– Layer 2 forwarding application

Sample application to show forwarding between multiple ports based on the Layer 2 information:

```
dpdk-l2fwd -c 0x1 -n 1 --vdev 'net_enetfec' -- -p 0x1
dpdk-l2fwd -c 0x1 -n 1 --vdev 'net_enetqos' -- -p 0x1
dpdk-l2fwd -c 0x3 -n 1 --vdev 'net_enetqos' --vdev 'net_enetfec' -- -p 0x3 -T 0
 -P
```

In the command above: `-c` refers to the core mask for the cores to be assigned to DPDK. `-p` is the port mask for the ports to be used by the application. Other command line parameters may also be provided. For a complete list, see L2Forwarding Sample Application (in Real and Virtualized Environments).

Note:

DPDK L2fwd application periodically prints the I/O stats. To avoid CPU core to be interrupted because of these scheduled prints, the `-T 0` option can be appended atthe end of the command line.

l3fwd– Layer 3 forwarding application

Sample application to show forwarding between multiple ports based on the Layer 3 information:

```
dpdk-l3fwd -c 0x1 --vdev='net_enetfec' -n 1 -- -p 0x1 --config="(0,0,0)" -P --
 parse-ptype
dpdk-l3fwd -c 0x1 --vdev='net_enetqos' -n 1 -- -p 0x1 --config="(0,0,0)" -P --
 parse-ptype
dpdk-l3fwd -c 0x3 -n 1 --vdev 'net_enetqos' --vdev 'net_enetfec' -- -p 0x3 -P --
config="(0,0,1)(1,0,1)" --parse-ptype
```

In the command above: `-c` refers to the core mask for the cores to be assigned to DPDK. `-p` is the port mask for the ports to be used by application. `--config` is the `(Port, Queue, Core)` configuration used by the application for attaching cores to queues on each port. Other command line parameters may also be provided. For a complete list, see L3Forwarding Sample Application.

dpdk-testpmd

Sample application used for the functionality test. It ensures that the traffic generator to board connectivity is proper. You may run `dpdk-testpmd` in `tx_only` mode to validate, if the packets are going out on specific interfaces. For the information about the `testpmd` application and its supported arguments, see webdocumentation.

- For TX only:

```
dpdk-testpmd -c 0x1 -n 1 --vdev='net_enetfec' -- -i --portmask=0x1 --nb-ports=1
 --forward-mode=rxonly
```

```
dpdk-testpmd -c 0x3 -n 1 --vdev 'net_enetqos' -- -i --portmask=0x2 --nb-ports=1
 --forward-mode=txonly
```

- For RX only:

```
dpdk-testpmd -c 0x1 -n 1 --vdev='net_enetfec' -- -i --portmask=0x1 --nb-ports=1
 --forward-mode=txonly
dpdk-testpmd -c 0x1 -n 1 --vdev 'net_enetqos' -- -i --portmask=0x2 --nb-ports=1
 --forward-mode=rxonly
```

- For IO:

```
dpdk-testpmd -c 0x1 -n 1 --vdev=’net_enetfec’ -- -i --portmask=0x1 --nb-ports=1
 --forward-mode=io
dpdk-testpmd -c 0x1 -n 1 --vdev 'net_enetqos' -- -i --portmask=0x2 --nb-ports=1
 --forward-mode=io
dpdk-testpmd -c 0x3 -n 1 --vdev 'net_enetqos' --vdev 'net_enetfec' -- -i --
portmask=0x3 --nb-ports=2 --forward-mode=io
```

10.5.4.2 Executing DPDK applications on i.MX 95 platform

On i.MX 95, DPDK supports VF for both `enetc-1g` and `enetc-10g` interfaces.

Note:

Users must provide the `--vdev` argument with the value `net_tap0` for tap interface and `virtio_user0` for the i.MX 95 platform in TAP/Virtio-ENETC use-case.

10.5.4.2.1 Setting up the ENETC Ethernet interfaces

Note: *In the following steps, we are creating VF0 and running DPDK on it.*

Perform the following steps to set up each ENETC interface:

1. Configure multiple queues (if required as per usecase).
   Devlink utilty in Linux OS can be used to configure multiple queues in the initialization process of PF. The number of rings allocated to each SI is at least 1.
   a. Example to set the number of rings of PF to **1**, VF0 to **6**, and VF1 to **1**:

   ```
   devlink dev param set pci/0002:00:10.0 name si_num_rings cmode driverinit
    value 0106
   ```

   Note:
   Here, `0002:00:10.0` is the PF PCI address. The `value` is a string type value. Numerically higher VFs have lower bytes of the string. For example, if one ENETC supports 2 VFs, and the string is set to "304" or "0304", then the ring allocation for each VF is shown below:

   ```
   VF0 = 0x04
   VF1 = 0x03
   ```

   The number of rings of PFs is not explicitly presented in the string, because the remaining number of rings is reserved for PF.
   b. Run the devlink reload command to apply the new configuration.

   ```
   devlink dev reload pci/0002:00:10.0
   ```

   c. Verify the value of `si_num_rings`.

   ```
   devlink dev param show pci/0002:00:10.0 name si_num_rings
   ```

Example output:

```
pci/0002:00:10.0:
name si_num_rings type driver-specific
values:
cmode driverinit value 0106
```

Note: *For Arm Cortex-M7, VF0 is configured to support 3 queues and VF1 is configured to support 4 queues and it is not user configurable.*

2. Bind VFs to the DPDK.

```
echo 1 > /sys/bus/pci/devices/0002\:00\:00.0/sriov_numvfs
```

Note: *Use `dpdk-devbind.py -s` to identify the PCI address of the 1g and 10g ports, like `0002:00:00.0` in the example above.*

Check `dmesg` logs to identify the PCI address of the VF created.

Example kernel logs for VF:

```
[ 2002.156985] fsl_enetc_vf 0002:00:12.0: enabling device (0000 -> 0002)
[ 2002.517997] uio_pci_generic 0002:00:12.0: enabling device (0000 -> 0002)
[ 2002.543684] uio_pci_generic 0002:00:12.0: No IRQ assigned to device: no
 support
for interrupts?
```

Use `dpdk-devbind.py` to bind VF to DPDK:

```
dpdk-devbind.py -b uio_pci_generic <vf-pci-addr>
```

Note: *Users must use the `igb_uio` module instead of `uio_pci_genric` for link status interrupts. According to the logs above, `0002:00:12.0` is the `<vf-pci-addr>`.*

3. Enable trust for the specified VF user by specifying the `PF eth interface`. (Not required on the Cortex-M7.)

```
ip link set <eth-interface> vf 0 trust on
```

Check the `eth` interface using:

```
eg. ./dpdk-devbind.py -s [In below o/p eth0 is the PF eth interface]
 0002:00:00.0 'Device e101' if=eth0 drv=fsl_enetc4 unused=vfio-
pci,uio_pci_generic
```

10.5.4.2.2 Running DPDK on i.MX 95

Execute the following command to run the DPDK application on i.MX 95:

```
./dpdk-l2fwd -c 0x3 -n 1 -- -p 0x1
./dpdk-l3fwd -c 0x2 -n 1 -- -p 0x1 -P --config="(0,0,1)"
./dpdk-testpmd -c 0x3 -n 1 -- -i --portmask=0x1
./dpdk-ip_fragmentation -c 0x2 -n 1 -- -p 0x1
./dpdk-ip_reassembly -c 0x2 -n 1 -- -p 0x1
```

Examples to run DPDK with multi-queues:

```
./dpdk-l3fwd -c 0x3f -n 1 -- -p 0x1 -P --config="(0,0,1) (0,1,2) (0,2,3) (0,3,4)
 (0,4,5) (0,5,0)"
./dpdk-testpmd -c 0x3f -n 1 -- -i --portmask=0x1 --rxq=5 --txq=5
```

10.5.4.2.3 OpenSSL-based applications

OpenSSL-based applications include the following:

- **`dpdk-test` application:**
  The `dpdk-test` application is a command-line interface that facilitates running various tests or test suites. It can be used to do functional testing of the drivers and libraries. The binary is tested for various algos and protocols in Crypto PMDs.
  ```
  dpdk-test --vdev 'crypto_openssl' --log-level=6
  RTE>> cryptodev_openssl_asym_autotest
  ```
  For more details, see dpdk-test.
- **`dpdk-test-crypto-perf` application:**
  The `dpdk-test-crypto-perf` tool is a DPDK utility that allow to measure the performance parameters of PMDs available in the Crypto tree. There are two measurement types available: throughput and latency. This application is tested using the `crypto_perf_test.sh` script.
  ```
  crypto_perf_test.sh openssl
  ```
  For more details, see dpdk-test-crypto-perf.
- **IPsec Gateway `dpdk-ipsec-secgw` application:**
  General setup:
  - For IPsec application, two DUTs need to be configured as endpoint 0 (ep0) and endpoint 1 (ep1).
  - Connect Port 1 and Port 1 of the ep0 and ep1 to each other (back-to-back).
  - Connect Port 0 and Port 0 of the ep0 and ep1 to packet generator (for example, Spirent).

  The sample configurations for both ep0 and ep1 are available at `/etc/dpdk/`.
  Custom port mappings, SA/SP and the routes can be configured in the corresponding configuration file named `ep0.cfg` and `ep1.cfg` for respective endpoint.
  - Endpoint 0 command:
    ```
    dpdk-ipsec-secgw -c 0x8 --vdev 'crypto_openssl' -- -p 0x2 -P --
    config="(1,0,3)" -f /etc/dpdk/ep0_64X64.cfg
    ```
  - Endpoint 1 command:
    ```
    dpdk-ipsec-secgw -c 0x8 --vdev 'crypto_openssl' -- -p 0x2 -P --
    config="(1,0,3)" -f /etc/dpdk/ep1_64X64.cfg
    ```

  For more details, see dpdk-ipsec-secgw.

10.5.4.2.4 PSI-VSI MBOX messaging

DPDK supports VF to send messages to PF by the VSI-TO-PSI messaging mechanism. It can be used for supporting features not under VF's control.

Currently, the following features have been supported using this mechanism:

- VF primary MAC based filtering
- Enable/Disable Promiscuous mode
- Enable/Disable Allmulti MAC mode
- Get link status
- Get link speed
- MAC Exact Match filtering
- VLAN Exact Match filtering
- Enable/Disable VLAN promiscous mode
- Link status interrupts

1. Follow Section 10.5.4.2.1.
2. (Optional) Control the timeout and delay time counters with `env.` variables.

   ```
   export ENETC4_VSI_WAIT_DELAY_UPDATE=<val>
   export ENETC4_VSI_WAIT_TIMEOUT_UPDATE=<val>
   ```

   Note: *Using these parameters, users have control over the waiting time for PSI to process message sent by VSI.*
3. Enable trust for the specified VF user by specifying the `PF eth interface`. (Not required on the Cortex-M7.)

   ```
   ip link set <eth-interface> vf 0 trust on
   ```

   Check the `eth` interface using:

   ```
   eg. ./dpdk-devbind.py -s [In below o/p eth0 will be the PF eth interface]
   0002:00:00.0 'Device e101' if=eth0 drv=fsl_enetc4 unused=vfio-
   pci,uio_pci_generic
   ```
4. Run the `dpdk-testpmd` application:

   ```
   ./dpdk-testpmd -c 0x3 -n 1 -- -i --portmask=0x1
   ```
5. These features can be tested using `dpdk-testpmd` using its runtime functions below:

   ```
   mac_addr set <port_id> <valid_mac_address> /* Set Primary MAC */
   set promisc <port_id> off/on               /* Enable/Disable promiscuous mode
    */
   set allmulti <port_id> off/on              /* Enable/Disable allmulti MAC
    mode */
   show port info <port_id>                   /* Get link status/speed */
   ```

   See webdocumentation for the information on the `dpdk-testpmd` application runtime functions.

To add the MAC adddress exact match entry:

1. Disable the promiscuous mode to test the MAC entries filtering.

   ```
   set promisc <port_id> off
   ```
2. Run the following command to add the exact match MAC address entry:

   ```
   mac_addr add <port_id> <valid_mac_address>
   ```

To add/remove the VLAN exact match entry:

Run the following commands on the `dpdk-testpmd` the command line interface to disable the VLAN promisc mode and set the VLAN filter mode for the port:

```
vlan set filter on <port_id>
```

Note: *By default, the VLAN promisc mode is enabled.*

Once the VLAN promisc is off, add the exact match VLAN entries:

```
rx_vlan add <vlan_id> <port_id>
```

If required, the entry can be removed using the following command:

```
rx_vlan rm <vlan_id> <port_id>
```

To enable promisc mode and unset VLAN filter mode for the port:

```
vlan set filter off <port_id>
```

Note: *Currently, the maximum MAC and VLAN address exact match filter table entries that can be added are 4 per ENETC.*

To register link status interrupts:

DPDK is dependant on the `igb_uio` module instead of `uio_pci_genric` for interrupts.

Run the following commands to bind the device to `igb_uio`:

```
insmod igb_uio.ko or modprobe igb_uio
echo igb_uio > /sys/bus/pci/devices/0002\:00\:12.0/driver_override
echo 0002:00:12.0 > /sys/bus/pci/drivers/fsl_enetc_vf/unbind
echo 0002:00:12.0 > /sys/bus/pci/drivers/igb_uio/bind
```

Run `dpdk-testpmd` to check the status interrupts:

```
./dpdk-testpmd -c 0xc --log-level=*:debug -n 1 -- -i --portmask 0x1
```

Open a new terminal and try to change the link status using either the following commands or attach and de-attach the cable:

```
$ ifconfig eth1 down
$ ifconfig eth1 up
```

Note: *Here,* `eth1` *is the PF interface.*

Then users can observe the link change notifications on the console. It may take a few seconds to get the notification of link up.

10.5.4.2.5 Running DPDK on VF1

i.MX 95 supports maximum two VFs. To test DPDK with second VF, perform the following steps. In this use-case, two VFs are created, VF0 is owned by the kernel and VF1 is owned by DPDK.

1. Run the `devbind.py` script for the device ID.
   ```
   ./dpdk-devbind.py -s
   ```
2. Create two VFs, one for DPDK and one for the kernel.
   ```
   echo 2 > /sys/bus/pci/devices/0002\:00\:10.0/sriov_numvfs
   ```
3. Check `dmesg` logs to identify the PCI address of the VF1 created.
   ```
   Example kernel logs for VF:
   [107.122861] pci 0002:00:12.0: [1131:ef00] type 00 class 0x020001
   [107.230531] fsl_enetc_vf 0002:00:12.0: enabling device (0000 -> 0002)
   [ 107.232173] pci 0002:00:14.0: [1131:ef00] type 00 class 0x020001
   [ 107.342547] fsl_enetc_vf 0002:00:14.0: enabling device (0000 -> 0002)
   ```
4. Use `dpdk-devbind.py` to bind VF1 to DPDK:
   ```
   dpdk-devbind.py -b uio_pci_generic <vf1-pci-addr>
   ```
   As per the logs above, `0002:00:14.0` is the `<vf1-pci-addr>`.

5. Enable trust for the specified VF user by specifying the `PF eth interface`.

   ```
   ip link set <eth-interface> vf 1 trust on
   ```

   Check `eth` interface using:

   ```
   eg. ./dpdk-devbind.py -s [In below o/p eth0 is the PF eth interface]
   0002:00:00.0 'Device e101' if=eth0 drv=fsl_enetc4 unused=vfio-
   pci,uio_pci_generic
   ```

6. Run the following commands:

   ```
   ./dpdk-l2fwd -c 0x3 -n 1 -- -p 0x1
   ./dpdk-l3fwd -c 0x2 -n 1 -- -p 0x1 -P --config="(0,0,1)"
   ```

Note:

- *i.MX 95 supports the maximum frame size to 2000 Bytes.*
- *Availability of Multiple queues is dependent on the ENETC PF driver (Arm core kernel driver or Cotex-M7 core driver).*
- *RSS is enabled on using multiple queues.*
  - *The RSS secret key is a random key.*
  - *Enabled RSS hash functions are IP source address, IP destination address, UDP/TCP source port, and UDP/TCP destination port.*
- *Receive queues are equally configured in the RSS indirection table to receive flows.*
- *Hardware Packet type parsing for supported packets is enabled by default.*
- *Checksum validation offload is enabled by default.*
- *Only 1G Intel Gigabit PCI-E Network Adapter EXPI9301CTBLK is supported.*
- *Only ENETC 10G interface is supported with the Cortex-M7 image.*
- *`ip_fragmentation` and `ip_reassembly` can receive only up to 2000 bytes.*
- *Link status change notifications are supported with the `igb_uio` framework only. It may take a few seconds to get the link status UP notification.*

10.6 Executing DPDK applications on i.MX 943 platform

Note: *Only the internal ENETC port supports VF, which is connected to the switch CPU port. External ENETC ports do not support VF.*

1. Create and configure the bridge with two switch ports:

   ```
    ip link add name br0 type bridge
    ip link set dev swp0 master br0
    ip link set dev swp1 master br0
    ip link set dev br0 up
    ip link set dev swp0 up
    ip link set dev swp1 up
   ```

2. Create 2 Virtual Functions (VFs):

   ```
   echo 2 > /sys/bus/pci/devices/0000:00:00.0/sriov_numvfs
   ```

3. Bind the VFs to the DPDK-compatible driver:

   ```
    dpdk-devbind.py -b uio_pci_generic 0000:00:08.0
    dpdk-devbind.py -b uio_pci_generic 0000:00:10.0
   ```

4. Enable trusted mode for both VFs:

   ```
    ip link set eth0 vf 0 trust on
   ```

```
  ip link set eth0 vf 1 trust on
```

5. Add static forwarding entries to the bridge's forwarding database (FDB):

   ```
   bridge fdb add 02:00:00:00:00:00 dev swp0 master static
   bridge fdb add 02:00:00:00:00:01 dev swp1 master static
   ```

   Note: *This means that if a packet has destination MAC 02:00:00:00:00:00, the bridge forwards it out `swp0`. If the destination is 02:00:00:00:00:01, it goes out through swp1. For `dpdk-l2fwd`, the destination MAC address is replaced by 02:00:00:00:00:TX_PORT_ID, hence this is set here. For `dpdk-l3fwd`, add the static MAC address through the command line. `dpdk-testpmd` sets the same MAC when running in MAC mode.*
6. Run the DPDK application:

   ```
   $ dpdk-l2fwd -c 0x3 -n 1 -- -p 0x3 -T 0
   $ dpdk-l3fwd -c 0x2 -n 1 -- -p 0x3 --config="(0,0,1), (1,0,1)"
   $ dpdk-testpmd -c 0x3 -n 1 -- -i --portmask=0x3 --forward-mode=mac
                 testpmd> set promisc all off
                 testpmd> start
   ```

Note: *Currently, the following test cases are not supported on i.MX 943:*

- *All multi-promisc mode*
- *Get Link status of VF*
- *MAC address exact match filter*
- *Multi-queue*
- *Link status interrupts*
- *Jumbo frame*
- *DPDK IP fragmentation*
- *DPDK reassembly*

10.7 IEEE 802.1Qbv

10.7.1 Overview

- IEEE 802.1Qbv allows time-based scheduling of network traffic.
- The Gate Control List (GCL) is a fundamental component of the IEEE 802.1Qbv protocol.
- The GCL (Gate C Logic) entries decide the gate open/close timings.
- The queues are allowed to transmit only when the gate is open.
- The GCL operates in a cyclic manner, repeating its sequence of entries periodically.

10.7.2 Steps to test the QBV test case scenario for ENETC

To test the TSN-QBV scenario for ENETC, perform the following steps:

1. Configure multiple queues for VF0.
   Use the following commands to configure 5 queues for VF0.

   ```
   devlink dev param set pci/0002:00:10.0 name si_num_rings cmode driverinit
    value 0105
   devlink dev reload pci/0002:00:10.0
   devlink dev param show pci/0002:00:10.0 name si_num_rings
   ```

   Note: *Maximum queues for VF0 can be **5** because the minimum **2** is required for PF by using the `tc` command to configure gate scheduling and **1** is reserved for VF1.*

2. Bind VFs to the DPDK.

   ```
   echo 1 > /sys/bus/pci/devices/0002\:00\:10.0/sriov_numvfs
   dpdk-devbind.py -b uio_pci_generic 0002:00:12.0
   ip link set eth1 vf 0 trust on
   ```

3. Configure the QBV rule using the `tc` command.
   Use the following commands to configure the gate scheduling:

   ```
    tc qdisc replace dev eth1 parent root handle 200 taprio num_tc 2 map 0 1
    queues 1@0 1@1 base-time 10000 sched-entry S 1 100000 sched-entry S 0 50000
    sched-entry S 2 100000  sched-entry S 0 50000 sched-entry S 4 100000 sched-
   entry S 0 50000  sched-entry S 8 100000 sched-entry S 0 50000  sched-entry
    S 10 100000  sched-entry S 0 50000  sched-entry S 20 100000 sched-entry S 0
    50000  sched-entry S 40 100000 sched-entry S 0 50000 sched-entry S 80 100000
    flags 0x2
   ```

   - Each queue is configured to a particular TC.
   - In this command, gate is configured to open for a particular queue for 100 microsec and then all queue gates are closed for 50 microsec between every gate scheduling.
   - Configuring all gates closed for some time can be beneficial when analyzing the captured data packets.
   - The priority for each queue is configured by users in DPDK using the devargs argument.
   - Although `num_tc` and map define only two PF queues, the GCL includes entries for multiple queues, including VF queues (for example, VF0). This is intentional so that the GCL can be effective for DPDK applications.
   - In the `tc` command, queue IDs are referenced using the mask values (for example, S 1, S 2, etc.). These map directly to TX queues in DPDK when the GCL is applied. For example, S 1 corresponds to the txq0 in DPDK, S 2 maps to txq1, and so on.

   Note: *For more details on `tc qdisc` and its usage, see the `tc qdisc` main page: Run `man tc` on your terminal for details.*

4. Check if the rule is properly implemented.

   ```
   tc -d -s qdisc show dev eth1 root
   ```

5. Run the `test-pmd` command.

   ```
   ./dpdk-testpmd -c 0xf -n 1 -a 0002:00:12.0,enetc4_txq_prior="1|2|3|4|5" --
      -i --nb-ports=1 --nb-cores=3 --forward-mode=txonly --txq=3
      --txonly_vlan_multiq_enable --txpkts=1500
   ```

   - The priority for each queue is configured by users in DPDK using the devargs argument as shown in the command above.
   - The priorities for each queue are separated by '|' in devargs.

   In `tc qdisc` GCL: Queue IDs (S 1, S 2, etc.) are scheduled for specific time slices.
   In DPDK: These queue IDs map to transmit queues as follows:

   - S 1 -> txq0
   - S 2 -> txq1
   - S 4 -> txq2, and so on

6. Capture the packets on Spirent and analyze the Wireshark output.
   The output should be observed in the way the rule is configured using the `tc` command.
   According to the `tc` command above, the expected results should be:
   VLAN priority 0 packets are displayed for 100 microsec, and then a break of 50 microsec should be observed. Then again for 100 microsec VLAN priority 1 packets are displayed, and then a break of 50 microsec, and so on.

10.8 DPDK-FPR application

10.8.1 Overview

DPDK-FPR is a DPDK-based Fast Path Routing application supported on i.MX 95 and i.MX 93.

The DPDK-FPR application provides the following capabilities:

- Routing between DPDK interfaces using Longest Prefix Match (LPM).
- Supports both IPv4 and IPv6 traffic.
- Learns Routes and Neighbors from the Linux kernel using Netlink, with automatic updates managed by kernel.
- Forwards non-IPv4 and non-IPv6 packets to kernel.
- Provides IP forwarding and rate limiting to kernel traffic.
- Support 5 tuples ACL based packets drop.
- Provides core-based statistics.
- Includes a Command Line Interface (CLI) based on Unix sockets.
- Configurable through a configuration file.

When running the DPDK-FPR application with an Ethernet interface, an equivalent tap interface (named `dtap*`) is created in the kernel.

This tap interface allows the application to send traffic to the kernel in scenarios where DPDK relies on the kernel or dependencies like routing and neighbor table information.

The MAC address of the dtap interface matches that of the Ethernet interface used by DPDK. Configuration of the dtap interface is managed through a callback script provided by the user.

10.8.2 DPDK compilation

Perform the following steps:

1. See Section 10.3.2 for Compiling DPDK using `meson`.
2. To install the DPDK in a particular directory, export the following `env.` variable for compiling the DPDK.

   ```
   export DESTDIR=<path to install>
   ```

3. Compile and install the DPDK:

   ```
   meson <params>
   ninja -C <build> install
   ```

10.8.3 Application compilation

Execute the following commands:

```
export ARCH=aarch64
export PATH=$PATH:<toolchain path>
export CC=aarch64-linux-gnu-gcc
export AR=aarch64-linux-gnu-ar
export PKG_CONFIG_LIBDIR=$PKG_CONFIG_LIBDIR:<DESTDIR>/usr/local/lib/pkgconfig/
make
```

Note:

1. *Update `CC` according to the toolchain GCC binary name.*
2. *Modify the `libdpdk.pc` file for correct installation path. Perform the following steps:*
   a. *Edit the `libdpdk.pc` file at the location: `<DESTDIR>/usr/local/lib/pkgconfig/libdpdk.pc`.*
   b. *Update the first line `prefix` path to: `prefix=<DESTDIR>/usr/local`.*

10.8.4 Running the application

Perform the following steps to run `dpdk-fpr`.

Note: *For the i.MX 93 platform:*

- *Use the `/etc/dpdk-fpr/fpr_imx93.conf` file to run `dpdk-fpr` on this platform.*
- *ENET-FEC limitation: Enable promiscuous to resolve arp of IPv6 traffic.*

1. Prepare the forwarding setup for the DPDK application. See Section 10.5.1.
2. Update `fpr.conf` and up files according to the use case. See Section 10.8.6.
   Note: *All the configuration files are available in the `/etc/dpdk-fpr` folder in `rootfs`, and the `dpdk-fpr` binary is in the `/usr/bin` folder in `rootfs`.*
3. Run the `dpdk-fpr` application.

```
cd /etc/dpdk-fpr
./dpdk-fpr --configfile ./fpr.conf
```

4. Open a new shell.
5. Connect to FPR using `socat`:

```
- socat - UNIX-CONNECT:/tmp/fpr.sock
- help
```

6. Resolve the ARP and run traffic.

10.8.5 CLI commands help accessible through socat

Perform the following steps:

1. Show help:

```
help
```

2. Show the neighbour table (similar with the ARP table in the Linux kernel, only valid rules):

```
neigh ipv4/ipv6
```

3. Look up a route entry in the LPM table:

```
lpm_lkp <IP>
```

4. Show the LPM status (Added/Deleted routes):

```
lpm_stats ipv4/ipv6
```

5. Show the status:

```
 # Detailed cores stats
  stats
 # Detailed cores stats in JSON format
  stats -j
 # Display cores stats after every <sec>
  stats -c <sec>
```

6. Dump the full neighbour table:

```
dump_neigh ipv4 <filepath e.g. /home/root/temp.txt>
cat /home/root/temp.txt
```

7. Dump the current log levels and log type IDs:

```
dumploglevel  <filepath e.g. /home/root/level.txt>
cat /home/root/level.txt
```

8. Set the global log level:

```
# Value can be 0 to 8: 0 is no logs and 8 is debug level
loglevel <level>
```

9. Enable/Disable the log type levels:

```
logtype <type_id> <enable/disable>
For type_id: Use "dumploglevel" to see type id of a DPDK component.
For all log types, type_id is 255.
enable/disable: 0 for log disable and 1 for log enable.
Level of enabled log will be global log level which can be set by "loglevel"
```

10. IP rate limit:

```
rlimit <IP> <PPS>
0 PPS indicates rate limit is disabled.
```

Note: *FPR limits the rate to approximate PPS value.*

11. Add IPv4/IPv6 ACL rules to drop packets:

```
acl_add ipv4/ipv6 <file_path>
```

12. Show/Clear the port:

```
# Display port information
show port info <portid>
# Display port specific statistics
show port stats <portid>
# Display port specific extended statistics
show port xstats <portid>
```

Note:

- *Replace `show` with `clear` to clear the statistics.*
- *These features are available only if the underlying driver supports these features.*

10.8.6 Configuration file options

The configuration file options include the following:

- Options defined under the section `[EAL]`:

```
arg = <list of EAL arguments>
```

Note:
– *Refer to DPDK-EAL for DPDK official EAL document arguments.*
– *Pass the tap interfaces as `vdev` in the EAL argument list.*
– *The number of `vdev` must be equal to the actual number of the Ethernet devices.*

- Options defined under the section `[fpr]`:

```
    - callback-setup = <script>
Give shell script path to config kernel interfaces.
```

```
Script can have commands for IP address, routes and ARPs
   - promiscuous = <0/1>
1 to enable and 0 to disable promiscuous
   - multicast = <0/1>
1 to enable and 0 to disable multicast
   - kni_rate_limit = <PPS>
Rate limit the packets in PPS sent to the kernel interface. 0 value disables
 rate limit.
   - unixsock = <PATH>
Override cmdline unixsock path (default: "/tmp/fpr.sock)
   - rule_ipv4 = <file path>
File path for IPv4 ACL rules. Only packet drop action is supported. Use "/dev/
null" as value to     not to configure any ACL rule. Refer: L3FWD-ACL
   - rule_ipv6 = <file path>
File path for IPv6 ACL rules. Only packet drop action is supported. Use "/dev/
null" as value to   not to configure any ACL rule. Refer: L3FWD-ACL
   - max-pkt-len = <len>
Increase or decrease the maximum RX length.
max-pkt-len = MTU(Maximum Transmission Unit) + 14(src mac + dst mac) + 4(CRC)
MTU will also be driven from this value.
  - rate_limit = <file path>
Define rate limite for various IP range. FPR will limit the rate to approximate
 PPS value.
  - aclneon = 1
To enable Neon algorithm for ACL
```

- Options defined under the section `[port <portid>]`:
  Note: *`<portid>` is the ID of the port to be used.*

```
 - eal queues = <queue id> <lcore id>
Only single queue is supported so queue id must be 0
 - kni = <lcore id>
Configuration of equivalent kernel interface.
User will have to create the interface via EAL arguments.
```

Note: *To comment a line in the configuration file, put `;` before the line.*

10.8.7 Limitations

The DPDK-FPR application has the following limitations:

- The default gateway for routes is not supported.
- No IPv6 extension headers are supported.

10.9 MTCP

10.9.1 i.MX 95 setup

- Connect two i.MX 95 boards in back-to-back setup.
- See Section 10.5.2.1, Section 10.5.2.2, and Section 10.5.4.2.1 to prepare the i.MX 95 boards.

10.9.2 "client" Application

Application measures read and write throughput for given number of seconds.

1. Rename the sample files `sample_route.conf` and `sample_arp.conf` under `/usr/bin/mtcp` to `route.conf` and `arp.conf` respectively.
2. Copy `route.conf` and `arp.conf` to the `./config` directory.

3. Update the `route` and `arp` entries according to the IP addresses and MAC addresses of the interfaces.
   Note: *The port MAC address and IP address (if not given in the configuration file) are shown on the application initialization. Therefore, execute the application, get the IP addresses and MAC addresses, fill the `route` and `arp` configuration, and re-run the application.*
4. Update `client.conf` with the required details and correct port names to be used for traffic. The port names must be separated by commas (,).
   Note: *For i.MX 95, the PCI address format, for example, 0000.00.10.0, for the port is used instead of the port name.*
   For details, see Section 10.9.4.
5. On the first board, run the following command:

```
./client send <destination IP> <port> <time in seconds>
```

6. On the second board, run the following command:

```
./client wait <destination IP> <port> <time in seconds>
```

   Where,
   - `<port>`: Any TCP port number
   - `client send <destination IP>`: IP address of the neighbor
   - `client wait <destination IP>`: Destination IP address of the client interface

   Take two boards i.MX95_1 and i.MX95_2 as an example.
   - i.MX95_1 port (0000:00:12.0) IP address is 16.0.0.80 and MAC address is 00:04:9F:08:9A:AB.
   - i.MX95_2 port (0002:00:12.0) IP address is 16.0.0.90 and MAC address is 00:04:9F:08:DA:61.

   Update the arp entries in the `config/arp.conf` file:
   - i.MX95_1: 16.0.0.90/32 00:04:9F:08:DA:61
   - i.MX95_2: 16.0.0.80/32 00:04:9F:08:9A:AB

   Update the route entries in the `config/route.conf` file:
   - i.MX95_1: 16.0.0.90/0 0000:00:12.0
   - i.MX95_2: 16.0.0.80/0 0002:00:12.0

   Run the client application:
   - i.MX95_1: `./client send 16.0.0.90 1025 10`
   - i.MX95_2: `./client wait 16.0.0.90 1025 10`

   Note: *The application can run on only 1 core.*

10.9.3 Webserver

Epserver is the server application. `ewget` reads the file we write at the server side. It is similar to `wget`.

1. Rename the sample files `sample_route.conf` and `sample_arp.conf` under `/usr/bin/mtcp` to `route.conf` and `arp.conf` respectively.
2. Copy `route.conf` and `arp.conf` to the `./config` directory.
3. Update the `route` and `arp` entries according to the IP addresses and MAC addresses of the interfaces.
   Note: *The port MAC address and IP address (if not given in the configuration file) are shown on the application initialization. Therefore, execute the application, get the IP addresses and MAC addresses, fill the `route` and `arp` configuration, and re-run the application.*
4. Update `epserver.conf` and `epwget.conf` with the required details and correct port names to be used for traffic. The port names must be separated by commas (,).
   Note: *For i.MX 95, the PCI address format, for example, 0000.00.10.0, for the port is used instead of the port name.*
   For details, see Section 10.9.4.
5. Create a random txt file (for example, `a.txt`) on the server board in the `/home/www` directory. Create a directory `www` if it does not exist.

6. On the first board, run the following command:

```
./epserver -p /home/www -f epserver.conf -N 8
```

7. On the second board, run the following command:

```
./epwget <Server IP/file name> <number of requests> -N 1 -c 8000 -f
 epwget.conf -o <file name>
e.g. ./epwget 10.0.0.112/a.txt 10 -N 1 -c 8000 -f epwget.conf -o b.txt
```

   Where,

   - `-p` is the path of the server home directory where all the files are present.
   - `-f` is the configuration file name.
   - `-N` is the number of cores.
   - `-c` is the number of concurrent connections.
   - `-o` (optional) is the output file in which the content of `a.txt` receives.

   Note: *`epwget` can work on only 1 core.*

10.9.4 MTCP configuration

The following is the description for the `Port`, `eal_args`, and `stat_print` parameters:

- `Port`: List of the port names to be used for traffic. Port names must be separated by commas (,).
  Note: *For i.MX 95, the PCI address format, for example, 0000.00.10.0, for the port is used instead of the port name.*
  – Example for multiple-port configuration:
    To use ENETC-1G and ENETC-10G ports, the port configuration is like:

```
port = 0002:00:02.0,0002:00:12.0
```

  – Example for port configuration with IP addresses:
    To use ENETC-1G port with the IP address 192.168.1.1 and the netmask value 255.255.255.0, and ENETC-10G with the IP address 192.168.2.1 and the netmask value 255.255.255.0.

```
port = 0002:00:02.0@192.168.1.1/24,0002:00:12.0@192.168.2.1/24
```

  Note:

  – *The IP value must be immediate after the port name and (@) separated.*
  – *The `port` configuration parameter accepts user's given IP address for each port. If the IP address is not given for the port, the MTCP stack generates random IP addresses.*

- `eal_args`: List of DPDK EAL library configuration. See EAL for the detailed list of supported parameters.

```
eal_args = --log-level=8 -b
```

  The parameters must be separated by a space character.
  Note: *Do not use `eal_args` for core-mask. `num_cores` can be used for the number of cores.*

- `stat_print`: List of the port names to be used for statistics. Port names must be separated by space characters.
  – Example to enable statistics for a single DPDK port:
    To use only ENETC-1G, `stat_print` configuration is like:

```
stat_print = 0002:00:02.0
```

  – Example to enable statistics for multiple DPDK ports:
    To use ENETC-1G and ENETC-10G ports, `stat_print` configuration is like:

```
stat_print = 0002:00:02.0, 0002:00:12.0
```

Note: *There are other parameters like:*
- *Number of memory channels:* `num_mem_ch`
- *Receive buffer size:* `rcvbuf`
- *Send buffer size:* `sndbuf`
- *TCP timeout seconds:* `tcp_timeout`
- *TCP timewait seconds:* `tcp_timewait etc`

All parameters can be configured according to use case. Their description is mentioned in respective configuration files `client.conf`, `epserver.conf`, *and* `epwget.conf`.

10.10 Troubleshooting

- If DPDK example applications are not found in rootfs, ensure you are using the correct rootfs (`rootfs.wic.zst`), which integrates the DPDK.
- If applications are not initialized properly, ensure HugePages are added as mentioned in the sections above.
- If Ethernet port is not found, ensure that you are using the correct DTB.

11 eXpress Data Path (XDP)

eXpress Data Path (XDP) is a high-performance, programmable network data path in the Linux kernel. It allows packet processing at the earliest point in the kernel's networking stack, enabling ultra-low latency and high-throughput applications. XDP framework makes it possible to perform high-speed packet processing within BPF applications. To enable faster response to network operations, XDP runs a BPF program as soon as possible, immediately as packet is received by the network interface.

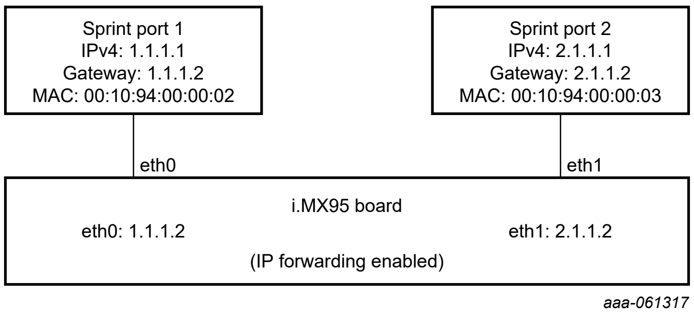

Figure 47. NAT flow setup and testing guide

Hardware requirements:

- i.MX 95, or any Linux-based board with two available Ethernet interfaces
- Two Spirent ports or Linux PCs

Test methods:

The `xdp_fp` program can be tested in the following three modes:

- Bridge Fastpath Mode: `xdp_fp_bridge`
  – Acts as a Layer 2 bridge for fast packet forwarding.
- IP Forwarding Fastpath Mode with Dynamic Rules: `xdp_fp with CMM`
  – Uses a Connection Management Module to dynamically manage routing.
- IP Forwarding Fastpath Mode with Static Rules: `xdp_fp -u`
  – Uses pre-defined static routing rules for deterministic packet forwarding.

11.1 IP Forwarding Fastpath Mode with Static User Rules

This mode enables IP forwarding and NAT using static rules defined by the user in an `input.txt` file. It does not require CMM and is ideal for controlled environments with fixed traffic patterns.

Run XDP with Static Rules:

1. Configure Ethernet interfaces.

   ```
   /etc/cmm/network_setup.sh eth0 1.1.1.2 eth1 2.1.1.2
   ```

2. Launch `xdp_fp` with Static Rules.

   ```
   /opt/xdp/xdp_fp -a eth0 eth1 -u
   ```

Note: *Ensure that `input.txt` is present in the current directory.*

Configure `input.txt`:

Edit the `input.txt` file located in `/opt/xdp/` and add the following rule:

```
nat_saddr=2.1.1.2
nat_daddr=8.8.8.8
nat_sport=1036
nat_dport=1024
mtu=1500
saddr=1.1.1.1
daddr=8.8.8.8
sport=1024
dport=1024
protocol=17
interface=eth1
```

Add a static route:

```
ip route add 8.8.8.0/24 via 2.1.1.1
```

Traffic generation and packet capture:

1. On Spirent Port 1:
   - Source IP: 1.1.1.1
   - Gateway: 1.1.1.2
   - Destination IP: 8.8.8.8
   - Destination MAC: MAC of eth0
   - UDP Source Port: 1024
   - UDP Destination Port: 1024
2. Resolve ARP and send traffic.
3. Observe the SNAT behaviour. Initial packets are processed by Linux (slow path). Once the connection is established, subsequent packets are handled by XDP (fast path).
4. Capture packets at **Spirent Port 2**. You should see:

   ```
   Time                    Source     Destination  Protocol  Length  Info
   2025-04-28 18:27:24.346281  2.1.1.2   8.8.8.8     UDP      128     1036 ->
    1024 Len=82
   2025-04-28 18:27:24.346281  2.1.1.2   8.8.8.8     UDP      128     1036 ->
    1024 Len=82
   ```

View statistics:

```
/opt/xdp/xdp_fp -s
```

De-attach program

```
/opt/xdp/xdp_fp -d eth0 eth1
```

11.2 IP Forwarding Fastpath Mode with Dynamic Rules (CMM)

This mode enables high-performance Layer 3 IP forwarding using dynamic connection management through the Connection Management Module (CMM). It supports both IPv4 and IPv6 traffic, VLANs, and NAT operations.

Run CMM and XDP:

1. Configure Ethernet interfaces and enable forwarding.
   ```
   /etc/cmm/load_cmm_modules.sh .
   /etc/cmm/network_setup.sh eth0 1.1.1.2 eth1 2.1.1.2
   ```
2. Launch `xdp_fp`.
   ```
   /opt/xdp/xdp_fp -a eth0 eth1
   ```
3. Start CMM.
   ```
   cmm
   ```

Traffic test steps:

1. On Spirent Port 1:
   - Source IP: 1.1.1.1
   - Gateway: 1.1.1.2
   - Destination IP: 2.1.1.1
   - Destination MAC: MAC of eth0
2. On Spirent Port 2:
   - Source IP: 2.1.1.1
   - Gateway: 2.1.1.2
   - Destination IP: 1.1.1.1
   - Destination MAC: MAC of eth1
3. Resolve ARP and send traffic.

Behaviour:

- Initial packets are processed by Linux (slow path) to establish connections.
- Once established, subsequent packets are handled by XDP (fast path).

Check statistics:

```
/opt/xdp/xdp_fp -s
```

Check CMM connections and routes:

```
telnet 127.0.0.1 2103
# Login: admin / admin
sh conn
```

```
sh route
quit
```

De-attach program:

```
/opt/xdp/xdp_fp -d eth0 eth1
```

Tested use cases:

1. **UDP & TCP IPv4 Traffic**
   Standard IP forwarding with dynamic connection tracking.
2. **SNAT and DNAT (via iptables)**
   Example:
   Use `iptables` to configure NAT rules for source and destination address translation.
   Quick setup for SNAT:

   ```
   /etc/cmm/load_cmm_modules.sh
   /etc/cmm/network_setup.sh eth0 1.1.1.2 eth1 2.1.1.2
   /opt/xdp/xdp_fp -a eth0 eth1
   cmm
   ip route add 9.1.1.0/24 via 2.1.1.1
   iptables -t nat -A POSTROUTING -p udp -j SNAT --to-source 2.1.1.2:10000
   ```

   Traffic test:
   - Spirent Port 1:
     - Source IP: 1.1.1.1, GW: 1.1.1.2, Destination IP: 9.1.1.1
     - Destination MAC: MAC of eth0
   - Spirent Port 2:
     - Capture to see NATed traffic.

   Resolve ARP for gateways 1.1.1.1 and 2.1.1.1 on the board.
3. **VLAN traffic forwarding**
   Quick setup:

   ```
   /etc/cmm/load_cmm_modules.sh
   /etc/cmm/network_setup.sh eth0 1.1.1.2 eth1 2.1.1.2
   ip link add link eth0 name eth0.10 type vlan id 10
   ip link add link eth1 name eth1.20 type vlan id 20
   ip link set eth0.10 up
   ip link set eth1.20 up
   ip addr add 1.1.10.2/24 dev eth0.10
   ip addr add 2.1.20.2/24 dev eth1.20
   /opt/xdp/xdp_fp -a eth0 eth1
   cmm
   ```

   Check VLANs:

   ```
   telnet 127.0.0.1 2103
   # Login: admin / admin
   sh vlan
   ```

   Traffic test:
   - Spirent Port 1: Source IP 1.1.10.1, GW 1.1.10.2, Destination IP 2.1.20.1, Destination MAC of eth0
   - Spirent Port 2: Source IP 2.1.20.1, GW 2.1.20.2, Destination IP 1.1.20.1, Destination MAC of eth1
4. **UDP IPv6 forwarding**
   Quick setup:

   ```
   /etc/cmm/load_cmm_modules.sh
   /etc/cmm/network_setup.sh eth0 2001:db8:1::2 eth1 2001:db8:2::2
   ```

```
/opt/xdp/xdp_fp -a eth0 eth1
cmm
```

Traffic test:

- Spirent Port 1: Source IP 2001:DB8:1::1, GW 2001:DB8:1::2, Destination IP 2001:DB8:2::1, Destination MAC of eth0
- Spirent Port 2: Source IP 2001:DB8:2::1, GW 2001:DB8:2::2, Destination IP 2001:DB8:1::1, Destination MAC of eth1

Resolve ARP:

```
ip -6 neigh
```

5. **UDP IPv4 multi-flow performance**
   Quick setup:

```
/etc/cmm/load_cmm_modules.sh
/etc/cmm/network_setup.sh --no-rxhash-off eth0 1.1.1.2 eth1 2.1.1.2
/opt/xdp/xdp_fp -a eth0 eth1
cmm
ip route add 2.1.20.0/24 via 2.1.1.1 dev eth1
ip route add 1.1.10.0/24 via 1.1.1.1 dev eth0
```

Traffic test:

- Spirent Port 1:
  - Source IP: 1.1.10.3, GW: 1.1.1.2, Destination IP: 2.1.20.3
  - IPv4 Modifier: Count=128; Step=0.0.0.1
  - Destination MAC: MAC of eth0
- Spirent Port 2:
  - Source IP: 2.1.20.3, GW: 2.1.1.2, Destination IP: 1.1.10.2
  - IPv4 Modifier: Count=128; Step=0.0.0.1
  - Destination MAC: MAC of eth1

Resolve ARP for gateways 1.1.1.1 and 2.1.1.1 on the board.

Check per-core statistics:

```
/opt/xdp/xdp_fp -s
```

11.3 Bridge Fastpath Mode

This section describes how to run the `xdp_fp` program in Layer 2 Bridge Fastpath mode on the i.MX platform.

Run XDP Fastpath in bridge mode

1. Configure Ethernet interfaces.
   Execute the following commands to set up the environment:

```
/etc/cmm/load_cmm_modules.sh .
/etc/cmm/network_setup.sh -p eth0 1.1.1.2 eth1 2.1.1.2
```

2. Launch `xdp_fp` in bridge mode.

```
/opt/xdp/xdp_fp -a eth0 eth1 -b
```

Note: *Change the interface names (for example, eth0, eth1) according to the hardware configuration.*

Traffic test steps:

1. On Spirent Port 1, configure a traffic stream with:
   - Source MAC: 00:10:94:00:00:02

  - Destination MAC: 00:10:94:00:00:03
2. On Spirent Port 2, configure a traffic stream with:
   - Source MAC: 00:10:94:00:00:03
   - Destination MAC: 00:10:94:00:00:02
3. Send traffic from both ports.

Behaviour:

- Initially, packets are classified as **unmatched** while the XDP bridge learns MAC addresses.
- Once learned, subsequent packets will be classified as **matched**.

View statistics:

To view runtime statistics:

```
/opt/xdp/xdp_fp -s
```

De-attach program:

To stop and detach the XDP bridge program:

```
/opt/xdp/xdp_fp -d eth0 eth1 -b
```

11.4 XDP-FP: NAT64

NAT64 implementation is written in the XDP program and not dependent on the kernel/CMM.

Compilation Note:

Compile `xdp_fp` with `-DNAT64_SIIT`. Pass it in both `CLANG_FLAGS` and `CFLAGS`.

12 Unit Tests

12.1 System

12.1.1 OProfile

12.1.1.1 Test name

- `autorun-oprofile.sh`

12.1.1.2 Location

`/unit_tests/OProfile/`

12.1.1.3 Functionality

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

12.1.1.4 Configuration

None.

12.1.1.5 Use case and expected output

```
./autorun-oprofile.sh
```

12.1.2 Owire

12.1.2.1 Test name

- `autorun-owire.sh`

12.1.2.2 Location

`/unit_tests/OWire/`

12.1.2.3 Functionality

Test the EEPROM functionality.

12.1.2.4 Configuration

None.

12.1.2.5 Use case and expected output

```
./autorun-owire.sh
```

12.1.3 Power Management

12.1.3.1 Test name

- `/unit_tests/Power_Management/suspend_random_auto.sh`
- `/unit_tests/Power_Management/suspend_quick_auto.sh`

12.1.3.2 Location

`/unit_tests/Power_Management/`

12.1.3.3 Functionality

Enables low power mode on and wakes up the different cores on all i.MX boards.

12.1.3.4 Configuration

None.

12.1.3.5 Use case and expected output

```
$ /unit_tests/Power_Management/suspend_random_auto.sh
or
$ /unit_tests/Power_Management/suspend_quick_auto.sh
```

Expected output on i.MX 7D SABRE SD board:

```
# /unit_tests/Power_Management/suspend_random_auto.sh
rtcwakeup.out: wakeup from "mem" using rtc0 at Wed Feb 22 22:55:29 2017
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
PM: suspend of devices complete after 632.862 msecs
PM: suspend devices took 0.640 seconds
PM: late suspend of devices complete after 1.258 msecs
PM: noirq suspend of devices complete after 1.198 msecs
Disabling non-boot CPUs ...
CPU1: shutdown
Turn off Mega/Fast mix in DSM
Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.832 msecs
imx-sdma 30bd0000.sdma: loaded firmware 4.2
PM: early resume of devices complete after 0.930 msecs
PM: resume of devices complete after 483.310 msecs
PM: resume devices took 0.480 seconds
Restarting tasks ... done.
==============================
suspend 0 times
==============================
wakeup 7 seconds, sleep 16 seconds
rtcwakeup.out: wakeup from "mem" using rtc0 at Wed Feb 22 22:55:42 2017
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
PM: suspend of devices complete after 630.328 msecs
PM: suspend devices took 0.640 seconds
PM: late suspend of devices complete after 1.252 msecs
PM: noirq suspend of devices complete after 1.203 msecs
Disabling non-boot CPUs ...
CPU1: shutdown
Turn off Mega/Fast mix in DSM
Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.777 msecs
imx-sdma 30bd0000.sdma: loaded firmware 4.2
PM: early resume of devices complete after 0.873 msecs
PM: resume of devices complete after 483.406 msecs
PM: resume devices took 0.480 seconds
Restarting tasks ... done.
==============================
suspend 1 times
==============================
wakeup 11 seconds, sleep 20 seconds
rtcwakeup.out: wakeup from "mem" using rtc0 at Wed Feb 22 22:56:10 2017
37PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
PM: suspend of devices complete after 651.761 msecs
PM: suspend devices took 0.660 seconds
PM: late suspend of devices complete after 1.245 msecs
PM: noirq suspend of devices complete after 1.193 msecs
```

```
Disabling non-boot CPUs ...
CPU1: shutdown
Turn off Mega/Fast mix in DSM
Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.728 msecs
imx-sdma 30bd0000.sdma: loaded firmware 4.2
PM: early resume of devices complete after 0.859 msecs
PM: resume of devices complete after 483.441 msecs
PM: resume devices took 0.480 seconds
Restarting tasks ... done.
==============================
suspend 2 times
==============================
wakeup 3 seconds, sleep 12 seconds
rtcwakeup.out: wakeup from "mem" using rtc0 at Wed Feb 22 22:56:34 2017
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
PM: suspend of devices complete after 641.321 msecs
PM: suspend devices took 0.650 seconds
PM: late suspend of devices complete after 1.258 msecs
PM: noirq suspend of devices complete after 1.195 msecs
Disabling non-boot CPUs ...
CPU1: shutdown
Turn off Mega/Fast mix in DSM
Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.730 msecs
imx-sdma 30bd0000.sdma: loaded firmware 4.2
PM: early resume of devices complete after 0.857 msecs
PM: resume of devices complete after 483.451 msecs
PM: resume devices took 0.480 seconds
Restarting tasks ... done.
==============================
suspend 3 times
==============================
wakeup 9 seconds, sleep 8 seconds
rtcwakeup.out: wakeup from "mem" using rtc0 at Wed Feb 22 22:56:56 2017
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
38Suspending console(s) (use no_console_suspend to debug)
PM: suspend of devices complete after 641.492 msecs
PM: suspend devices took 0.650 seconds
PM: late suspend of devices complete after 1.255 msecs
PM: noirq suspend of devices complete after 1.201 msecs
Disabling non-boot CPUs ...
CPU1: shutdown
Turn off Mega/Fast mix in DSM
Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.731 msecs
imx-sdma 30bd0000.sdma: loaded firmware 4.2
PM: early resume of devices complete after 0.861 msecs
PM: resume of devices complete after 483.476 msecs
PM: resume devices took 0.480 seconds
Restarting tasks ... done.
```

```
^c
```

```
# /unit_tests/Power_Management/suspend_quick_auto.sh
rtcwakeup.out: wakeup from "mem" using rtc0 at Wed Feb 22 23:01:16 2017
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
PM: suspend of devices complete after 632.891 msecs
PM: suspend devices took 0.640 seconds
PM: late suspend of devices complete after 1.254 msecs
PM: noirq suspend of devices complete after 1.200 msecs
Disabling non-boot CPUs ...
CPU1: shutdown
Turn off Mega/Fast mix in DSM
Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.734 msecs
imx-sdma 30bd0000.sdma: loaded firmware 4.2
PM: early resume of devices complete after 0.862 msecs
PM: resume of devices complete after 483.417 msecs
PM: resume devices took 0.480 seconds
Restarting tasks ... done.
================================
suspend 1 times
================================
rtcwakeup.out: wakeup from "mem" using rtc0 at Wed Feb 22 23:01:19 2017
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
PM: suspend of devices complete after 631.833 msecs
39PM: suspend devices took 0.640 seconds
PM: late suspend of devices complete after 1.253 msecs
PM: noirq suspend of devices complete after 1.242 msecs
Disabling non-boot CPUs ...
CPU1: shutdown
Turn off Mega/Fast mix in DSM
Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.729 msecs
imx-sdma 30bd0000.sdma: loaded firmware 4.2
PM: early resume of devices complete after 0.862 msecs
PM: resume of devices complete after 483.416 msecs
PM: resume devices took 0.480 seconds
Restarting tasks ... done.
================================
suspend 2 times
================================
rtcwakeup.out: wakeup from "mem" using rtc0 at Wed Feb 22 23:01:22 2017
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
PM: suspend of devices complete after 633.624 msecs
PM: suspend devices took 0.640 seconds
PM: late suspend of devices complete after 1.252 msecs
PM: noirq suspend of devices complete after 1.204 msecs
Disabling non-boot CPUs ...
```

```
CPU1: shutdown
Turn off Mega/Fast mix in DSM
Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.733 msecs
imx-sdma 30bd0000.sdma: loaded firmware 4.2
PM: early resume of devices complete after 0.853 msecs
PM: resume of devices complete after 483.450 msecs
PM: resume devices took 0.480 seconds
Restarting tasks ... done.
^c
```

12.1.4 Remote Processor Messaging

12.1.4.1 Test name

- `mxc_mcc_tty_test.out`

12.1.4.2 Location

`/unit_tests/Remote_Processor_Messaging`

12.1.4.3 Functionality

Test communication between Cortex-A and Cortex-M cores.

12.1.4.4 Configuration

To run the i.MX RPMsg test program, perform the following operations: Make sure that the proper Cortex-M4 processor RTOS and Linux images are used. The following are examples for the i.MX7Dual board:

- `rpmsg_pingpong_sdk_7dsdb.bin`: ping-pong test used on the i.MX 7Dual SDB board
- `rpmsg_str_echo_sdk_7dsdb.bin`: TTY string echo test used on the i.MX 7Dual SDB board
- `rpmsg_pingpong_sdk_7dval.bin`: ping-pong test used on the i.MX 7Dual 12x12 LPDDR3 Arm2 board
- `rpmsg_str_echo_sdk_7dval.bin`: TTY string echo test used on the i.MX 7Dual 12x12 LPDDR3 Arm2 board

Load the Cortex-M4 processor RTOS image, and kick it off in U-Boot. Load the Cortex-M4 processor RTOS image by the TFTP server or by the bootable SD card in U-Boot.

1. Load the Cortex-M4 processor RTOS image by the TFTP server:
   a. Boot into U-Boot and stop.
   b. Use the following command to TFTP the responding Cortex-M4 processor RTOS image and boot it.
   ```
   => dhcp 0x7f8000 10.192.242.53:rpmsg_pingpong_sdk_7dval.bin; bootaux
    0x7f8000
   ```
2. Load the Cortex-M4 processor RTOS image by the SD card:
   a. Create a bootable SD card by using MFGtools.
   b. Copy the Cortex-M4 processor RTOS files to the first partition formatted with the VFAT file system.
3. Change the default Cortex-M4 processor RTOS name of the U-Boot.
   ```
   => setenv m4image '<The name of the M4/RTOS image>';save
   ```

4. Set up a bootargs used by the Cortex-M4 processor.

```
=> setenv run_m4_tcm 'if run loadm4image; then cp.b ${loadaddr} 0x7f8000
 0x8000;
=> bootaux 0x7f8000; fi'; save
```

5. Modify the original bootcmd by adding `run run_m4_tcm`.

```
=> setenv bootcmd "run run_m4_tcm; <original contents of the bootcmd>"; save
```

Note: *`uart_from_osc` is required by i.MX 6SoloX when the Cortex-M4 processor RTOS image is running. Therefore, the `mmcargs` of U-Boot should be modified on i.MX 6SoloX.*

```
=> setenv mmcargs 'setenv bootargs console=${console},${baudrate} root=
${mmcroot}, uart_from_osc';save
```

6. Run the RPMsg test program.
   Make sure that `imx_rpmsg_pingpong.ko` and `imx_rpmsg_tty.ko` are built out. Use `insmod imx_rpmsg_pingpong.ko` or `insmod imx_rpmsg_tty.ko` to run the test program.
   Note: *NOTE Do not run different test programs at the same time.*

12.1.4.5 Use case and expected output

Run the following command and ensure that the RPMsg TTY receiving program is running at the backend when starting RPMsg TTY tests.

```
# ./mxc_mcc_tty_test.out /dev/ttyRPMSG30 115200 R 100 1000 &
Expected output:
mxc_mcc_tty_test.out:
$ insmod imx_rpmsg_tty.ko
$ imx_rpmsg_tty rpmsg0: new channel: 0x400 -> 0x1!
$ Install rpmsg tty driver!
$ echo deadbeaf > /dev/ttyRPMSG30
$ imx_rpmsg_tty rpmsg0: msg(<- src 0x1) deadbeaf len 8
```

12.1.5 Watchdog (WDOG)

12.1.5.1 Test name

- `autorun-wdog.sh`
- `wdt_driver_test.out`

12.1.5.2 Location

`/unit_tests/Watchdog/`

12.1.5.3 Functionality

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

12.1.5.4 Configuration

None.

12.1.5.5 Use case and expected output

```
Use case
./autorun-wdog.sh
or
./wdt_driver_test.out 1 2 0 &
Expected output
This should generate a reset after 3 seconds (a 1 second time-out and a 2 second
 sleep).
or
./wdt_driver_test.out 2 1 0
The system should keep running without being reset. This test requires the
 kernel to be executed
with the "jtag=on" on some platforms. Press "Ctrl+C" to terminate this test
 program.
```

12.2 Storage

12.2.1 Media Local Bus

12.2.1.1 Test name

- `mxc_mlb150_test`

12.2.1.2 Location

`/unit_tests/Media_Local_Bus/`

12.2.1.3 Functionality

MediaLB is an on-PCB or inter-chip communication bus specifically designed to standardize a common hardware interface and software API library.

12.2.1.4 Configuration

In menu configuration, enable the following module:

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

Test is only supported on i.MX 6SoloX, i.MX 6QuadPlus, i.MX 6Quad, i.MX 6DualLite.

12.2.1.5 Use case and expected output

```
./mxc_mlb150_test [-v] [-h] [-b] [-f fps] [-t casetype] [-q sync quadlets] [-p
 isoc
packet length]\n"
-v verbose
-h help
-b block io test
-f FPS, 256/512/1024/2048/3072/4096/6144
-t CASE, CASE can be 'sync', 'ctrl', 'async', 'isoc'
-q SYNC QUADLETS, quadlets per frame in sync mode, can be 1, 2, or 3
-p Packet length, package length in isoc mode, can be 188 or 196
```

12.2.2 MMC/SD/SDIO Host

12.2.2.1 Test name

- `autorun-mmc-blockrw.sh`
- `autorun-mmc-fdisk.sh`
- `autorun-mmc-fs.sh`
- `autorun-mmc-mkfs.sh`
- `autorun-mmc.sh`

12.2.2.2 Location

`/unit_tests/MMC_SD_SDIO/`

12.2.2.3 Functionality

The conjunction of MMC SD SDIO tests exercises the following instructions:

- MMC/SD read/write test.
- MMC/SD block read/write test.
- MMC/SD `fdisk` test.
- MMC/SD file system test.
- MMC/SD `mkfs` test.

12.2.2.4 Configuration

None.

12.2.2.5 Use case and expected output

All test return "Pass" if successful.

```
./autorun-mmc-blockrw.sh
./autorun-mmc-fdisk.sh
./autorun-mmc-fs.sh
./autorun-mmc-mkfs.sh
./autorun-mmc.sh
```

12.2.3 MMDC

12.2.3.1 Test name

- `mmdc2`

12.2.3.2 Location

`/unit_tests/MMDC/`

12.2.3.3 Functionality

MMDC profiling utility.

12.2.3.4 Configuration

The following parameters are used to customize the `mmcd2` test:

- `export MMDC_SLEEPTIME`: define profiling duration (500 ms by default).
- `export MMDC_LOOPCOUNT`: define profiling times (**1** by default, **-1** means infinite loop).
- `export MMDC_CUST_MADPCR1`: customize `madpcr1`.

12.2.3.5 Use case and expected output

The expected output shows the profiling results:

```
./mmdc2 [ARM:DSP1:DSP2:GPU2D:GPU2D1:GPU2D2:GPU3D:GPU3D2:GPUVG:VPU:M4:PXP:USB:SUM]
```

12.2.4 SATA

12.2.4.1 Test name

- `autorun-ata.sh`

12.2.4.2 Location

`/unit_tests/SATA/`

12.2.4.3 Functionality

This test writes data to the SATA drive connected to the SATA connector on the i.MX board. The data is then read back and compared to what was written.

12.2.4.4 Configuration

Module required: `pata_fsl.ko`.

Hardware required: SATA drive. Only i.MX 6Quad and 6QuadPlus have SATA support.

12.2.4.5 Use case and expected output

```
./autorun-ata.sh
Expected output
Test should return "HDD test passes" if successful.
```

12.3 Connectivity

12.3.1 Enhanced Configurable Serial Peripheral Interface (ECSPI)

12.3.1.1 Test name

- `mxc_spi_test1.out`

12.3.1.2 Location

`/unit_tests/ECSPI/`

12.3.1.3 Functionality

This test sends bytes of the last parameter to a specific SPI device. The maximum transfer bytes are 4096 bytes for bits per word less than 8 (including 8), 2048 bytes for bits per word between 9 and 16, and 1024 bytes for bits per word larger than 17 (including 17). SPI writes data received data from the user into TX FIFO and waits for the data in the RX FIFO. Once the data is ready in the RX FIFO, it is read and sent to the user.

12.3.1.4 Configuration

For the i.MX 6QuadPlus/6Quad/6Dual Auto boards, this requires the ECSPI device tree. This feature is disabled with the default device tree.

12.3.1.5 Use case and expected output

```
./mxc_spi_test1.out -D 0 -s 1000000 -b 8 E6E0
./mxc_spi_test1.out -D 1 -s 1000000 -b 8 -H -O -C E6E0E6E00001E6E00000
Usage:
./mxc_spi_test1.out [-D spi_no] [-s speed] [-b bits_per_word] [-H] [-O] [-C]
 $lt;value>
<spi_no> - CSPI Module number in [0, 1, 2]
<speed> - Max transfer speed
<bits_per_word> - bits per word
-H - Phase 1 operation of clock
-O - Active low polarity of clock
-C - Active high for chip select
<value> - Actual values to be sent
```

12.3.2 ETM

12.3.2.1 Test name

- `etm`

12.3.2.2 Location

`/unit_tests/ETM/`

12.3.2.3 Functionality

Embedded Trace Macrocell. The ETM is an optional debug component that enables reconstruction of program execution. The ETM is designed as a high-speed, low-power debug tool that only supports instruction trace. This ensures that area is minimized, and that gate count is reduced.

12.3.2.4 Configuration

None.

12.3.2.5 Use case and expected output

```
# ./etm -h
Usage: ./etm [options]
Options:
--etm-3.3 ETM v3.3 trace data
--etm-3.4-alt-branch ETM v3.4 trace data with alternative branch encoding
```

```
--pft-1.1 PFT v1.1 trace data
--cycle-accurate Cycle-accurate tracing was enabled (Default 1)
--contextid-bytes Number of Context ID bytes (Default 4)
--formatter Enable Formatter Unpacking
--sourceid-match Enable Source ID from formatter. Also enables formatter
--print-long-waits Highlight long waits
--print-input Print input data
--print-config Print configuration data
--help Print usage information
```

12.3.3 Inter-IC (I2C)

12.3.3.1 Test name

- `mxc_i2c_slave_test.out`

12.3.3.2 Location

`/unit_tests/I2C/`

12.3.3.3 Functionality

None.

12.3.3.4 Configuration

None.

12.3.3.5 Use case and expected output

None.

12.3.4 Keyboard

12.3.4.1 Test name

- `autorun-keypad.sh`
- `mxc_keyb_test.sh`

12.3.4.2 Location

`/unit_tests/Keyboard/`

12.3.4.3 Functionality

Tests keyboard input through the USB.

12.3.4.4 Configuration

Connect Keyboard to the USB OTG port.

12.3.4.5 Use case and expected output

```
./autorun-keypad.sh
Outputs:
Print "PASS" status
./mxc_keyb_test.sh
Output:
An event will occur when a key is pressed
```

12.3.5 Low Power Universal Asynchronous Receiver/Transmitter (LPUART)

12.3.5.1 Test name

- `autorun-mxc_uart.sh`
- `mxc_uart_stress_test.out`
- `mxc_uart_test.out`
- `mxc_uart_xmit_test.out`

12.3.5.2 Location

`/unit_tests/UART/`

12.3.5.3 Functionality

These tests excercise the low-level UART driver, which is responsible for supplying information, such as the UART port information and a set of control functions to the core UART driver.

12.3.5.4 Configuration

None.

12.3.5.5 Use case and expected output

```
./autorun-mxc_uart.sh
./mxc_uart_stress_test.out /dev/ttymxc#
./mxc_uart_test.out /dev/ttymxc#
./mxc_uart_xmit_test.out /dev/ttymxc#
```

12.3.6 USB

12.3.6.1 Test name

- `autorun-usb-gadget.sh`
- `autorun-usb-host.sh`

12.3.6.2 Location

`/unit_tests/USB/`

12.3.6.3 Functionality

This tests excerise the universal serial bus (USB) driver, which implements a standard Linux driver interface to the CHIPIDEA 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.

12.3.6.4 Configuration

Modules required:

- `/lib/modules/$(kernel_version)/kernel/drivers/usb/gadget/g_ether.ko`
- `/lib/modules/$(kernel_version)/kernel/drivers/usb/gadget/arcotg_udc.ko`
- `/lib/modules/$(kernel_version)/kernel/drivers/usb/host/ehci-hcd.ko`

12.3.6.5 Use case and expected output

```
./autorun-usb-gadget.sh
or
./autorun-usb-host.sh
```

12.4 Graphics

12.4.1 Graphics Processing Unit (GPU)

12.4.1.1 Test name

- `gpu.sh`
- `gpuinfo.sh`

12.4.1.2 Location

`/unit_tests/GPU`

12.4.1.3 Functionality

GPU function test:

- `tutorial3`: Test the OpenGL ES 1.1 basic function.
- `tutorial4_es20`: Test the OpenGL ES 2.0 basic function.
- : Test the OpenVG 1.1 basic function.
- : Test the Raster 2D and LibVivanteDK API.

12.4.1.4 Configuration

For `gpu.sh` and `gpuinfo.sh` to work, add the following line to the target board `defconfig` file:

```
CONFIG_MXC_GPU_VIV=y
```

Hardware required: LVDS Display Panel and i.MX SoC with a GPU.

12.4.1.5 Use case and expected output

```
./gpu.sh
```

Expected output frames are drawn properly on the screen:

- `tutorial3`: A cube with texture rotating in the middle of the screen.
- `tutorial4_es20`: Draws a glass sphere inside a big sphere (enviroment mapping). The glass sphere shows both reflection and refraction effects.
- `tiger`: A tiger spinning on the screen.
- `tvui`: Draws several movie clips and a TV control panel.

Example output is:

```
# ./gpu.sh
---- Running < gpu.sh > test ----
/unit_tests/GPU /unit_tests/GPU
Rendered 100 frames in 624 milliseconds: 160.26 fps
id=43, a,b,g,r=0,8,8,8, d,s=16,0, AA=0,openvgbit=71
frames:100 -- fps:58.997051
press ESC to escape...
./gpu.sh: line 28: cd: /opt/viv_samples/hal/: No such file or directory
/unit_tests/GPU
---- Test < gpu.sh > ended ----
```

```
./gpuinfo.sh
```

Information about the GPU is shown on the console.

```
# ./gpuinfo.sh
----  Running < gpuinfo.sh > test  ----
GPU Info
gpu : 0
model : 2000
revision : 5108
product : 0
eco : 0
gpu : 8
model : 320
revision : 5007
product : 0
eco : 0
gpu : 9
model : 355
revision : 1215
product : 0
eco : 0
VIDEO MEMORY:
gcvPOOL_SYSTEM:
Free : 134217728 B
Used : 0 B
Total : 134217728 B
gcvPOOL_CONTIGUOUS:
Used : 0 B
gcvPOOL_VIRTUAL:
Used : 0 B
NON PAGED MEMORY:
```

```
Used : 0 B
Paged memory Info
lowMem: 0 bytes
highMem: 0 bytes
CMA memory info
cma: 138485760 bytes
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
Idle percentage:0.000.000.000.000.000.00%
>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>
---- Test < gpuinfo.sh > ended ----
```

12.5 Video

12.5.1 Display

12.5.1.1 Test name

- `autorun-fb.sh`
- `mxc_tve_test.sh`
- `mxc_fb_test.out`
- `mxc_epdc_fb_test.out`
- `mxc_epdc_v2_fb_test.out`
- `mxc_spdc_fb_test.out`
- `mxc_fb_vsync_test.out`

12.5.1.2 Location

`/unit_tests/Display/`

12.5.1.3 Functionality

The tests under the display directory test some of the display options that are available with the i.MX family of boards. Some of the devices that can be tested include LVDS, HDMI, and EPDC panels.

Specifically, the `mxc_fb_test.out` tests the following features:

- Basic FB operation
- Set background/foreground to 16 bpp FB
- Global Alpha blending
- Color key test
- Frame buffer panning
- Gamma test

Additionally, the EPDC tests of `mxc_epdc_fb_test.out` and `mxc_epdc_v2_fb_test.out` test the following features:

- Basic updates
- Rotation updates
- Grayscale frame buffer updates
- Auto-waveform selection updates
- Panning updates
- Overlay updates
- Auto-updates

- Animation mode updates
- Fast updates
- Partial to full update transitions
- Test Pixel Shifting effect
- Colormap updates
- Collision test mode
- Stress test
- RGB565, Y8 frame buffer format
- 0, 90, 180, 270 degree frame buffer rotation
- Frame buffer panning
- Use of the alternate frame buffer
- Auto-waveform mode selection
- Automatic update mode
- The force-monochrome update feature and animation mode updates
- Support for using a grayscale colormap
- Snapshot, Queue, and Queue, and Merge update schemes
- EPDC Collision Test mode

12.5.1.4 Configuration

To run some tests, changes to the `defconfig` file for the target board are required. These changes will add functionality in which the following tests depend on.

For `autorun-fb.sh`, `mxc_fb_test.out`, and `mxc_fb_vsync_test.out`, add the following to the target board `defconfig` file:

```
CONFIG_FB=y
CONFIG_FB_MXC=y
CONFIG_FB_MXC_EDID=y
CONFIG_FB_MXC_SYNC_PANEL=y
CONFIG_FB_MXC_LDB=y
CONFIG_FB_MXC_HDMI=y
```

For `mxc_epdc_fb_test.out` and `mxc_epdc_v2_fb_test.out`, add the following to the target board `defconfig` file:

```
CONFIG_FB=y
CONFIG_FB_MXC=y
CONFIG_FB_MXC_EINK_PANEL=y
CONFIG_MFD_MAX17135=y
CONFIG_REGULATOR_MAX17135=y
CONFIG_MXC_PXP=y
CONFIG_DMA_ENGINE=y
```

12.5.1.5 Use case and expected output

```
# ./autorun-fb.sh
```

Expected output is:

```
---- Running < autorun-fb.sh > test ----
Checking for devnode: /dev/fb0
```

```
autorun-fb.sh: PASS devnode found: /dev/fb0
FB Blank test
Screen should be off
FB Color test
Setting FB to 16-bpp
Setting FB to 24-bpp
Setting FB to 32-bpp
FB panning test
autorun-fb.sh: Exiting PASS
Exiting PASS.
```

```
# ./mxc_tve_test.sh
```

Expected output is:

```
---- Running < mxc_tve_test.sh > test ----
Setting TV to NTSC mode
/unit_tests/Display/mxc_tve_test.sh: line 9: echo: write error: Invalid argument
/unit_tests/Display/mxc_tve_test.sh: line 11: /unit_tests/mxc_v4l2_output.out:
 No such
file or directory
Blank the display
Unblank the display
Setting TV to PAL mode
/unit_tests/Display/mxc_tve_test.sh: line 22: echo: write error: Invalid
 argument
/unit_tests/Display/mxc_tve_test.sh: line 23: /unit_tests/mxc_v4l2_output.out:
 No such
file or directory
Blank the display
Unblank the display
```

```
# ./mxc_fb_test.out
```

Expected output is shown below. The test should pass without any failure messages, and the display on the panel should be correct. For each test, a sequence of updates should be reflected on the screen. For almost all tests, the text printed on the debug console describes the image that should be observed on the screen. For i.MX 6Quad, fb0 and fb1 are used for tests. fb0 is the background frame buffer, and fb1 is the foreground overlay frame buffer.

```
Opened fb: /dev/fb0 (DISP4 BG - DI1)
DISP4 BG - DI1: screen info: 1024x768 (virtual: 1024x1536) @ 32-bpp
Opened fb: /dev/fb1 (DISP4 FG)
DISP4 FG: screen info: 240x320 (virtual: 240x960) @ 16-bpp
@DISP4 BG - DI1: Set colorspace to 16-bpp
@DISP4 FG: Set colorspace to 16-bpp
Prepared DISP4 BG - DI1 (black) and DISP4 FG (white). Verify the screen and
 press any
key to continue!
@DISP4 BG - DI1: Succesfully changed screen to 1024x768 (virtual: 1024x768) @16-
bpp
@DISP4 FG: Succesfully changed screen to 240x320 (virtual: 240x320) @16-bpp
Testing global alpha blending...
Fill the FG in black (screen is 240x320 @ 16-bpp)
Fill the BG in white (screen is 1024x768 @ 16-bpp)
Alpha is 0, FG is opaque
Alpha is 255, BG is opaque
```

```
Color key enabled
Color key disabled
Global alpha disabled
Pan test start.
@DISP4 FG: Set the colorspace to 16-bpp
Pan test done.
@DISP4 BG - DI1: Set colorspace to 16-bpp
Pan test start.
@DISP4 BG - DI1: Set the colorspace to 16-bpp
Pan test done.
Gamma test start.
Gamma 0.800000
Gamma 1.000000
Gamma 1.500000
Gamma 2.200000
Gamma 2.400000
Gamma test end.
Test bpp start
@DISP4 BG - DI1: Set colorspace to 32-bpp
@DISP4 BG - DI1: Fill the screen in red
@DISP4 BG - DI1: Set colorspace to 24-bpp
@DISP4 BG - DI1: Fill the screen in blue
@DISP4 BG - DI1: Set colorspace to 16-bpp
@DISP4 BG - DI1: Fill the screen in green
Test bpp end
```

```
# ./mxc_epdc_fb_test.out [-h] [-a] [-n]
EPDC framebuffer driver test program.
Usage: mxc_epdc_fb_test [-h] [-a] [-p delay] [-u s/q/m] [-n <expression>]
-h Print this message
-a Enabled animation waveforms for fast updates (tests 8-9)
-p Provide a power down delay (in ms) for the EPDC driver
0 - Immediate (default)
-1 - Never
<ms> - After <ms> milliseconds
-u Select an update scheme
s - Snapshot update scheme
q - Queue update scheme
m - Queue and merge update scheme (default)
-n Execute the tests specified in expression
Expression is a set of comma-separated numeric ranges
If not specified, all tests except Stress are run
Example:
./mxc_epdc_fb_test.out -n 1-3,5,7
EPDC tests:
1 - Basic Updates
2 - Rotation Updates
3 - Grayscale Framebuffer Updates
4 - Auto-waveform Selection Updates
5 - Panning Updates
6 - Overlay Updates
7 - Auto-Updates
8 - Animation Mode Updates
9 - Fast Updates
10 - Partial to Full Update Transitions
11 - Test Pixel Shifting Effect
12 - Colormap Updates
13 - Collision Test Mode
14 - Stress Test
```

```
15 - Dithering Y8->Y1 Test
16 - Dithering Y8->Y4 Test
17 - Hardware Dithering Test
18 - Advanced Algorithm Test
```

The full set of tests should pass without any failure messages. For each test, a sequence of updates should be reflected on the screen. For almost all tests, the text printed on the debug console describes the image that should be observed on the screen.

`mxc_epdc_v2_fb_test.out`: The full set of tests should pass without any failure messages. For each test, a sequence of updates should be reflected on the screen. For almost all tests, the text printed on the debug console describes the image that should be observed on the screen.

```
# ./mxc_spdc_fb_test.out
---- Running < ./mxc_spdc_fb_test.out > test ----
Unable to open /dev/fb5
```

```
# ./mxc_fb_vsync_test.out
Usage:
/unit_tests/Display# ./mxc_fb_vsync_test.out <fb #> <count>
<fb #> the framebuffer number
<count> the frames to be rendered
Example:
/unit_tests/Display# echo 0 > /sys/class/graphics/fb0/blank
/unit_tests/Display# ./mxc_fb_vsync_test.out 0 100
```

Expected output is the following when using 100 for the `<count>` argument.

```
total time for 100 frames = 1655674 us = 60 fps
```

12.5.2 High-Definition Multimedia Interface (HDMI) and Display Port (DP) overview

12.5.2.1 Test name

- `mxc_cec_test.out`

12.5.2.2 Location

`/unit_tests/HDMI/`

12.5.2.3 Functionality

Verify the HDMI CEC function and send the power-off command to HDMI sink.

12.5.2.4 Configuration

For `mxc_cec_test.out` to work, add the following line to the target board `defconfig` file:

```
CONFIG_MXC_HDMI_CEC=y
```

The hardware should support HDMI, and TV should support HDMI CEC.

12.5.2.5 Use case and expected output

```
./mxc_cec_test.out
```

12.5.3 Video Processing Unit (VPU)

12.5.3.1 Test for i.MX 6

- `autorun-vpu.sh`
- `mxc_vpu_test.out`

12.5.3.1.1 Location

`/unit_tests/VPU/`

12.5.3.1.2 Functionality

The VPU test exercises the following options on the Video Processing Unit (VPU):

- Decode one stream and display on the LCD.
- Decode a stream and save it to a file.
- Decode a stream using a config file.
- Encode a YUV stream and save it to a file.
- Encode an image from the camera and decode it to display on the LCD.
- Decode multiple streams with different formats simultaneously.
- Decode and encode simultaneously.
- Output to TV out.
- Test VPU with VDI (HW deinterlace via IPU).

12.5.3.1.3 Configuration

This tests require `libvpu.so` under `/usr/lib/` and LCD display. This test requires the i.MX 6QuadPlus/6Quad/6Dual SoC.

12.5.3.1.4 Use case and expected output

```
./autorun-vpu.sh
Decode one stream and display on the LCD.
To test MPEG-4 decode and display to screen:
./mxc_vpu_test.out -D "-i /usr/vectors/file.m4v -f 0"
To test H.263 decode and display to screen:
./mxc_vpu_test.out -D "-i /usr/vectors/file.263 -f 1"
To test H.264 decode and display to screen:
./mxc_vpu_test.out -D "-i /usr/vectors/file.264 -f 2"
You can get the mp4 test file from the imx-test.git server.
It is located under test/mxc_vpu_test/configs/akiyo.mp4.
Decode a stream and save to a file.
To test MPEG-4 decode and save to file:
./mxc_vpu_test.out -D "-i /usr/vectors/file.m4v -f 0 -o out.yuv"
To test H.263 decode and save to file:
./mxc_vpu_test.out -D "-i /usr/vectors/file.263 -f 1 -o out.yuv"
To test H.264 decode and save to file:
./mxc_vpu_test.out -D "-i /usr/vectors/file.264 -f 2 -o out.yuv"
```

```
Decode a stream using a config file.
Change options in config file, e.g, config_dec. Input correct input filename,
 output filename, format,
./mxc_vpu_test.out -C config_dec
Encode a YUV stream and save to a file.
To test MPEG-4 encode and save to a file:
./mxc_vpu_test.out -E "-i file.yuv -w 240 -h 320 -f 0 -o file.mpeg4"
To test H.263 encode and save to a file:
./mxc_vpu_test.out -E "-i file.yuv -w 240 -h 320 -f 1 -o file.263"
To test H.264 encode and save to a file:
./mxc_vpu_test.out -E "-i file.yuv -w 240 -h 320 -f 2 -o file.264"
Encode an image from the camera and decode it to display on the LCD.
To encode an MPEG4 image from the camera and display on the LCD: that
./mxc_vpu_test.out -L "-f 0 -w 1280 -h 720"
To encode an H263 image from the camera and display on the LCD:
./mxc_vpu_test.out -L "-f 1 -w 1280 -h 720"
To encode an H264 image from the camera and display on the LCD:
./mxc_vpu_test.out -L "-f 2 -w 1280 -h 720"
Decode multiple streams with different formats simultaneously.
Decoder one H264 and one MPEG4 streams.
./mxc_vpu_test.out -D "-i/vectors/file.264 -f 2" -D "-i ./akiyo.mp4 -f 0 -o
 akiyo.yuv"
Decode and encode simultaneously.
Encode one MPEG-4 stream and decode one H.264 stream simultaneously.
./mxc_vpu_test.out -E "-w 176 -h 144 -f 0 -o enc.m4v" -D "-i/vectors/file.264 -f
Test VPU with TV out.
Decoder one stream as normal VPU test. For example, H264 video stream:
./mxc_vpu_test.out -D "-i filename -f 2"
Test VPU with VDI (HW deinterlace via IPU).
Select one stream with top and bottom fields are interlaced.
av_stress2_dsmcc4m_1_C1_V11_A6.mp4_track1.h264
To decode the stream and display on LCD:
./mxc_vpu_test.out -D "-i av_stress2_dsmcc4m_1_C1_V11_A6.mp4_track1.h264 -f2"
To decode the stream and display on LCD using high motion stream video De
 Interlacing algorithm:
./mxc_vpu_test.out -D "-i av_stress2_dsmcc4m_1_C1_V11_A6.mp4_track1.h264 -v h -
f2"
To decode the stream and display on LCD using low motion stream video De
 Interlacing algorithm:
./mxc_vpu_test.out -D "-i av_stress2_dsmcc4m_1_C1_V11_A6.mp4_track1.h264 -v l -
f2"
To decode the stream and display on LCD having input in NV12 pixel format:
./mxc_vpu_test.out -D "-i av_stress2_dsmcc4m_1_C1_V11_A6.mp4_track1.h264 -v
```

12.5.3.2 Test for i.MX 8M Quad

12.5.3.2.1 Location

`/unit_tests/VPU/hantro`

12.5.3.2.2 Functionality

The VPU test exercises the following option on the VPU:

- Decode a stream and save it to a file.

12.5.3.2.3 Use case and expected output

Example for decoding different codecs:

```
/unit_tests/VPU/hantro/g2dec -P -b -ibs -Oout.yuv test.hevc
/unit_tests/VPU/hantro/g2dec -P -b -iivf -Oout.yuv test.vp9
/unit_tests/VPU/hantro/hx170dec -P -Oout.yuv test.h264
/unit_tests/VPU/hantro/mx170dec -P -Oout.yuv test.mpeg4
/unit_tests/VPU/hantro/m2x170dec -P -Oout.yuv test.mpeg2
/unit_tests/VPU/hantro/vx170dec -P -Oout.yuv test.vc1
/unit_tests/VPU/hantro/vp8x170dec -P -Oout.yuv test.vp8
/unit_tests/VPU/hantro/vp6dec -P -Oout.yuv test.vp6
/unit_tests/VPU/hantro/rvx170dec -P -Oout.yuv test.rv
/unit_tests/VPU/hantro/jx170dec -P -Oout.yuv test.jpg
/unit_tests/VPU/hantro/ax170dec -P -Oout.yuv test.avs
```

12.5.3.3 Test for i.MX 8M Mini

12.5.3.3.1 Location

`/unit_tests/VPU/hantro`

12.5.3.3.2 Functionality

The VPU test exercises the following option on the VPU:

- Decode a stream and save it to a file.
- Encode a YUV stream and save it to a file.

12.5.3.3.3 Use case and expected output

Example for decoder:

```
/unit_tests/VPU/hantro/g2dec -P -b -ibs -Oout.yuv test.hevc
/unit_tests/VPU/hantro/g2dec -P -b -iivf -Oout.yuv test.vp9
/unit_tests/VPU/hantro/hx170dec -P -Oout.yuv test.h264
/unit_tests/VPU/hantro/vp8x170dec -P -Oout.yuv test.vp8
```

Example for encoder:

```
/unit_tests/VPU/hantro/h264_testenc -w176 -h144 -o temp.h264 -i test.yuv
/unit_tests/VPU/hantro/vp8_testenc -w176 -h144 -o temp.h264 -i test.yuv
```

12.5.3.4 Test for i.MX 8QuadXPlus, 8QuadMax, and i.MX 9

12.5.3.4.1 Location

`/unit_tests/V4L2_VPU/`

12.5.3.4.2 Functionality

The VPU test exercises the following option on the VPU:

- Decode a stream and save it to a file.

- Encode a YUV stream and save it to a file.

12.5.3.4.3 Use case and expected output

Example for decoder, which helps to list the `ifmt` value for different codecs:

```
/unit_tests/V4L2_VPU/mxc_v4l2_vpu_test.out parser --key 0 --name input.h264
 --fmt h264 decoder --key 1  --source 0  convert --key 2 --source 1 --fmt i420
 ofile --key 3 --source 2 --name output.yuv
```

Example for encoder (H.264 only):

```
/unit_tests/V4L2_VPU/mxc_v4l2_vpu_test.out ifile --key 0 --name
 input_1920_1080.yuv --fmt i420 --size 1920 1080 convert --key 1 --source 0 --
fmt nv12 encoder --key 2 --source 1 --size 1920 1080 --gop 60 --fmt h264 --qp
 25 --bitrate 5000000 --framerate 30 --profile 0 ofile --key 3 --source 2 --name
 output.h264
```

Execute `/unit_tests/V4L2_VPU/mxc_v4l2_vpu_test.out help` for more details.

12.5.4 JPEG Encoder and Decoder

12.5.4.1 Test name

- `encoder_test`
- `decoder_test`

12.5.4.2 Location

`/unit_tests/JPEG`

12.5.4.3 Functionality

The `encoder_test` receives a raw file in one of the supported formats as input and produces a JPEG file as output, with the same resolution and pixel format as the input, unless cropping is performed. The application puts the raw file in one V4L2 output buffer, enqueues it into the driver, and expects to dequeue the JPEG image in one capture buffer.

The `decoder_test` receives a JPEG file in one of the supported formats as input and produces a raw file as output, with the same resolution and pixel format as the input. The application puts the JPEG file in one V4L2 output buffer, enqueues it into the driver, and expects to dequeue the raw image in one capture buffer.

12.5.4.4 Configuration

No special configuration.

12.5.4.5 Use case and expected output

Run the applications to get the usage:

```
./decoder_test.out
```

Usage:

```
./decoder_test.out -d </dev/videoX> -f <FILENAME> -w <width> -h <height> -p
 <pixel_format> [-n <iterations>] [-x]
Supported pixel formats:
               yuv420: 2-planes, Y and UV-interleaved, same as NV12M
              yuv420s: 2-planes, Y and UV-interleaved, contiguous, same as NV12
            yuv420-12: 2-planes, Y and UV-interleaved, non-contiguous, 12-bit
 precision
           yuv420s-12: 2-planes, Y and UV-interleaved, contiguous, 12-bit
 precision
               yuv422: packed YUYV
            yuv422-12: packed YUYV, 12-bit precision
                rgb24: packed RGB (obsolete)
                bgr24: packed BGR
             bgr24-12: packed BGR, 12-bit precision
               yuv444: packed YUV
            yuv444-12: packed YUV, 12-bit precision
                 gray: Y8 Single Component
              gray-12: Y12 Single Component
                 argb: packed ARGB (obsolete)
                 abgr: packed ABGR
              abgr-12: packed ABGR, 12-bit precision
Optional arguments:
 -x: print a hexdump of the result
 -n: number of iterations for enqueue/dequeue loop
 -q: quality factor 1..100, for encoder only
 -W <crop width> -H <crop height> (optional, supported only for encoder)
```

Supported formats:

```
yuv420: 2-planes, Y and UV-interleaved, same as NV12
yuv422: packed YUYV
rgb24: packed RGB
yuv444: packed YUV
gray: Y8 or Y12 or Single Component
argb: packed ARGB
```

The input file has to be a JPEG file that matches the specified width, height, and pixel format. The output is a raw file called `outfile` in the current folder, with the same width, height, and pixel format as the input.

```
./encoder_test.out
```

Usage:

```
./encoder_test.out -d </dev/videoX> -f <FILENAME> -w <width> -h <height> -p
 <pixel_format> [-n <iterations>] [-x]
Supported pixel formats:
               yuv420: 2-planes, Y and UV-interleaved, same as NV12M
              yuv420s: 2-planes, Y and UV-interleaved, contiguous, same as NV12
            yuv420-12: 2-planes, Y and UV-interleaved, non-contiguous, 12-bit
 precision
           yuv420s-12: 2-planes, Y and UV-interleaved, contiguous, 12-bit
 precision
               yuv422: packed YUYV
            yuv422-12: packed YUYV, 12-bit precision
                rgb24: packed RGB (obsolete)
                bgr24: packed BGR
             bgr24-12: packed BGR, 12-bit precision
               yuv444: packed YUV
            yuv444-12: packed YUV, 12-bit precision
                 gray: Y8 Single Component
              gray-12: Y12 Single Component
                 argb: packed ARGB (obsolete)
                 abgr: packed ABGR
              abgr-12: packed ABGR, 12-bit precision
Optional arguments:
 -x: print a hexdump of the result
 -n: number of iterations for enqueue/dequeue loop
 -q: quality factor 1..100, for encoder only
 -W <crop width> -H <crop height> (optional, supported only for encoder)
```

Supported formats:

```
yuv420: 2-planes, Y and UV-interleaved, same as NV12
yuv422: packed YUYV
rgb24: packed RGB
yuv444: packed YUV
gray: Y8 or Y12 or Single Component
argb: packed ARGB
```

The input file has to be a raw file that matches the specified width, height, and pixel format. The output is a JPEG file called `outfile.jpeg` in the current folder, with the same width, height, and pixel format as the input, unless cropping is performed.

12.6 Audio

12.6.1 Advanced Linux Sound Architecture (ALSA) System on a Chip (ASoC) Sound

12.6.1.1 Test name

- `mxc_tuner_test.out`

12.6.1.2 Location

`/unit_tests/ALSA/`

12.6.1.3 Functionality

Test audio capabilities using ALSA.

12.6.1.4 Configuration

ALSA is supported on all i.MX for test aplay, arecord and speaker-test. To use this tuner test, it requires tuner hardware available only on the i.MX 6 Auto reference boards.

12.6.1.5 Use case and expected output

None.

12.6.2 Asynchronous Sample Rate Converter (ASRC) on i.MX 6/i.MX 8QuadMax/i.MX 8QuadXPlust

12.6.2.1 Test name

- `mxc_asrc_test.out`

12.6.2.2 Location

`/unit_tests/ASRC`

12.6.2.3 Functionality

Converts WAV to different sample rates.

12.6.2.4 Configuration

None.

12.6.2.5 Use case and expected output

```
#/unit_tests/ASRC/mxc_asrc_test.out -to 48000 /unit_tests/ASRC/audio8k16S.wav
audio48k16S.wav
---- Running < /unit_tests/ASRC/mxc_asrc_test.out >  test ----
Pair A requested
All tests passed with success
```

More usages for `mxc_asrc_test.out` can be obtained by the following command:

```
#/unit_tests/ASRC/mxc_asrc_test.out -h
---- Running  < /unit_tests/ASRC/mxc_asrc_test.out > test ----
**************************************************
* Test aplication for ASRC
* Options :
-to <output sample rate> <origin.wav$gt; <converted.wav>
<input clock source> <output clock source>
input clock source types are:
0 -- INCLK_NONE
1 -- INCLK_ESAI_RX
2 -- INCLK_SSI1_RX
3 -- INCLK_SSI2_RX
4 -- INCLK_SPDIF_RX
5 -- INCLK_MLB_CLK
6 -- INCLK_ESAI_TX
7 -- INCLK_SSI1_TX
8 -- INCLK_SSI2_TX
9 -- INCLK_SPDIF_TX
10 -- INCLK_ASRCK1_CLK
default option for output clock source is 0
output clock source types are:
0 -- OUTCLK_NONE
1 -- OUTCLK_ESAI_TX
2 -- OUTCLK_SSI1_TX
3 -- OUTCLK_SSI2_TX
4 -- OUTCLK_SPDIF_TX
5 -- OUTCLK_MLB_CLK
```

```
6 -- OUTCLK_ESAI_RX
7 -- OUTCLK_SSI1_RX
8 -- OUTCLK_SSI2_RX
9 -- OUTCLK_SPDIF_RX
10 -- OUTCLK_ASRCK1_CLK
default option for output clock source is 10
**************************************************
```

12.7 Security

12.7.1 Display Content Integrity Checker (DCIC)

12.7.1.1 Test name

- `mxc_dcic_test.out`

12.7.1.2 Location

`/unit_tests/DCIC/`

12.7.1.3 Functionality

The goal of the DCIC is to verify that a safety-critical information sent to a display is not corrupted.

12.7.1.4 Configuration

None.

12.7.1.5 Use case and expected output

```
# ./mxc_dcic_test.out -bw 18 -dev 1
```

Expected output for `mxc_dcic_test.out`:

```
Opened fb0
open /dev/dcic1
bpp=16, bus_width=18
Config ROI=1
Config ROI=3
Config ROI=5
ROI=0,crcRS=0x0, crcCS=0x0
ROI=1,crcRS=0x6cd6b18d, crcCS=0x6cd6b18d
ROI=2,crcRS=0x0, crcCS=0x0
ROI=3,crcRS=0xc9da7ae6, crcCS=0xc9da7ae6
ROI=4,crcRS=0x0, crcCS=0x0
ROI=5,crcRS=0xb5ba1453, crcCS=0xb5ba1453
ROI=6,crcRS=0x0, crcCS=0x0
ROI=7,crcRS=0x0, crcCS=0x0
ROI=8,crcRS=0x0, crcCS=0x0
ROI=9,crcRS=0x0, crcCS=0x0
ROI=10,crcRS=0x0, crcCS=0x0
ROI=11,crcRS=0x0, crcCS=0x0
ROI=12,crcRS=0x0, crcCS=0x0
ROI=13,crcRS=0x0, crcCS=0x0
ROI=14,crcRS=0x0, crcCS=0x0
```

```
ROI=15,crcRS=0x0, crcCS=0x0
All ROI CRC check success!
```

12.7.2 SIM

12.7.2.1 Test name

- `mxc_sim_test.out`

12.7.2.2 Location

`/unit_tests/SIM/`

12.7.2.3 Functionality

Basic testing of the SIM card interface.

12.7.2.4 Configuration

None.

12.7.2.5 Use case and expected output

```
/unit_tests/mxc_sim_test.out
Expected output
atr[0]= 0x3b atr[1]= 0x68 atr[2]= 0x0 atr[3]= 0x0 atr[4]= 0x0 atr[5]= 0x73
 atr[6]=
0xc8
atr[7]= 0x40 atr[8]= 0x0 atr[9]= 0x0 atr[10]= 0x90 atr[11]= 0x0
rx[0] = 0x6e rx[1] = 0x0
rx[0] = 0x6d rx[1] = 0x0
rx[0] = 0x6e rx[1] = 0x0
```

12.7.3 SNVS Real Time Clock (SRTC)

12.7.3.1 Test name

- `autorun-rtc.sh`
- `rtctest.out`
- `rtcwakeup.out`

12.7.3.2 Location

`/unit_tests/SRTC/`

12.7.3.3 Functionality

These tests check the Real Time Clock (RTC) module, which 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.

12.7.3.4 Configuration

For autorun-rtc.sh, rtctest.out and rtcwakeup.out to work add the following line to the target board defconfig file:

```
CONFIG_RTC_DRV_SNVS=y
```

12.7.3.5 Use case and expected output

```
./autorun-rtc.sh
or
./rtctest.out $lt;arg>
--full run all tests
--no-periodic don't run periodic interrupt tests
or
./rtcwakeup.out -d rtc0 -m mem -s 10
Expected output
autorun-rtc.sh: Exit with PASS results.
rtctest.out: The program ends with "Test complete" status.
rtcwakeup.out: System is wakeup after 10s.
```

Expected output for i.MX 7D SABRE-SD:

- `autorun-rtc.sh`:

```
autorun-rtc.sh
i.MX7D
Checking for devnode: /dev/rtc0
autorun-rtc.sh: PASS devnode found: /dev/rtc0
Running test case: ./rtctest.out --no-periodic
RTC Driver Test Example.
Counting 5 update (1/sec) interrupts from reading /dev/rtc0: 1 2 3 4 5
Again, from using select(2) on /dev/rtc0: 1 2 3 4 5
Current RTC date/time is 21-2-2017, 23:13:07.
Alarm time now set to 23:13:12.
Waiting 5 seconds for alarm... okay. Alarm rang.
*** Test complete ***
Typing "cat /proc/interrupts" will show 1 more events on IRQ rtc.
autorun-rtc.sh: PASS test case: ./rtctest.out --no-periodic
rtc irqs before running unit test: 303
rtc irqs after running unit test: 314
so rtc irqs during test was:
11
checking rtc interrupts PASS
autorun-rtc.sh: Exiting PASS
```

- `rtctest.out --full`:

```
./rtctest.out --full
RTC Driver Test Example.
Counting 5 update (1/sec) interrupts from reading /dev/rtc0: 1 2 3 4 5
Again, from using select(2) on /dev/rtc0: 1 2 3 4 5
Current RTC date/time is 21-2-2017, 23:14:48.
Alarm time now set to 23:14:53.
Waiting 5 seconds for alarm... okay. Alarm rang.
Periodic IRQ rate was 1Hz.
Counting 20 interrupts at:
2Hz:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
4Hz:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
```

```
8Hz:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
16Hz: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
32Hz: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
64Hz: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
*** Test complete ***
Typing "cat /proc/interrupts" will show 131 more events on IRQ rtc.
```

- `rtctest.out --no-periodic`:

```
/rtctest.out --no-periodic
RTC Driver Test Example.
Counting 5 update (1/sec) interrupts from reading /dev/rtc0: 1 2 3 4 5
Again, from using select(2) on /dev/rtc0: 1 2 3 4 5
Current RTC date/time is 21-2-2017, 23:16:24.
Alarm time now set to 23:16:29.
Waiting 5 seconds for alarm... okay. Alarm rang.
*** Test complete ***
Typing "cat /proc/interrupts" will show 1 more events on IRQ rtc.
```

- `rtcwakeup.out -d rtc0 -m mem -s 10`:

```
./rtcwakeup.out -d rtc0 -m mem -s 10
rtcwakeup.out: wakeup from "mem" using rtc0 at Wed Feb 22 23:17:29 2017
PM: Syncing filesystems ... done.
Freezing user space processes ... (elapsed 0.001 seconds) done.
Freezing remaining freezable tasks ... (elapsed 0.001 seconds) done.
Suspending console(s) (use no_console_suspend to debug)
PM: suspend of devices complete after 639.100 msecs
PM: suspend devices took 0.640 seconds
PM: late suspend of devices complete after 1.236 msecs
PM: noirq suspend of devices complete after 1.202 msecs
Disabling non-boot CPUs ...
CPU1: shutdown
Turn off Mega/Fast mix in DSM
Enabling non-boot CPUs ...
CPU1 is up
PM: noirq resume of devices complete after 0.756 msecs
imx-sdma 30bd0000.sdma: loaded firmware 4.2
PM: early resume of devices complete after 0.972 msecs
PM: resume of devices complete after 483.302 msecs
PM: resume devices took 0.480 seconds
Restarting tasks ... done.
```

13 Note About the Source Code in the Document

Example code shown in this document has the following copyright and BSD-3-Clause license:

Copyright 2025 NXP Redistribution and use in source and binary forms, with or without modification, are permitted provided that the following conditions are met:

1. Redistributions of source code must retain the above copyright notice, this list of conditions and the following disclaimer.
2. Redistributions in binary form must reproduce the above copyright notice, this list of conditions and the following disclaimer in the documentation and/or other materials provided with the distribution.
3. Neither the name of the copyright holder nor the names of its contributors may be used to endorse or promote products derived from this software without specific prior written permission.

THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS" AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES

14 Revision History

14.1 Revision History

This table provides the revision history.

Revision history

Document ID	Release date	Description
RM00293 v.LF6.12.34_2.1.0	25 September 2025	Upgraded to the 6.12.34 kernel, i.MX 943 A0 and i.MX 95 B0 support with Beta quality, and added the i.MX 8MP/91/93/95 FRDM boards support.
RM00293 v.LF6.12.20_2.0.0	26 June 2025	Upgraded to the 6.12.20 kernel, U-Boot v2025.04, TF-A 2.11, OP-TEE 4.6.0, Yocto 5.2 Walnascar, and added the i.MX 943 as Alpha quality.
RM00293 v.LF6.12.3_1.0.0	31 March 2025	Upgraded to the 6.12.3 kernel.
RM00293 v.LF6.6.52_2.2.0	16 December 2024	Upgraded to the 6.6.52 kernel.
RM00293 v.LF6.6.36_2.1.0	30 September 2024	Upgraded to the 6.6.36 kernel.
IMXLXRM_6.6.23_2.0.0	28 June 2024	Upgraded to the 6.6.23 kernel, U-Boot v2024.04, TF-A v2.10, OP-TEE 4.2.0, Yocto 5.0 Scarthgap, and added the i.MX 91 as Alpha quality, i.MX 95 as Beta quality.
IMXLXRM v.LF6.6.3_1.0.0	29 March 2024	Upgraded to the 6.6.3 kernel, removed the i.MX 91, and added the i.MX 95 as Alpha Quality.
IMXLXRM v.LF6.1.55_2.2.0	12/2023	Upgraded to the 6.1.55 kernel.
IMXLXRM v.LF6.1.36_2.1.0	10/2023	Updated the BCH schemes in Backward Compatibility.
IMXLXRM v.LF6.1.36_2.1.0	09/2023	Upgraded to the 6.1.36 kernel and added the i.MX 91P.
IMXLXRM v.LF6.1.22_2.0.0	06/2023	Upgraded to the 6.1.22 kernel.
IMXLXRM v.LF6.1.1_1.0.0	03/2023	Upgraded to the 6.1.1 kernel.
IMXLXRM v.LF5.15.71_2.2.0	12/2022	Upgraded to the 5.15.71 kernel.
IMXLXRM v.LF5.15.52_2.1.0	09/2022	Upgraded to the 5.15.52 kernel, and added the i.MX 93.
IMXLXRM v.LF5.15.32_2.0.0	06/2022	Upgraded to the 5.15.32 kernel, U-Boot 2022.04, and Kirkstone Yocto.
IMXLXRM v.LF5.15.5_1.0.0	03/2022	Upgraded to the 5.15.5 kernel, Honister Yocto, and Qt6.
IMXLXRM v.LF5.10.72_2.2.0	12/2021	Upgraded the kernel to 5.10.72 and updated the BSP.
IMXLXRM v.LF5.10.52_2.1.0	09/2021	Updated for i.MX 8ULP Alpha and the kernel upgraded to 5.10.52.
IMXLXRM v.LF5.10.35_2.0.0	06/2021	Upgraded to 5.10.35 kernel.

Revision history*...continued*

Document ID	Release date	Description
IMXLXRM v.LF5.10.9_1.0.0	03/2021	Upgraded to 5.10.9 kernel.
IMXLXRM v.L5.4.70_2.3.0	01/2021	Updated the command lines in Section "Running the Arm Cortex-M4 image".
IMXLXRM v.L5.4.70_2.3.0	12/2020	i.MX 5.4 consolidated GA for release i.MX boards including i.MX 8M Plus and i.MX 8DXL.
IMXLXRM v.L5.4.47_2.2.0	09/2020	i.MX 5.4 Beta2 release for i.MX 8M Plus, Beta for 8DXL, and consolidated GA for released i.MX boards.
IMXLXRM v.L5.4.24_2.1.0	06/2020	i.MX 5.4 Beta release for i.MX 8M Plus, Alpha2 for 8DXL, and consolidated GA for released i.MX boards.
IMXLXRM v.L5.4.3_2.0.0	04/2020	i.MX 5.4 Alpha release for i.MX 8M Plus and 8DXL EVK boards.
IMXLXRM v.LF5.4.3_1.0.0	03/2020	i.MX 5.4 Kernel and Yocto Project Upgrades.
IMXLXRM v.L4.19.35_1.1.0	10/2019	i.MX 4.19 Kernel and Yocto Project Upgrades.
IMXLXRM v.L4.19.35_1.0.0	07/2019	i.MX 4.19 Beta Kernel and Yocto Project Upgrades.
IMXLXRM v.L4.14.98_2.0.0_ga	04/2019	i.MX 4.14 Kernel upgrade and board updates.
IMXLXRM v.L4.14.78_1.0.0_ga	01/2019	i.MX 6, i.MX 7, i.MX 8 family GA release.
IMXLXRM v.L4.14.62_1.0.0_beta	11/2018	i.MX 4.14 Kernel Upgrade, Yocto Project Sumo upgrade.
IMXLXRM v.L4.9.123_2.3.0_8mm	09/2018	i.MX 8M Mini GA release.
IMXLXRM v.L4.9.88_2.2.0_8qxp-beta2	07/2018	i.MX 8QuadXPlus Beta2 release.
IMXLXRM v.L4.9.88_2.1.0_8mm-alpha	06/2018	i.MX 8M Mini Alpha release.
IMXLXRM v.L4.9.88_2.0.0-ga	05/2018	i.MX 7ULP and i.MX 8M Quad GA release.
IMXLXRM v.L4.9.51_imx8mq-ga	03/2018	Added i.MX 8M Quad GA.
IMXLXRM v.L4.9.51_8qm-beta2/8qxp-beta	02/2018	Added i.MX 8QuadMax Beta2 and i.MX 8QuadXPlus Beta.
IMXLXRM v.L4.9.51_imx8mq-beta	12/2017	Added i.MX 8M Quad.
IMXLXRM v.L4.9.51_imx8qm-beta1	12/2017	Added i.MX 8QuadMax.
IMXLXRM v.L4.9.51_imx8qxp-alpha	11/2017	Initial release.

Legal information

Contents