# MCIMX28x Linux

Reference Manual

Mas Elettronica Sas
Via Risorgimento 16/C
35030 Selvazzano Dentro (PD)
Italy.

# Contents

## Sommario

## *Revision History*

| Rev. | Document Code | Released | Written | Verified | Approved |
|---|---|---|---|---|---|
| 1.0 | | 15/07/2014 | N.Convertino | S.Mascetti | S.Mascetti |
| | | | | | |
| | | | | | |

# About This Book

The Linux Board Support Package (BSP) represents a porting of the Linux Operating System (OS) to the i.MX processors and its associated reference boards. The BSP supports many hardware features on the platforms and most of the Linux OS features that are not dependent on any specific hardware feature.

## Audience

This document is targeted to individuals who will port the i.MX Linux BSP to customer-specific products. The audience is expected to have a working knowledge of the Linux 2.6 kernel internals, driver models, and i.MX processors.

## Conventions

This document uses the following notational conventions:

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

## Definitions, Acronyms, and Abbreviations

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

**Definitions and Acronyms**

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

| | |
|---|---|
| CRC | Cyclic Redundancy Check—Bit error protection method for data communication |
| CSI | Camera Sensor Interface |
| DFS | Dynamic Frequency Scaling |
| DMA | Direct Memory Access—an independent block that can initiate memory-to-memory data transfers |
| DPM | Dynamic Power Management |
| DRAM | Dynamic Random Access Memory |
| DVFS | Dynamic Voltage Frequency Scaling |
| EMI | External Memory Interface—controls all IC external memory accesses (read/write/erase/program) from all the masters in the system |
| Endian | Refers to byte ordering of data in memory. Little endian means that the least significant byte of the data is stored in a lower address than the most significant byte. In big endian, the order of the bytes is reversed |
| EPIT | Enhanced Periodic Interrupt Timer—a 32-bit set and forget timer capable of providing precise interrupts at regular intervals with minimal processor intervention |
| FCS | Frame Checker Sequence |
| FIFO | First In First Out |
| FIPS | Federal Information Processing Standards—United States Government technical standards published by the National Institute of Standards and Technology (NIST). NIST develops FIPS when there are compelling Federal government requirements such as for security and interoperability but no acceptable industry standards |
| FIPS-140 | Security requirements for cryptographic modules—Federal Information Processing Standard 140-2(FIPS 140-2) is a standard that describes US Federal government requirements that IT products should meet for Sensitive, but Unclassified (SBU) use |
| Flash | A non-volatile storage device similar to EEPROM, where erasing can be done only in blocks or the entire chip. |
| Flash path | Path within ROM bootstrap pointing to an executable Flash application |
| Flush | Procedure to reach cache coherency. Refers to removing a data line from cache. This process includes cleaning the line, invalidating its VBR and resetting the tag valid indicator. The flush is triggered by a software command |
| GPIO | General Purpose Input/Output |
| hash | Hash values are produced to access secure data. A hash value (or simply hash), also called a message digest, is a number generated from a string of text. The hash is substantially smaller than the text itself, and is generated by a formula in such a way that it is extremely unlikely that some other text produces the same hash value. |
| I/O | Input/Output |
| ICE | In-Circuit Emulation |
| IP | Intellectual Property |
| IPU | Image Processing Unit —supports video and graphics processing functions and provides an interface to video/still image sensors and displays |
| IrDA | Infrared Data Association—a nonprofit organization whose goal is to develop globally adopted specifications for infrared wireless communication |
| ISR | Interrupt Service Routine |
| **Term** | **Definition** |
| JTAG | JTAG (IEEE Standard 1149.1) A standard specifying how to control and monitor the pins of compliant devices on a printed circuit board |
| Kill | Abort a memory access |
| KPP | KeyPad Port—16-bit peripheral used as a keypad matrix interface or as general purpose input/output (I/O) |
| line | Refers to a unit of information in the cache that is associated with a tag |

| | |
|---|---|
| LRU | Least Recently Used—a policy for line replacement in the cache |
| MMU | Memory Management Unit—a component responsible for memory protection and address translation |
| MPEG | Moving Picture Experts Group—an ISO committee that generates standards for digital video compression and audio. It is also the name of the algorithms used to compress moving pictures and video |
| MPEG standards | Several standards of compression for moving pictures and video:<br>•MPEG-1 is optimized for CD-ROM and is the basis for MP3<br>•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<br>•MPEG-4 is a standard for low-bandwidth video telephony and multimedia on the World-Wide Web |
| MQSPI | Multiple Queue Serial Peripheral Interface—used to perform serial programming operations necessary to configure radio subsystems and selected peripherals |
| MSHC | Memory Stick Host Controller |
| NAND Flash | Flash ROM technology—NAND Flash architecture is one of two flash technologies (the other being NOR) used in memory cards such as the Compact Flash cards. NAND is best suited to flash devices requiring high capacity data storage. NAND flash devices offer storage space up to 512-Mbyte and offers faster erase, write, and read capabilities over NOR architecture |
| NOR Flash | See NAND Flash |
| PCMCIA | Personal Computer Memory Card International Association—a multi-company organization that has developed a standard for small, credit card-sized devices, called PC Cards. There are three types of PCMCIA cards that have the same rectangular size (85.6 by 54 millimeters), but different widths |
| physical address | The address by which the memory in the system is physically accessed |
| PLL | Phase Locked Loop—an electronic circuit controlling an oscillator so that it maintains a constant phase angle ( a lock) on the frequency of an input, or reference, signal |
| RAM | Random Access Memory |
| RAM path | Path within ROM bootstrap leading to the downloading and the execution of a RAM application |
| RGB | The RGB color model is based on the additive model in which Red, Green, and Blue light are combined to create other colors. The abbreviation RGB comes from the three primary colors in additive light models |
| RGBA | RGBA color space stands for Red Green Blue Alpha. The alpha channel is the transparency channel, and is unique to this color space. RGBA, like RGB, is an additive color space, so the more of a color placed, the lighter the picture gets. PNG is the best known image format that uses the RGBA color space |
| RNGA | Random Number Generator Accelerator—a security hardware module that produces 32-bit pseudo random numbers as part of the security module |
| ROM | Read Only Memory |
| **Term** | **Definition** |
| ROM bootstrap | Internal boot code encompassing the main boot flow as well as exception vectors |
| RTIC | Real-Time Integrity Checker—a security hardware module |
| SCC | SeCurity Controller—a security hardware module |
| SDMA | Smart Direct Memory Access |
| SDRAM | Synchronous Dynamic Random Access Memory |
| SoC | System on a Chip |
| SPBA | Shared Peripheral Bus Arbiter—a three-to-one IP-Bus arbiter, with a resource-locking mechanism |
| SPI | Serial Peripheral Interface—a full-duplex synchronous serial interface for connecting low-/medium-bandwidth external devices using four wires. SPI devices communicate using a master/slave relationship over two data lines |

| | |
|---|---|
| | and two control lines: *Also see SS, SCLK, MISO, and MOSI* |
| SRAM | Static Random Access Memory |
| SSI | Synchronous-Serial Interface—standardized interface for serial data transfer |
| TBD | To Be Determined |
| UART | Universal Asynchronous Receiver/Transmitter—asynchronous serial communication to external devices |
| UID | Unique ID–a field in the processor and CSF identifying a device or group of devices |
| USB | Universal Serial Bus—an external bus standard that supports high speed data transfers. The USB 1.1 specification supports data transfer rates of up to 12 Mb/s and USB 2.0 has a maximum transfer rate of 480 Mbps. A single USB port can be used to connect up to 127 peripheral devices, such as mice, modems, and keyboards. USB also supports Plug-and-Play installation and hot plugging |
| USBOTG | USB On The Go—an extension of the USB 2.0 specification for connecting peripheral devices to each other. USBOTG devices, also known as dual-role peripherals, can act as limited hosts or peripherals themselves depending on how the cables are connected to the devices, and they also can connect to a host PC |
| word | A group of bits comprising 32-bits |

# Suggested Reading

The following documents contain information that supplements this guide

# Chapter 1 Introduction

The i.MX family Linux Board Support Package (BSP) supports the Linux Operating System (OS) on the following processor:

- i.MX28 Applications Processor

The purpose of this software package is to support Linux on the i.MX family of Integrated Circuits (ICs) and their associated platforms (EVK). It provides the necessary software to interface the standard open-source Linux kernel to the i.MX hardware. The goal is to enable Freescale customers to rapidly build products based on i.MX devices that use the Linux OS.

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

The BSP is not intended to be used for silicon verification. While it can play a role in this, the BSP functionality and the tests run on the BSP do not have sufficient coverage to replace traditional silicon verification test suites.

## 1.1 Software Base

The i.MX BSP is based on version 2.6.35.3 of the Linux kernel from the official Linux kernel web site (http://www.kernel.org). It is enhanced with the features provided by Freescale.

## 1.2 Features

Table 1-1 describes the features supported by the Linux BSP for specific platforms.

| Feature | Description | Chapter Source | Applicable Platform |
|---|---|---|---|
| **Machine Specific Layer** | | | |
| MSL | Machine Specific Layer (MSL) supports interrupts, Timer, Memory Map, GPIO/IOMUX, SPBA, SDMA.<br>•Interrupts (AITC/AVIC): The Linux kernel contains common ARM code for handling interrupts. The MSL contains platform-specific implementations of functions for interfacing the Linux kernel to the | Chapter 3, “Machine Specific Layer (MSL)” | All |

| | | | |
|---|---|---|---|
| | interrupt controller.<br>•Timer (GPT): The General Purpose Timer (GPT) is set up to generate an interrupt as programmed to provide OS ticks. Linux facilitates timer use through various functions for timing delays, measurement, events, alarms, high resolution timer features, and so on. Linux defines the MSL timer API required for the OS-tick timer and does not expose it beyond the kernel tick implementation.<br>•GPIO/EDIO/IOMUX: The GPIO and EDIO components in the MSL provide an abstraction layer between the various drivers and the configuration and utilization of the system, including GPIO, IOMUX, and external board I/O. The IO software module is board-specific, and resides in the MSL layer as a self-contained set of files. I/O configuration changes are centralized in the GPIO module so that changes are not required in the various drivers.<br>•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. | | |
| DMAC | Both AHB-to-APBH and AHB-to-APBX DMA support configurable DMA descript chain. | Chapter 4, “Direct Memory Access Controller (DMAC) API” | i.MX28 |

| **Power Management Drivers** | | | |
|---|---|---|---|
| Low-level PM Drivers | The low-level power management driver is responsible for implementing hardware-specific operations to meet power requirements and also to conserve power on the development platforms. Driver implementations are often different for different platforms. It is used by the DPM layer. | Chapter 8, “i.MX28 Static Power Management Driver” | i.MX28 |
| CPU Frequency Scaling | The CPU frequency scaling device driver allows the clock speed of the CPUs to be changed on the fly. | Chapter 7, “CPU Frequency Scaling (CPUFREQ) Driver” | i.MX28 |

| **Memory Drivers** | | | |
|---|---|---|---|
| NAND MTD | The NAND MTD driver interfaces with the integrated NAND controller. It can support various file systems, such as UBIFS, CRAMFS and JFFS2. The driver implementation supports the lowest level operations on the external NAND Flash chip, such | Chapter 14, “NAND Flash Driver | |

| | | | |
|---|---|---|---|
| | 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. | | |

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

| Bootloaders | | | |
|---|---|---|---|
| uBoot | uBoot is an open source boot loader. | See uBoot User guide | i.MX28 |

# Chapter 2 Architecture

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

## 2.1 Linux BSP Block Diagram

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

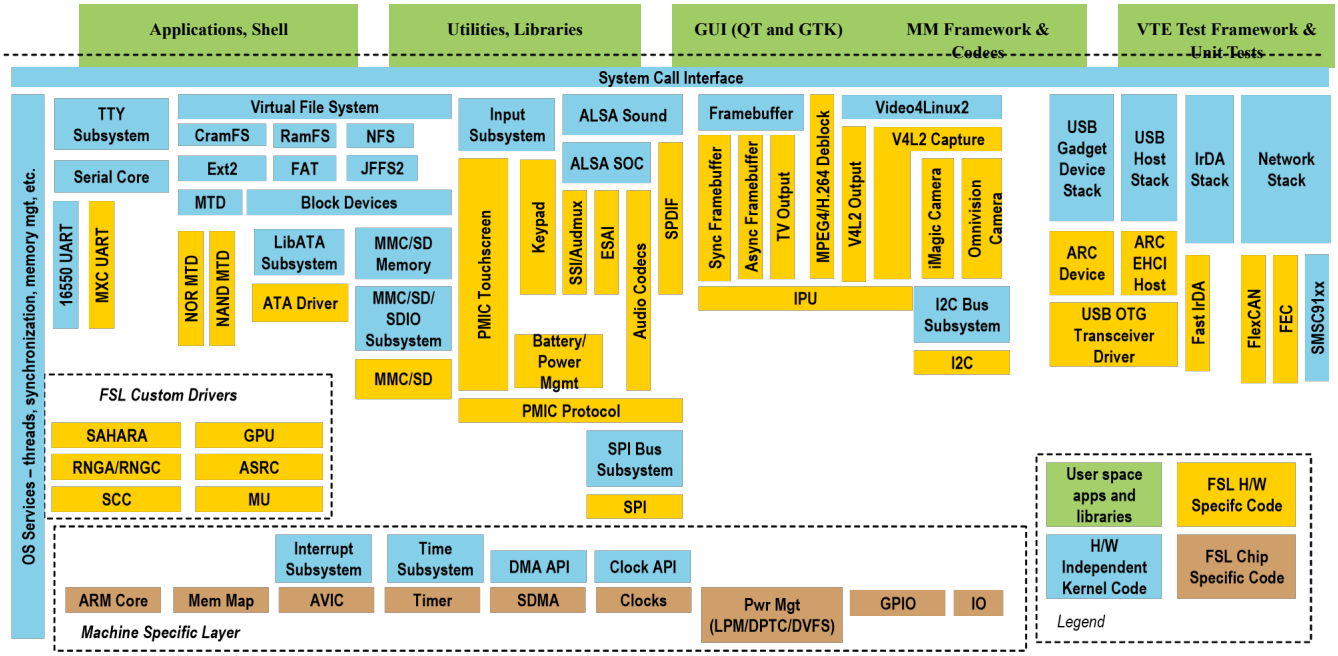



**Figure 2-1. BSP Block Diagram**

## 2.2 Kernel

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

- Process and thread management
- Memory management (memory mapping, allocation/deallocation, MMU, and L1/L2 cache control)
- Resource management (interrupts)
- Power management

- File systems (VFS, cramfs, ext2, ramfs, NFS, devfs, JFFS2, FAT, UBIFS)
- Linux Device Driver model
- Standardized APIs
- Networking stacks

ARM Linux Kernel customization to support each platform includes a custom kernel configuration and MSL implementation.

### 2.2.1 Kernel Configuration

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

- Embedded mode
- Module loading/unloading
- ARM9
- Supported file formats: ELF binaries, a.out, and ECOFF
- Block devices: Loopback, Ramdisk
- i.MX internal UART
- File systems: ext2, dev, proc, sysfs, cramfs, ramfs, JFFS2, FAT, pramfs
- Frame buffer
- Kernel debugging
- Automatic kernel module loading
- Power management
- Memory Technology Device (MTD) support
- USB Host/device multiplexing
- Unsorted Block Images (UBI) support
- Flash Translation Layer (FTL)
- CPU frequency scaling

### 2.2.2 Machine Specific Layer (MSL)

The MSL provides a machine-dependent implementation as required by the Linux kernel, such as memory map, interrupt, and timer. Each ARM platform has its own MSL directory under the arch/arm directory as listed in Table 2-1.

**Table 2-1. MSL Directories**

| Platform | Directory |
|---|---|
| i.MX28 | <ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx28 |

See Chapter 4, “Machine Specific Layer (MSL),” for more information.

#### 2.2.2.1 Memory Map

Before the kernel starts running in the virtual space, the physical-to-virtual address mapping for the I/O peripherals needs to be provided for the MMU to do the translation for memory/register accesses. The mapping is performed through a table structure in the MSL, specific to a particular platform, with each entry specifying a peripheral starting address of virtual addresses, starting address of physical addresses, and the size of the memory region and the type of the region.

#### 2.2.2.2 Interrupts

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

Together, they support the following capabilities:

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

#### 2.2.2.3 General Purpose Timer (GPT)

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

#### 2.2.2.4DMA API

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

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

•Channel interrupts control

#### 2.2.2.5 Input/Output (I/O)

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

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

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

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

#### 2.2.2.6 Pin Multiplexing

The pin multiplexing component is responsible for setting I/O pin configuration and routing. Each I/O pin is shared between up to three different i.MX28 modules or can be configured as a GPIO pin and controlled by software. The MSL implements a kernel-space API used by the MSL board specific components to set pins configurations corresponding to a particular board. The following features are implemented: • Pin resource manager to avoid conflicts on pin use

- •Pin voltage control
- •Pin strength control
- •Pin pull-up resistor control
- •Pin group configuration

#### 2.2.2.7 Shared Peripheral Bus Arbiter (SPBA)

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

## 2.3 Drivers

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

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

### 2.3.1 Universal Asynchronous Receiver/Transmitter (UART) Driver

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

#### 2.3.1.1 Debug Asynchronous Receiver/Transmitter (UART)

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

- •Interrupt driven transmit/receive of characters
- •Standard Linux baud rates up to 115 Kbps
- •Receive and transmit FIFOs support
- •Transmitting and receiving characters with 5, 6, 7 or 8-bit character lengths
- •Odd and even parity
- •CTS/RTS hardware flow control
- •Send and receive break characters through the standard Linux serial API
- •Recognize break and parity errors
- •Supports the standard TTY layer IOCTL calls
- •Console support required to bring up the command prompt through Debug serial port
- •Power management features by suspending and resuming UART ports

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

#### 2.3.1.2 Application Asynchronous Receiver/Transmitter (UART)

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

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

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

### 2.3.2 Real-Time Clock (RTC) Driver

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

### 2.3.3 Watchdog Timer (WDOG) Driver

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

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

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

### 2.3.4 DCP

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

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

### 2.3.5 i.MX28 Graphics

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

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

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

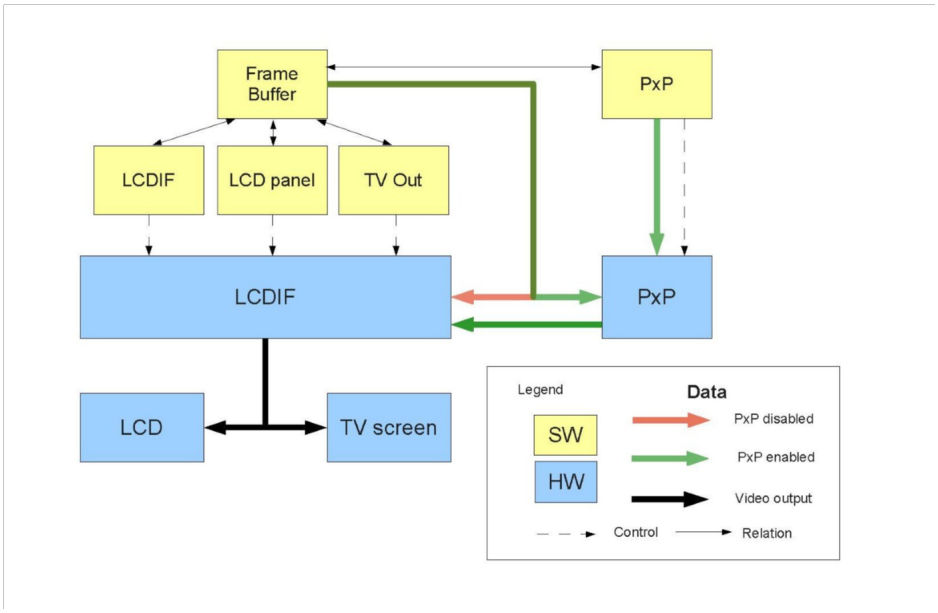



**Figure 2-2. i.MX28 Kernel Graphic Components**

### 2.3.5.1 LCDIF Driver

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

### 2.3.5.2 LCD Panel Drivers

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

### 2.3.5.3 Frame Buffer Driver

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

#### 2.3.5.4 Pixel Pipeline (PXP) Driver

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

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

### 2.3.6 Sound Driver

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

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

### 2.3.7 Keypad

The keypad driver interfaces Linux to the keypad ladder connected to the i.MX28 LRADC controller. The software operation of the driver follows the Linux keyboard architecture.The driver is driven by interrupts generated by the LRADC controller when changing a signal on the keypad ladder input pin. The driver reads a current voltage on the LRADC pin, detects which key is being pressed and sends a key code to the upper layer. The driver detects long key presses and reports them as multiple key press events.The keypad driver may be used as a wake-up source for low-power standby mode.

### 2.3.8 Memory Technology Device (MTD) Driver

MTDs in Linux cover all memory devices, such as RAM, ROM, and different kinds of Flashes. As each memory device has its own idiosyncrasies in terms of read and write, the MTD subsystem provides a unified and uniform access to the various memory devices.

Figure 2-3 shows the MTD architecture.

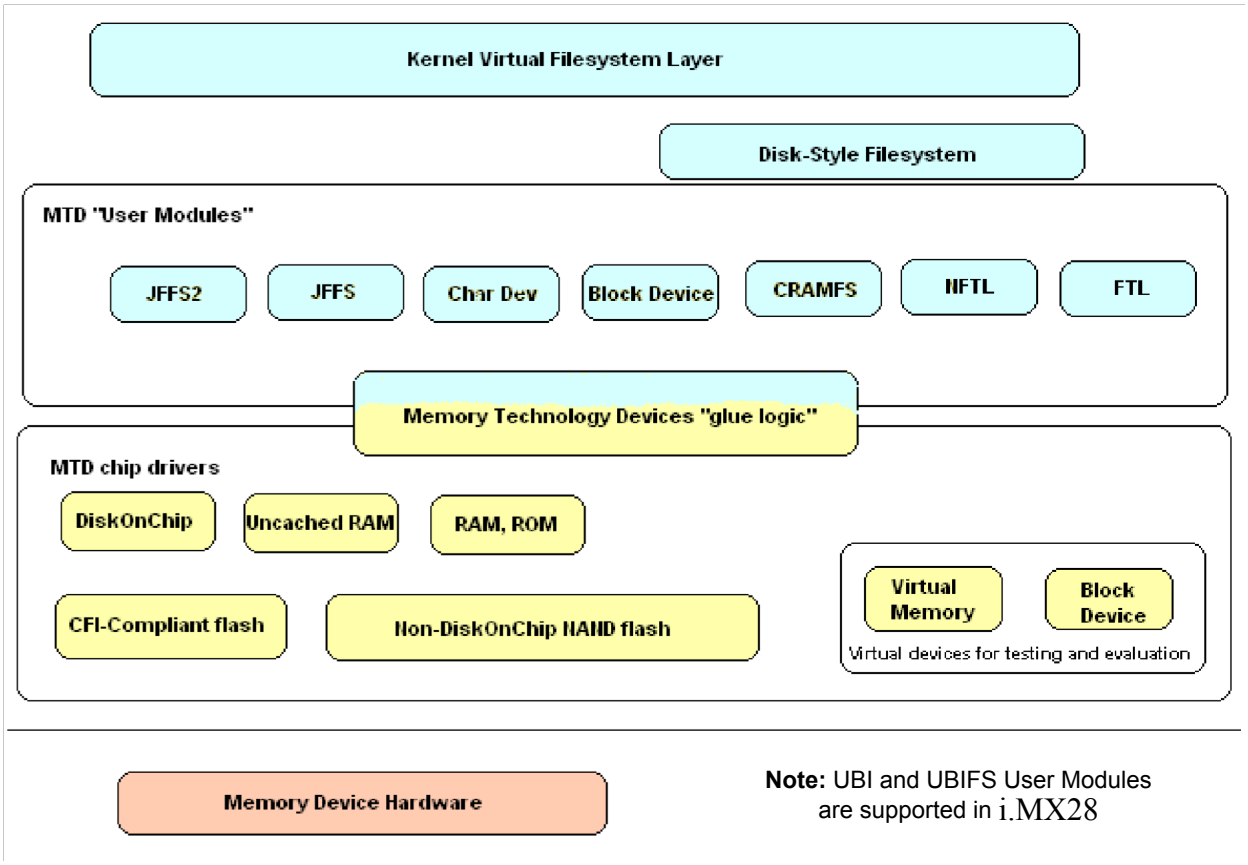



**Figure 2-3. MTD Architecture**

Figure 2-3 is excerpted from *Building Embedded Linux Systems*, which describes the MTD subsystem. The user modules should not be confused with kernel modules or any sort of user-land software abstraction. The term MTD user module refers to software modules within the kernel that enable access to the low-level MTD chip drivers by providing recognizable interfaces and abstractions to the higher levels of the kernel or, in some cases, to user space.

MTD chip drivers register with the MTD subsystem by providing a set of predefined callbacks and properties in the mtd_info argument to the add_mtd_device() function. The callbacks an MTD driver has to provide are called by the MTD subsystem to carry out operations, such as erase, read, write, and sync.

#### 2.3.8.1 GPMI/NAND

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

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

### 2.3.9 USB Driver

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

#### 2.3.9.1 USB Host-Side API Model

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

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

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

The device model seen by USB drivers is relatively complex:

- USB supports four kinds of data transfer (control, bulk, interrupt, and isochronous). Two transfer types use bandwidth as it is available (control and bulk), while the other two types of transfer (interrupt and isochronous) are scheduled to provide guaranteed bandwidth.
- The device description model includes one or more configurations per device, only one of which is active at a time. Devices that are capable of high speed operation must also support full speed configurations, along with a way to ask about the other speed configurations that might be used.

•Configurations have one or more interfaces. Interfaces may be standardized by USB Class specifications, or may be specific to a vendor or device.
•Interfaces have one or more endpoints, each of which supports one type and direction of data transfer such as bulk out or interrupt in.
•The only host-side drivers that actually touch hardware (reading/writing registers, handling IRQs, and so on) are the HCDs.

#### 2.3.9.2USB Device-Side Gadget Framework

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

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

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

#### 2.3.9.3USB OTG Framework

Systems need specialized hardware support to implement OTG, including a special Mini-AB jack and associated transceiver to support Dual-Role operation. They can act either as a host, using the standard Linux-USB host side driver stack, or as a peripheral, using the Gadget framework. To do that, the system software relies on small additions to those programming interfaces, and on a new internal component (here called an OTG Controller) affecting which driver stack connects to the OTG port. In each role, the system can re-use the existing pool of hardware-neutral drivers, layered on top of the controller driver interfaces (usb_bus or usb_gadget). Such drivers need at most minor changes, and most of the calls added to support OTG can also benefit non-OTG products.
•Gadget drivers test the is_otg flag, and use it to determine whether or not to include an OTG descriptor in each of their configurations.
•Gadget drivers may need changes to support the two new OTG protocols, exposed in new gadget attributes such as b_hnp_enable flag. HNP support should be reported through a user interface

(two LEDs could suffice), and is triggered in some cases when the host suspends the peripheral. SRP support can be user-initiated just like remote wakeup, probably by pressing the same button.

- On the host side, USB device drivers need to be taught to trigger HNP at appropriate moments, using usb_suspend_device(). That also conserves battery power, which is useful even for non-OTG configurations.
- Also on the host side, a driver must support the OTG Targeted Peripheral List, a whitelist used to reject peripherals not supported with a given Linux OTG host. This whitelist is product-specific—each product must modify otg_whitelist.h to match its interoperability specification.

Non-OTG Linux hosts, such as PCs and workstations, normally have some solution for adding drivers, so that peripherals that are not recognized can eventually be supported. That approach is unreasonable for consumer products that may never have their firmware upgraded, and where it is usually unrealistic to expect traditional PC/workstation/server kinds of support model to work. For example, it is often impractical to change device firmware after the product has been distributed, so driver bugs cannot normally be fixed if they are found after shipment.
Additional changes are required below those hardware-neutral usb_bus and usb_gadget driver interfaces but those are not discussed here. Those affect the hardware-specific code for each USB Host or Peripheral controller, and how the HCD initializes (since OTG can be active only on a single port). They also involve what may be called an OTG Controller Driver, managing the OTG transceiver and the OTG state machine logic as well as much of the root hub behavior for the OTG port. The OTG controller driver needs to activate and deactivate USB controllers depending on the relevant device role. Some related changes were required inside usbcore, so that it can identify OTG-capable devices and respond appropriately to HNP or SRP protocols.

### 2.3.10 General Drivers

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

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

#### 2.3.10.1 MMC/SD Host Driver

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

- MMC/SD cards

- Standard MMC/SD commands
- 1- bit or 4-bit operation

•Card insertion and removal events
•Write protection signal

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

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

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

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

### 2.3.10.4 Dynamic Power Management (DPM) Driver

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

•Operating points
•Operating states
•Policies
•Policy manager

#### 2.3.10.4.1 Policy Architecture

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

executing a particular DPM policy.

#### 2.3.10.4.2 Operating Points

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

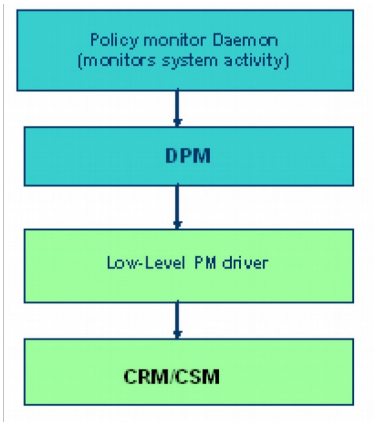
#### 2.3.10.4.3 Operating States

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

#### 2.3.10.4.4 Policy Managers

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

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



#### 2.3.10.5 Low-Level Power Management Driver

The low-level power management driver is responsible for implementing hardware-specific operations to meet power requirements and also to conserve power. Driver implementation may be different for different platforms. It is used by the DPM layer. This driver implements Dynamic Voltage and Frequency Scaling (DVFS) or Dynamic Frequency Scaling (DFS) techniques, depending on the platform, and low-power modes. The DVFS or DFS driver is used to change the frequency/voltage or frequency only when the DPM layer decides to change the operating point to meet the power

requirements. This is performed when the system is in RUN mode which helps in conserving power while the system is running. Low-power modes, such as WAIT and STOP are also implemented to save power. In all these cases, power consumption is managed by reducing the voltage/frequency and the severity of clock gating.

#### 2.3.10.6 Dynamic Voltage and Frequency Scaling (DVFS) Driver

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

#### 2.3.10.7 Backlight Driver

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

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

### 2.3.10.8 LED Driver

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

### 2.3.10.9 Power Source Manager and Battery Charger

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

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

Regardless of the power input, the power supply supplies voltage to several output voltage rails intended to power various on-chip and on-board components, such as ARM CPU core, SDRAM, peripheral I/O devices and so on. The way that these output voltages are generated depends on which power source is used. When the device is powered from a 5 V source, it uses internal voltage regulators to convert input voltage. When the device is powered from a battery source, it uses on-chip DC-DC converters. Certain software operations are required during transition from one power source to another, for which the power source manager driver is responsible. Also the power source manager notifies other drivers about power source changes.
Thei.MX28 power supply contains a built-in battery charger module capable of charging Li-Ion batteries. The battery charger driver implements a state machine that controls charging current and protects the battery from damage caused by under or overcharging.

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

### 2.3.10.10 CPUFreq Driver

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

## 2.4 Boot Loaders

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

- •Loads Linux kernel image to SDRAM
- •Obtains proper information for the Linux kernel
- •Passes control to the Linux kernel

**NOTE**

Not all boot loaders are supported on all boards.

### 2.4.1 i.MX28 Boot Loader

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

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

- •Boot prep
- •Linux prep
- •U-boot loader

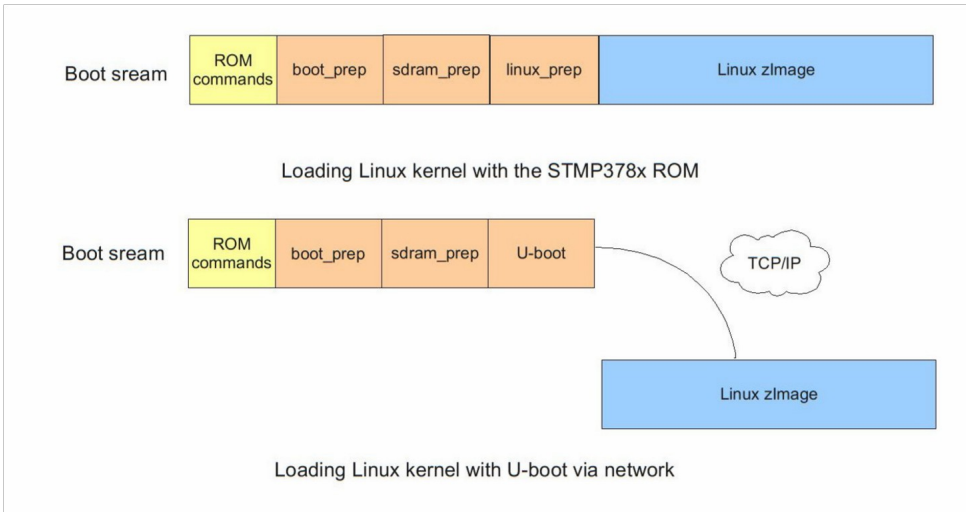



Figure 2-6 shows block diagrams of two boot stream images.

### 2.4.1.1 Boot Prep

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

### 2.4.1.2 Linux Prep

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

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

### 2.4.1.3 U-boot

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

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

# Chapter 3 Machine Specific Layer (MSL)

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

- Interrupts including GPIO and EDIO (only on certain platforms)
- Timer
- Memory map
- General Purpose Input/Output (GPIO) including IOMUX on certain platforms These

modules are normally available in the following directory:

< litb_dir>/rpm/BUILD/linux/arch/arm/mach-mx28 for imx28 platform The header files are implemented under the following directory:

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

The MSL layer contains not only the modules common to all the boards using the same processor, such as the interrupts and timer, but it also contains modules specific to each board, such as the memory map. The following sections describe the basic hardware and software operation and the software interfaces for MSL modules. First, the common modules, such as Interrupts and Timer are discussed. Next, the board-specific modules, such as Memory Map and General Purpose Input/Output (GPIO) (including IOMUX on some platforms) are detailed. Each of the following sections contains an overview of the hardware operation. For more information, see the corresponding device documentation.

## 3.1 Interrupts

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

### 3.1.1 Interrupt Hardware Operation

The Interrupt Collector module controls and prioritizes a maximum of 128 internal and external interrupt sources. Each source can be enabled and disabled by configuring the ENABLE bit in the dedicated Hardware Interrupt Collector Interrupt register. When an interrupt source is enabled and the

corresponding interrupt source is asserted, the Interrupt Collector asserts a normal or a fast interrupt request to the ARM core depending on the ENFIQ bit value in the dedicated Hardware Interrupt Collector Interrupt register.

The Interrupt Collectors interrupt requests are prioritized in the order of fast interrupts and normal interrupts in order of highest priority level. There are four normal interrupt levels, with zero level being the lowest priority. The interrupt levels are configurable through the PRIORITY bits of the Hardware Interrupt collector Interrupt register. Only in supervisor mode can the Interrupt Collector registers be accessed. A number of IRQ sources can be expanded by using GPIO lines to assert interrupts.

### 3.1.2 Interrupt Software Operation

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

model. The following file has a detailed description about the ARM interrupt architecture:
< ltib_dir>/rpm/BUILD/linux/Documentation/arm/Interrupts

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

### 3.1.3 Interrupt Source Code Structure

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

**Table 3-1. Interrupt Files List**

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

### 3.1.4 Interrupt Programming Interface

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

The irq_chip is associated with i.MX28normal 128 interrupt sources while mxs_gpio_chi is used for external GPIO interrupts. Each interrupt source is associated with one of the irq_chip structures with the set_irq_chip call. After initialization, the interrupt can be used by the drivers through the request_irq() and free_irq() functions to register device-specific interrupt handlers. Upon receiving the interrupt, the interrupt code uses get_irqnr_and_base to detect the interrupt source, acknowledges the interrupt using

the registered irq_chip structure set by the MSL, and calls the registered device-specific interrupt handler. Depending on the flags passed to the request_irq function, the code may disable the interrupt using an irq_chip call before executing the device-specific handler.

**Machine Specific Layer (MSL)**

## 3.2 Timer

The Linux kernel relies on the underlying hardware to provide support for both the system

timer (which generates periodic interrupts) and the dynamic timers (to schedule events). After the system timer interrupt occurs, it does the following:

- •Updates the system uptime
- •Updates the time of day
- •Reschedules a new process if the current process has exhausted its time slice
- •Runs any dynamic timers that have expired
- •Updates resource usage and processor time statistics  The timer hardware consists of four 32-bit 32 KHz timers.

### 3.2.1 Timer Hardware Operation

Each of the four timers consists of a 32-bit fixed count value and a 32-bit free-running count value. In most cases, the free-running count decrements to 0. When it decrements to 0, it sets an interrupt status bit associated with the counter, which causes:

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

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

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

### 3.2.2 Timer Software Operation

The timer software implementation provides an initialization function that initializes the GPT with the proper clock source, interrupt mode and interrupt interval. The timer then registers its interrupt service routine and starts timing. The interrupt service routine is required to service the OS for the purposes mentioned in Section 3.2, "Timer." Another function provides the time elapsed as the last timer interrupt.

### 3.2.3 Timer Features

The timer implementation supports the following features:

- •Functions required by Linux to provide the system timer and dynamic timers.
- •Generates an interrupt every 10 ms.

### 3.2.4 Timer Source Code Structure

The timer module is implemented in the arch/arm/plat-mxs/timer-match.c file.

### 3.2.5 Timer Programming Interface

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

## 3.3 Memory Map

A predefined virtual-to-physical memory map table is required for the device drivers to access to the device registers since the Linux kernel is running under the virtual address space with the Memory Management Unit (MMU) enabled.

### 3.3.1 Memory Map Hardware Operation

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

### 3.3.2 Memory Map Software Operation

A table mapping the virtual memory to physical memory is implemented for i.MX platforms as defined in the <ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx28/ mx28evk.cfile.

### 3.3.3 Memory Map Features

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

### 3.3.4 Memory Map Source Code Structure

The Memory Map module implementation is in mx28evk.c under the platform-specific MSL directory. The hardware.h header file is used to provide macros for all the IO module physical and virtual base addresses and physical to virtual mapping macros. All of the memory map source code is in the in the following directories:

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

< ltib_dir>/rpm/BUILD/linux/arch/arm/mach-imx

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

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

### 3.3.5 Memory Map Programming Interface

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

## 3.4 Pin Multiplexing

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

### 3.4.1 Pin Multiplexing Hardware Operation

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

The pin control interface has the following features:

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

### 3.4.2 Pin Multiplexing Software Operation

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

### 3.4.3 Pin Multiplexing Source Code Structure

The MSL Pin Multiplexing layer is implemented in the directories listed at the beginning of this chapter.

The files are listed in Table 3-4.

**Table 3-4. Pin Multiplexing Source Files**

| File | Description |
|---|---|
| mx28_pinsh | I/O pins definitions |

| | |
|---|---|
| pinctrl.c | Pin Multiplexing API implementation |

### 3.4.4 Pin Multiplexing Programming Interface

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

functions: enum pin_fun, enum pin_strength, enum pin_voltage

Define pin routing and configuration.

struct pin desc, struct pin group

Describe a group of pins.

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

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

void mxs_release_pin(unsigned id, char *label) Release the pin.

int mxs_request_pin_group(struct pin_group *pin_group, char *label) Request access to a group of pins.

void mxs_release_pin_group(struct pin_group *pin_group, char *label Release pin group.

void mxs_pin_strength(unsigned id, enum pin_strength strength, char *label) Set pin output strength.

void mxs_pin_voltage(unsigned id, enum pin_voltage voltage, char *label) Set pin output voltage.

void mxs_pin_pullup(unsigned id, int enable, char *label) Control pull up resistor of a pin.

### 3.4.5 GPIO With Pin Multiplexing

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

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

directories indicated at the beginning of this chapter.

# Chapter 4 Direct Memory Access Controller (DMAC) API

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

## 4.1 Hardware Operation

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

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

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

The DMA includes the following features:

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

DMA burst

**Direct Memory Access Controller (DMAC) API**

## 4.2 Software Operation

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

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

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

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

## 4.3 Source Code Structure

The header file, dmaengine.h, is available in the directory: arch/arm/plat-

mxs/include/mach/

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

**Table 4-1. DMA API Files**

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

## 4.4 Programming Interface

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

# Chapter 5 Persistent Bits Driver

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

## 5.1 Hardware Operation

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

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

## 5.2 Software Operation

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

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

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

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

For example, to read:

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

To write:

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

## 5.3 Source Code Structure

The persistent bit driver code listed in Table 5-1, is located in:
arch/arm/mach-mx28/include/mach/

arch/arm/mach-mx28

drivers/misc/

**Table 5-1. Persistent Bits Driver Files**

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

## 5.4 Menu Configuration Options

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

## 5.5 Programming Interface

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

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

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

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

# Chapter 6 Unique ID on Boot Media

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

- One-Time-Programmed (OTP) bits

## 6.1 Software Operation

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

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

The nature of UID storage forces some limits and assumptions:

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

## 6.2 Programming Interface

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

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

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

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

It completely removes the UID provider from the system.

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

## 6.3 Source Code Structure

The Unique ID module code listed in Table 6-1, is located in:
arch/arm/plat-mxs/include/mach/

arch/arm/plat-mxs/

**Table 6-1. Unique ID Files**

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

## 6.4 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_MXS_UNIQUE_ID = y
- CONFIG_MXS_UNIQUE_ID_OTP = y

# Chapter 7 CPU Frequency Scaling (CPUFREQ) Driver

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

## 7.1 Software Operation

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

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

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

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

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

The frequency 392727 is in KHz, which is 392.727 MHz. The maximum frequency can be checked using this command:

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

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

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

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

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

## 7.2 Source Code Structure

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

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

**Table 7-1. CPUFREQ Driver Files**

| File | Description |
|---|---|
| cpufreq.c | CPUFREQ functions |

## 7.3 Menu Configuration Options

The following Linux kernel configuration is provided for this module:

- CONFIG_CPU__FREQ—In menuconfig, this option is located under CPU Power Management > CPU Frequency scaling The following options can be selected:

  — CPU Frequency scaling
  — CPU frequency translation statistics
  — Default CPU frequency governor (userspace)
  — Performance
  governor —
  Powersave governor
  — Userspace governor for userspace frequency scaling
  — Conservative CPU frequency governor
  — CPU frequency driver for i.MX CPUs

### 7.3.1 Board Configuration Options

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

# Chapter 8 i.MX28 Static Power Management Driver

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

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

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

## 8.1 Hardware Operation

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

## 8.2 Software Operation

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

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

The wakeup sources and the system state can be set by the sysfs interface. To activate a wakeup

source, write 1 to /sys/bus/platform/devices/<device>/power/wakeup.

For example:

# echo 1 > /sys/bus/platform/devices/mxs-duart.0/power/wakeup To put the entire system into standby mode, run the following command:

# echo standby > /sys/power/state

## 8.3 Source Code Structure

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

**Table 8-1. Power Management Driver Files**

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

## 8.4 Menu Configuration Options

The following Linux kernel configurations are provided for this driver:

- CONFIG _PM [=Y]
  Generic configuration option to enable static power management. Once it is enabled, the source files listed above are automatically selected for compilation.

# Chapter 9 NAND GPMI Flash Driver

The NAND Flash Memory Technology Devices (MTD) driver is used in the Generic-Purpose Media Interface (GPMI) controller on the i.MX28. Only the hardware specific layer has to be implemented for the NAND MTD driver to operate. The rest of the functionality such as Flash read/write/erase is automatically handled by the generic layer provided by the Linux MTD subsystem for NAND devices.

## 9.1 Hardware Operation

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

## 9.2 Software Operation

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

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

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

### 9.2.1 Basic Operations: Read/Write

The NAND driver exports the following callbacks:

- mil_ecc_read_page (with ECC)
- mil_ecc_write_page ( with ECC )
- mil_read_byte ( without ECC )
- mil_read_buf ( without ECC )
- mil_write_buf (without ECC)
- mil_ecc_read_oob ( with ECC )



- mil_ecc_write_oob ( with ECC )

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

### 9.2.2 Error Correction

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

For BCH, the page laylout of 2K page is (2k + 64), the page layout of 4K page is (4k + 218).

### 9.2.3 Boot Control Block Management

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

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

### 9.2.4 Bad Block Handling

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

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

- mil_block_bad
- mil_scan_bbt

### 9.2.5 Special NAND supporting

## 9.3 Source Code Structure

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

- gpmi-nfc-main.c
- gpmi-nfc-mil.c
- gpmi-nfc-hal-common.c
- gpmi-nfc-hal-v0.c
- gpmi-nfc-hal-v1.c
- gpmi-nfc-hal-v2.c
- gpmi-nfc-event-reporting.c
- gpmi-nfc-rom-v0.c
- gpmi-nfc-rom-v1.c
- gpmi-nfc-rom-common.c
- gpmi-nfc.h
- gpmi-nfc-gpmi-regs-v0.h
- gpmi-nfc-gpmi-regs-v2.h
- gpmi-nfc-gpmi-regs-v3.h
- gpmi-nfc-bch-regs-v0.h
- gpmi-nfc-bch-regs-v1.h
- gpmi-nfc-bch-regs-v2.h

## 9.4 Menu Configuration Options

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

- CONFIG_MTD_NAND_GPMI_NFC = [Y | M]

In addition, these MTD options must be enabled:

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

In addition, these UBI options must be enabled:

- CONFIG_MTD_UBI=y
- CONFIG_MTD_UBI_WL_THRESHOLD=4096
- CONFIG_MTD_UBI_BEB_RESERVE=1
- CONFIG_UBIFS_FS=y
- CONFIG_UBIFS_FS_LZO=y
- CONFIG_UBIFS_FS_ZLIB=

# Chapter 10 I²C Driver

I²C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices. The I²C driver for Linux has two parts:

- I²C bus driver—low level interface that is used to talk to the I²C bus
- I²C chip driver—acts as an interface between other device drivers and the I²C bus driver

## 10.1 I²C Bus Driver Overview

The I²C bus driver is invoked only by the I²C chip driver and is not exposed to the user space. The standard Linux kernel contains a core I²C module that is used by the chip driver to access the I²C bus driver to transfer data over the I²C bus. The chip driver uses a standard kernel space API that is provided in the Linux kernel to access the core I²C module. The standard I²C kernel functions are documented in the files available under Documentation/i2c in the kernel source tree. This bus driver supports the following features:

- Compatible with the I²C bus standard
- Bit rates up to 400 Kbps
- Starts and stops signal generation/detection
- Acknowledge bit generation/detection
- Interrupt-driven, byte-by-byte data transfer
- Standard I²C master mode

## 10.2 I²C Device Driver Overview

The I²C device driver implements all the Linux I²C data structures that are required to communicate with the I²C bus driver. It exposes a custom kernel space API to the other device drivers to transfer data to the device that is connected to the I²C bus. Internally, these API functions use the standard I²C kernel space API to call the I²C core module. The I²C core module looks up the I²C bus driver and calls the appropriate function in the I²C bus driver to transfer data. This driver provides the following functions to other device drivers:

- •Read function to read the device registers
- •Write function to write to the device registers

The camera driver uses the APIs provided by this driver to interact with the camera.

## 10.3 Hardware Operation

The I²C module provides the functionality of a standard I²C master and slave. It is designed to be compatible with the standard Philips I²C bus protocol. The module supports up to 64 different clock frequencies that can be programmed by setting a value to the Frequency Divider Register (IFDR). It also generates an interrupt when one of the following occurs:

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

## 10.4 Software Operation

The I²C driver for Linux has two parts: an I²C bus driver and an I²C chip driver.

### 10.4.1 I²C Bus Driver Software Operation

The I²C bus driver is described by a structure called i2c_adapter. The most important field in this structure is struct i2c_algorithm *algo. This field is a pointer to the i2c_algorithm structure that describes how data is transferred over the I²C bus. The algorithm structure contains a pointer to a function that is called whenever the I²C chip driver wants to communicate with an I²C device.

During startup, the I²C bus adapter is registered with the I²C core when the driver is loaded. Certain architectures have more than one I²C module. If so, the driver registers separate i2c_adapter structures for each I²C module with the I²C core. These adapters are unregistered ( removed) when the driver is unloaded.

After transmitting each packet, the I²C bus driver waits for an interrupt indicating the end of a data transmission before transmitting the next byte. It times out and returns an error if the transfer

complete signal is not received. Because the I²C bus driver uses wait queues for its operation, other device drivers should be careful not to call the I²C API methods from an interrupt mode.

### 10.4.2 I²C Device Driver Software Operation

The I²C driver controls an individual I²C device on the I²C bus. A structure, i2c_driver, describes the I²C chip driver. The fields of interest in this structure are flags and attach_adapter. The flags field is set to a value I2C_DF_NOTIFY so that the chip driver can be notified of any new I²C devices, after the driver is loaded. The attach_adapter callback function is called whenever a new I²C bus driver is loaded in the system. When the I²C bus driver is loaded, this driver stores the i2c_adapter structure associated with this bus driver so that it can use the appropriate methods to transfer data.

## 10.5 Driver Features

The I²C driver supports the following features:

- I²C communication protocol
- I²C master mode of operation

**NOTE**

Tion.he I²C driver do not support the I²C slave mode of operat

## 10.7 Menu Configuration Options

- CONFIG_I2C_MXS

## 10.8 Programming Interface

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

## 10.9 Interrupt Requirements

The $I^2C$ module generates many kinds of interrupts. The highest interrupt rate is associated with the transfer complete interrupt as shown in Table 19-1.

**Table 19-1. $I^2C$ Interrupt Requirements**

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

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

# Chapter 11 MMC/SD/SDIO Host Driver

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

The MMC driver has the following features:

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

## 11.1 Hardware Operation

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

## 11.2 Software Operation

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

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

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

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

## 11.3 Driver Features

The MMC driver supports the following features:

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

## 11.4 Source Code Structure

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

## 11.5 Menu Configuration Options

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

- •CONFIG_MMC—Build support for the MMC bus protocol. In menuconfig, this option is available under
  Device Drivers > MMC/SD/SDIO Card
  support By default, this option is Y.
- •CONFIG_MMC_BLOCK—Build support for MMC block device driver, which can be used to mount the file system. In menuconfig, this option is available under Device Drivers > MMC/SD Card Support > MMC block device driver By default, this option is Y.
- •CONFIG_MMC_MXS—i.MX23/i.MX28 driver. In menuconfig, this option is available under Device Drivers > MMC/SD Card Support > Freescale MXC Multimedia Card Interface support.
- •CONFIG_MMC_UNSAFE_RESUME—Used for embedded systems which use a MMC/SD/SDIO card for rootfs. In menuconfig, this option is found under Device drivers > MMC/SD/SDIO Card Support > Allow unsafe resume.

## 11.6 Programming Interface

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

# Chapter 12 Universal Asynchronous Receiver-Transmitter (UART) Driver

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

## 12.1 Application UART

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

### 12.1.1 Hardware Operation

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

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

### 12.1.2 Software Operation

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

### 12.1.3 Source Code Structure

The application UART driver consists of the following files:

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

## 12.2 Debug UART

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

### 12.2.1 Hardware Operation

The debug UART performs:

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

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

### 12.2.2 Software Operation

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

### 12.2.3 Source Code Structure

The debug UART driver consists of the following files:

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

## 12.3 Menu Configuration Options

The following Linux kernel configurations are provided for this module:

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

# Chapter 13 USB Driver

The universal serial bus (USB) driver implements a standard Linux driver interface to the ARC USB-HS

OTG controller. The USB provides a universal link that can be used across a wide range of PC-to-peripheral interconnects. It supports plug-and-play, port expansion, and any new USB peripheral that uses the same type of port.

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

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

## 13.1 Architectural Overview

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

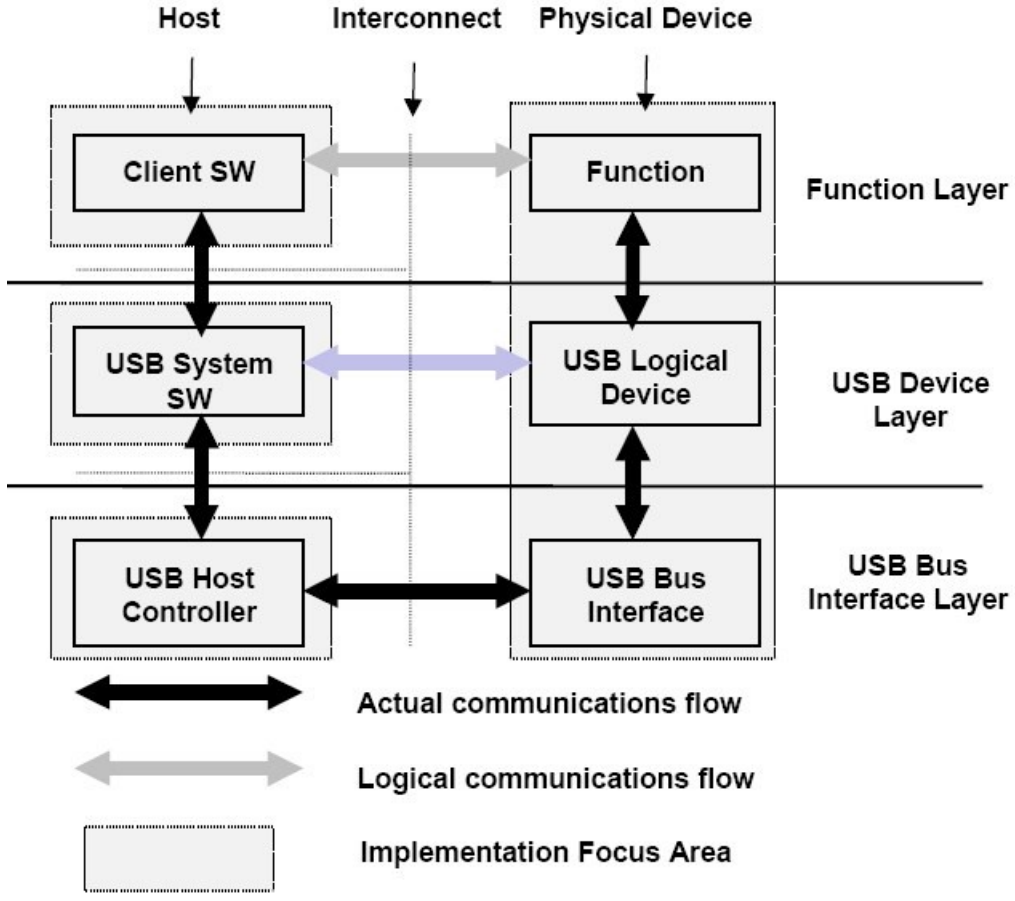



Figure 24-1. USB Block Diagram

## 13.2 Hardware Operation

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

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

## 13.3 Software Operation

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

```
static struct usb_ep_ops fsl_ep_ops = {
         .enable = fsl_ep_enable,


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

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

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

## 13.4 Driver Features

The USB stack supports the following features:

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

## 13.5 Source Code Structure

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

**Table 13-1. USB Driver Files**

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

Table 13-2 shows the platform related source files.

**Table 13-2. USB Platform Source Files**

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

Table 13-3 shows the platform-related source files in the directory:< ltib_dir>/rpm/BUILD/linux/arch/arm/mach-mx28/

**Table 13-3. USB Platform Header Files**

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

Table 13-4 shows the common platform source files in the directory:
< ltib_dir>/rpm/BUILD/linux/arch/arm/plat-mxs

**Table 13-4. USB Common Platform Files**

| File | Description |
|---|---|
| utmixc.c | Internal UTMI transceiver driver |
| usb_common.c | Common platform related part of USB driver |
| usb_wakeup.c | Handle usb wakeup events |

## 13.6 Menu Configuration Options

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

- •CONFIG_USB—Build support for USB
- •CONFIG_USB_EHCI_HCD—Build support for USB host driver. In menuconfig, this option is available under
  Device drivers > USB support > EHCI HCD (USB 2.0) support.
  By default, this option is M.
  CONFIG_USB_EHCI_ARC—Build support for selecting the ARC EHCI host. In menuconfig, this option is available underDevice drivers > USB support > Support for Freescale controller.
  By default, this option is Y.
- •CONFIG_USB_EHCI_ARC_H1—Build support for selecting the USB Host1. In menuconfig, this option is available underDevice drivers > USB support > Support for Host1 port on Freescale controller. By default, this option is Y.
- •CONFIG_USB_EHCI_ARC_OTG—Build support for selecting the ARC EHCI OTG host. In menuconfig, this option is available under

Device drivers > USB support > Support for Host-side USB > EHCI HCD (USB 2.0) support > Support for Freescale controller.
By default, this option is N.

- CONFIG_USB_STATIC_IRAM—Build support for selecting the IRAM usage for host. In menuconfig, this option is available under
  Device drivers > USB support > Use IRAM for USB.
  By default, this option is N.
- CONFIG_USB_EHCI_ROOT_HUB_TT—Build support for OHCI or UHCI companion. In menuconfig, this option is available under
  Device drivers > USB support > Root Hub Transaction Translators.
  By default, this option is Y selected by USB_EHCI_FSL && USB_SUPPORT.
- CONFIG_USB_STORAGE—Build support for USB mass storage devices. In menuconfig, this option is available under
  Device drivers > USB support > USB Mass Storage support.
  By default, this option is Y.
- CONFIG_USB_HID—Build support for all USB HID devices. In menuconfig, this option is available under
  Device drivers > HID Devices > USB Human Interface Device (full HID) support.
  By default, this option is Y.
- CONFIG_USB_GADGET—Build support for USB gadget. In menuconfig, this option is available under
  Device drivers > USB support > USB Gadget Support.
  By default, this option is M.
- CONFIG_USB_GADGET_ARC—Build support for ARC USB gadget. In menuconfig, this option is available under
  Device drivers > USB support > USB Gadget Support > USB Peripheral Controller (Freescale USB Device Controller).
  By default, this option is Y.
- CONFIG_USB_OTG—OTG Support, support dual role with ID pin detection.
  By default, this option is N.
- CONFIG_UTMI_MXC_OTG—USB OTG pin detect support for UTMI PHY, enable UTMI PHY for OTG support.
  By default, this option is N.
- CONFIG_USB_ETH—Build support for Ethernet gadget. In menuconfig, this option is available under
  Device drivers > USB support > USB Gadget Support > Ethernet Gadget (with CDC Ethernet Support).
  By default, this option is M.
- CONFIG_USB_ETH_RNDIS—Build support for Ethernet RNDIS protocol. In menuconfig, this option is available under
  Device drivers > USB support > USB Gadget Support > Ethernet Gadget (with CDC Ethernet Support) > RNDIS support.
  By default, this option is Y.
- CONFIG_USB_FILE_STORAGE—Build support for Mass Storage gadget. In menuconfig, this option is available under
  Device drivers > USB support > USB Gadget Support > File-backed Storage Gadget.
  By default, this option is M.

•CONFIG_USB_G_SERIAL—Build support for ACM gadget. In menuconfig, this option is available under
Device drivers > USB support > USB Gadget Support > Serial Gadget (with CDC ACM support).
By default, this option is M.

## 13.7 Programming Interface

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

## 13.8 Default USB Settings

Table 24-5 shows the default USB settings.

**Table 24-5. Default USB Settings**

| Platform | OTG HS | OTG FS | Host1 | Host2(HS) | Host2(FS) |
|---|---|---|---|---|---|
| i.MX28 EVK | enalbed | NA | enable | N/A | — |

By default, both usb device and host function are build-in kernel, otg port is used for device mode, and host 1 is used for host mode.

The default configuration does not enable OTG port for both device and host mode. To enable USB-OTG for both host and device mode, configure the kernel as follows and rebuild the kernel and modules:

•CONFIG_USB_EHCI_ARC_OTG—Enable support for the USB OTG port in HS/FS Host mode.built as Y
•CONFIG_USB_GADGET—USB Gadget Support: built as y
•CONFIG_USB_OTG —OTG Support: built as Y
•CONFIG_MXC_OTG—USB OTG pin detect support for UTMI PHY: built as Y
•build USB GADGET driver as M, for example:
CONFIG_USB_ETH CONFIG_USB_FILE_STORAGEthen , if you want to use EVK as mass storage device, insmod g_file_storage.ko file=/dev/mmcblk0p2
if you want to use the otg as ethernet, insmod g_ether.ko , then you can use ifconfig usb0 to configure the ip

## 13.9 System WakeUp

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

## 13.10 USB Wakeup usage

### 13.10.1 How to enable usb wakeup system ability

For otg port:

```
echo enabled > /sys/devices/platform/fsl-
```

usb2-otg/power/wakeup For device-only port:

```
echo enabled > /sys/devices/platform/fsl-
```

usb2-udc/power/wakeup For host-only port:

```
echo enabled > /sys/devices/platform/fsl-
ehci.x/power/wakeup ( x is the port num )
```

For usb child device `echo enabled >`

```
/sys/bus/usb/devices/1-
1/power/wakeup
```

### 13.10.2 What kinds of wakeup event usb support

Take USBOTG port as the example.

Device mode wakeup:

-connect wakeup: when usb line connects to usb port, the other port is connected to PC (Wakeup signal: vbus change) echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup

Host mode wakeup:

-connect wakeup: when usb device connects to host port (Wakeup signal: ID/(dm/dp) change )

echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup

-disconnect wakeup: when usb device disconnects to host port (Wakeup signal: ID/(dm/dp) change)

echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup

-remote wakeup: press usb device (such as press usb key at usb keyboard) when usb device connects to host port (Wakeup signal: ID/(dm/dp) change):

```
echo enabled > /sys/devices/platform/fsl-usb2-
otg/power/wakeup echo enabled > /sys/bus/usb/devices/1-
1/power/wakeup
```

NOTE: For the hub on board, it needs to enable hub's wakeup first. for remote wakeup, it needs to do below three steps:

echo enabled > /sys/devices/platform/fsl-usb2-otg/power/wakeup (enable the roothub's

wakeup) echo enabled > /sys/bus/usb/devices/1-1/power/wakeup (enable the second level hub's wakeup)
(1-1 is the hub name )

echo enabled > /sys/bus/usb/devices/1-1.1/power/wakeup (enable the usb device's wakeup, that device connects at second level hub)
(1-1.1 is the usb device name )

### 13.10.3 How to close the usb child device power

echo auto > /sys/bus/usb/devices/1-1/power/control echo auto > /sys/bus/usb/devices/1-1.1/power/control (If there is a hub at usb device)

# Chapter 14 Real Time Clock (RTC) Driver

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

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

## 14.1 Hardware Operation

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

## 14.2 Software Operation

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

**NOTE**

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

## 14.3 Source Code Structure

The RTC module is implemented in the <ltib_dir>/rpm/BUILD/linux/drivers/rtc directory. Table 14-1 shows the RTC module files. The source file for the RTC specifies the RTC function implementations.

**Table 14-1. RTC Driver File List**

| File | |
|---|---|
| rtc-mxs.c | RTC driver |



## 14.4 Programming Interface

All the Linux RTC functions are based on rtclib. The include/linux/rtc.h file specifies all the IOCTLs for the RTC. Table 14-1 shows the IOCTLs that are listed in include/linux/rtc.h and which are supported by the RTC driver.

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

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

- CONFIG_RTC_DRV_MXS [=M|Y]
  This is the configuration option for the RTC driver, which is dependent on the RTC_CLASS option. In menuconfig, this option is available under: Real Time Clock > Freescale MXS series
  SoC RTC

# Chapter 15 Watchdog (WDOG) Driver

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

## 15.1 Hardware Operation

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

## 15.2 Software Operation

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

# Chapter 16  External Devices

## 16. 1 Audio Codec

## 16.2 Ethernet

## 16.3 WiFi Bluetooth module

## 16.4 ADC

### 16.4.1 Software Operation

The available registers (sys interface) are the following:

adc0, adc1, adc2, adc3

root@freescale ~$ cat /sys/class/input/input0/device/**adc0,1,2,3**

## 16. 5 PM wake-up

### 16.5.1 Software Operation

Interrupts that are involved in the wake-up at the moment:

- **INT1** accelerometer
- **IN1_H**
- **IGN_H**

The micro can go into standby also via the RTC and be awakened after a set time.

To bring the micro in standby,  these are the steps:

- for the INTs:

root@freescale ~$ **echo standby > /sys/power/state**

- for RTC:

root@freescale ~$ **rtcwake -d /dev/rtc0 -s <time sec> standby**

# Chapter 17 Board Programming

## 17.1 SD

The archive containing the binaries is Install-imx28-SD.tgz.

These are the instructions:

- Program kernel

  copy the kernel imx28_ivt_linux_xxx.sb in rootfs/boot/imx28_ivt_linux.sb

  sudo ./mk_mx28_sd -b -x /dev/sdx

- Program filesystem

  sudo ./mk_mx28_sd -r -x /dev/sdx

- Program kernel + filesystem

  sudo ./mk_mx28_sd -x /dev/sdx

## 17.2 NAND

The archive the archive containing the binaries is Install-imx28-NAND.tgz.

- imx28_ivt_uboot.sb
- install_kernel.sh
- install_rootfs.sh
- install_u-boot.sh
- rootf.tgz
- uImage

The programming procedure must be performed by  linux from SD.

- To program **u-boot** is necessary to run the script:

```
$ cd Install-imx28-NAND
$ ./install_u-boot.sh
```

- To program the **kernel** is necessary to run the script:

```
$ ./install_kernel.sh
```

-To program **filesystem** is necessary to run the script

```
$ ./install_rootfs.sh
```

At the end you can turn off and restart the card from the NAND:

```
U-Boot 2009.08-dirty (feb 26 2014 - 08:20:25)

Freescale i.MX28 family
CPU:   454 MHz
BUS:   151 MHz
EMI:   205 MHz
GPMI:   24 MHz
DRAM:  128 MB
NAND:  256 MiB
In:    serial
Out:   serial
Err:   serial
Net:   got MAC address from IIM: 00:04:00:00:00:00
FEC0
Hit any key to stop autoboot:  0

NAND read: device 0 offset 0x240000, size 0x300000
 3145728 bytes read: OK
## Booting kernel from Legacy Image at 42000000 ...
   Image Name:   Linux-2.6.35.3-670-g914558e
   Image Type:   ARM Linux Kernel Image (uncompressed)
   Data Size:    2619884 Bytes =  2.5 MB
   Load Address: 40008000
   Entry Point:  40008000
   Verifying Checksum ... OK
   Loading Kernel Image ... OK
OK

Starting kernel ...

Uncompressing Linux... done, booting the kernel.
Linux version 2.6.35.3-670-g914558e (nicola@nicola) (gcc version 4.4.4
```

## Rohs compliance

The MCAM335x Standalone Embedded CPU Board comply with the European Union's Directive 2002/95/EC: "Restrictions of Hazardous Substances".

## Warranty Terms

MAS Elettronica guarantees hardware products against defects in workmanship and material for a period of one (1) year from the date of shipment. Your sole remedy and MAS Elettronica's sole liability shall be for MAS Elettronica, at its sole discretion, to either repair or replace the defective hardware product at no charge or to refund the purchase price. Shipment costs in both directions are the responsibility of the customer. This warranty is void if the hardware product has been altered or damaged by accident, misuse or abuse.

Disclaimer of Warranty
THIS WARRANTY IS MADE IN LIEU OF ANY OTHER WARRANTY, WHETHER EXPRESSED, OR IMPLIED, OF MERCHANTABILITY, FITNESS FOR A SPECIFIC PURPOSE, NON-INFRINGEMENT OR THEIR EQUIVALENTS UNDER THE LAWS OF ANY JURISDICTION, EXCEPT THE WARRANTY EXPRESSLY STATED HEREIN. THE REMEDIES SET FORTH HEREIN SHALL BE THE SOLE AND EXCLUSIVE REMEDIES OF ANY PURCHASER WITH RESPECT TO ANY DEFECTIVE PRODUCT.
Limitation on Liability
UNDER NO CIRCUMSTANCES SHALL MAS ELETTRONICA BE LIABLE FOR ANY LOSS, DAMAGE OR EXPENSE SUFFERED OR INCURRED WITH RESPECT TO ANY DEFECTIVE PRODUCT. IN NO EVENT SHALL MAS ELETTRONICA BE LIABLE FOR ANY INCIDENTAL OR CONSEQUENTIAL DAMAGES THAT YOU MAY SUFFER DIRECTLY OR INDIRECTLY FROM USE OF ANY PRODUCT.

# Contact Informations

Headquarters

Mas Elettronica Sas
Via Rossi 1
35030 Rubano (PD)
Italy.
Tel  +39 0498687469
Fax +39 0498687469

Sales : amm@maselettronica.com
Support: info@maselettronica.com