Chapter 1 16C652 Universal Asynchronous Receiver/Transmitter (UART) Driver

The external SCSC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) runs on the i.MX31 and i.MX31L multimedia applications processors. For driver development, the serial driver files for the 2.6 Linux kernel are modified.

Features of the SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) for the MXC family of processors include:

- Interrupt-driven transmit/receive of characters.
- Supports standard Linux baud rates from 115K baud down to 50 baud.
- Supports two UART ports on 16C652, four UART ports on 15C6552.
- Supports transmitting and receiving characters with 7-bit and 8-bit character lengths.
- Supports transmitting 1, 1.5 or 2 stop bits.
- Supports odd and even parity.
- Supports XON/XOFF software flow control.
- Supports CTS/RTS hardware flow control.
- Send and receive break characters through the standard Linux serial API.
- Recognizes frame and parity errors.
- Ability to ignore characters with break, parity and frame errors.
- Get and set UART port information through the TIOCGSSERIAL and TIOCSSERIAL TTY ioctls.
- Supports the standard TTY layer ioctl calls.
- Includes console support that is needed to bring up the command prompt through one of the UART ports.

Power management, autobaud detection and DMA are not supported by the SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART).

1.1 SC16C652/ST16C2552 UART Hardware Operation

The Universal Asynchronous Receiver/Transmitter (UART) controller is the key component of the serial communications subsystem of a computer. At the destination, a second UART re-assembles the bits into complete bytes. During transmission, the UART converts the bytes from the processors parallel bus to the serial bit stream. During receiving, the UART builds the serial bits into a parallel byte.

The UART is the SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) dual UART model, and is interfaced to the Peripheral Bus Controller (PBC). The Peripheral Bus Controller (PBC) is used to interface the CPU board bus with the busses used by peripherals. It provides an LPC (Low Pin Count) interface for access to the on-board UART controller.

The block diagram of the PBC interface is shown in Figure 1-1:

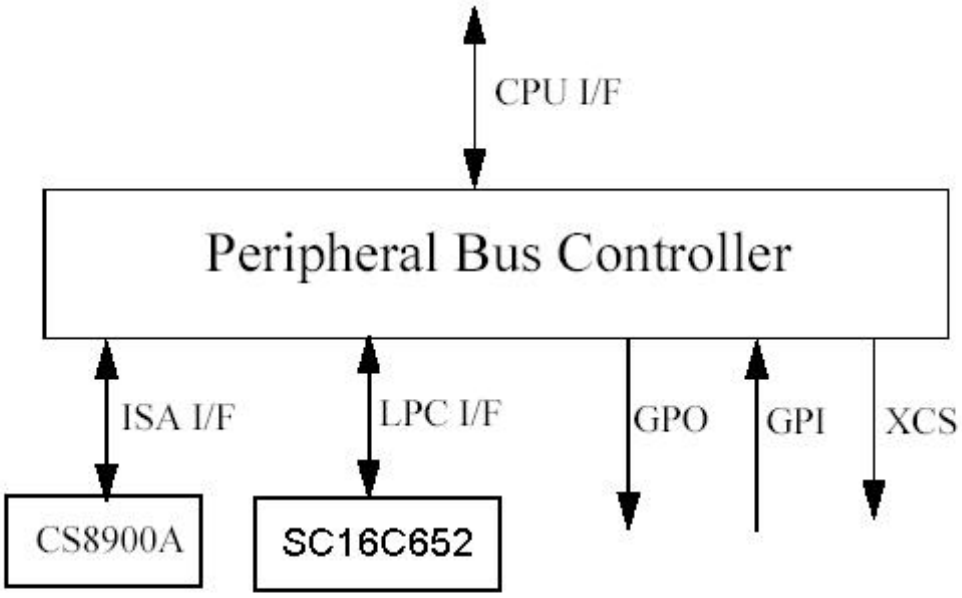

Figure 1-1. SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) PCB Interface

The Block diagram of SC16C652/ST16C6552 is found in Figure 1-2.

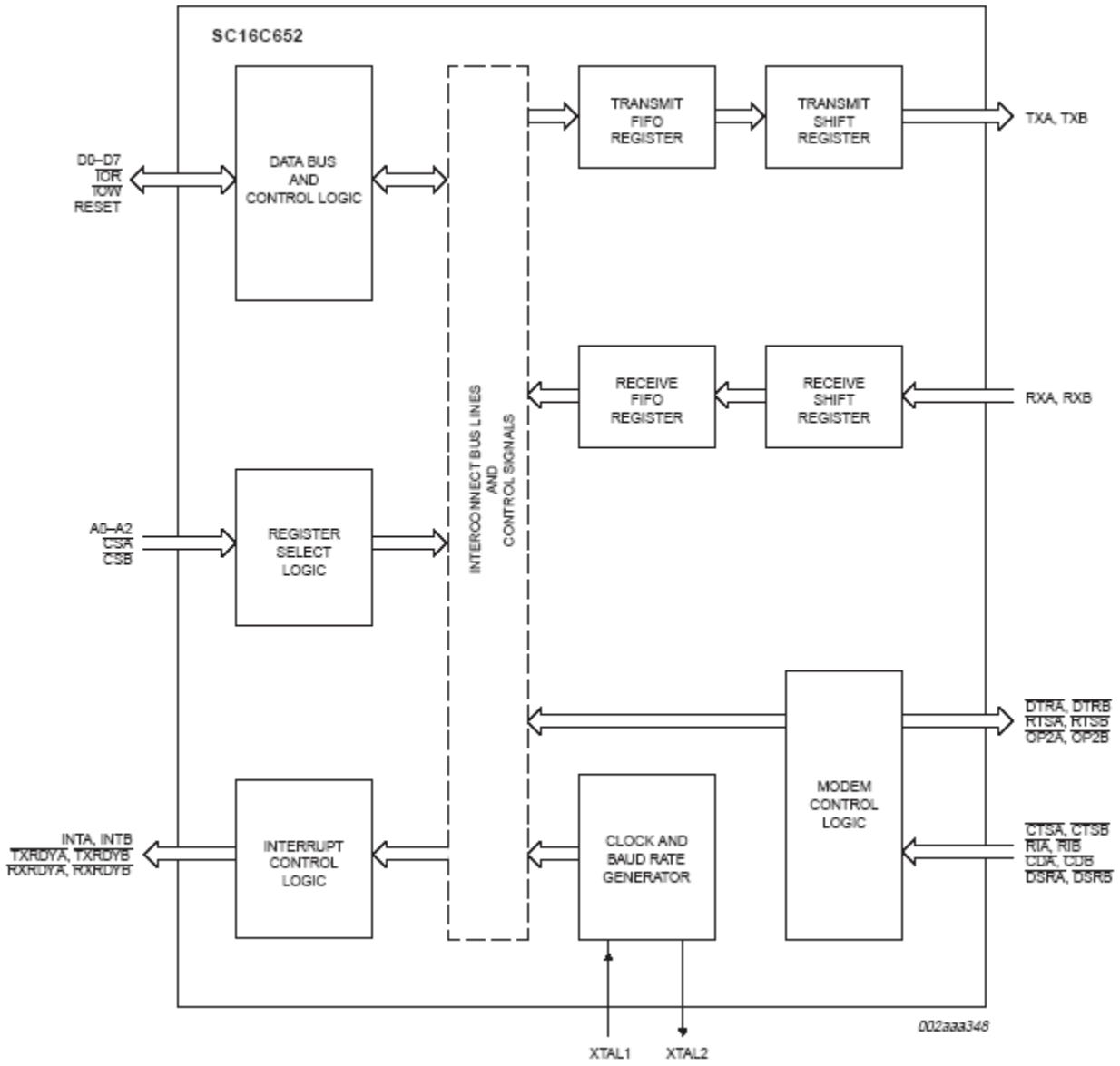

Figure 1-2. SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) Block Diagram

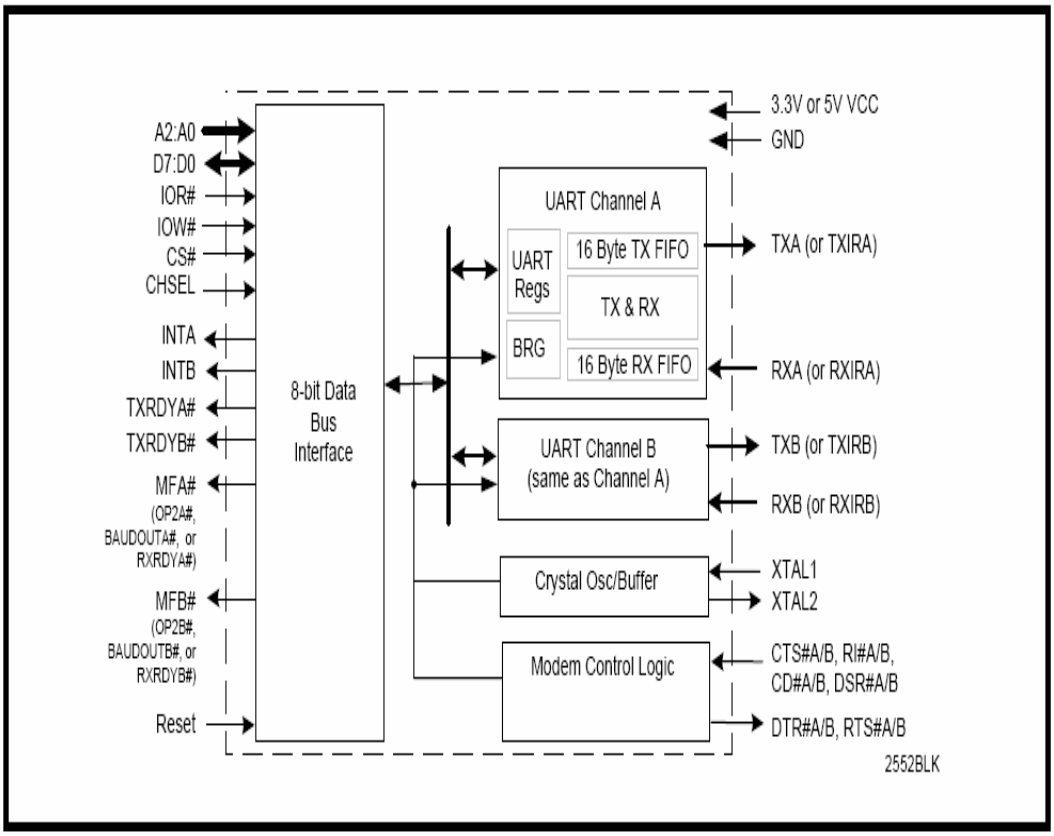

Figure 1-3. 16C652 UART Driver Block Diagram

The base address and the offset gives the location of the UART register. Addressing of the accessible registers of the Serial Port is shown in Figure 1-4.

DLAB*	A2	A1	A0	REGISTER NAME
0	0	0	0	Receive Buffer (read)
0	0	0	0	Transmit Buffer (write)
0	0	0	1	Interrupt Enable (read/write)
X	0	1	0	Interrupt Identification (read)
X	0	1	0	FIFO Control (write)
X	0	1	1	Line Control (read/write)
X	1	0	0	Modem Control (read/write)
X	1	0	1	Line Status (read/write)
X	1	1	0	Modem Status (read/write)
X	1	1	1	Scratchpad (read/write)
1	0	0	0	Divisor LSB (read/write)
1	0	0	1	Divisor MSB (read/write

Figure 1-4. Serial Port Register Addressing for the SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART)

NOTE

*DLAB - 7th bit of Line Control Register.

References:

- Peripheral Bus Controller Specs

1.2 SC16C652/ST16C2552 UART Software Operation

The SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) provides standard serial driver interface to the Linux operating system. The driver works in interrupt mode and makes use of the UART FIFO for optimum operation.

1.3 SC16C652/ST16C2552 UART Requirements

The SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) has the following requirements:

- The 16C652 UART is used as a debug port to bring up the console for development purposes.
- The SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) Driver, at a minimum must support baud rates up to 115200bps with no handshaking, 8-bit character length, 1 Stop bit and no Parity.
- The 16C652 UART driver must support at a minimum of one 16C652 UART.

1.4 SC16C652/ST16C2552 UART Source Code Structure

Table 1-1 lists the files associated with SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) development:

Table 1-1. 16552 UART Driver File List

File	Description
8250.h	Driver header file
8250.c	Driver source file
serial.h	Architecture specific configuration header file

1.5 16C652 UART Driver Configuration

This section documents the configuration options available for the external UART.

1.5.1 Linux Menu Configuration Options

The following menu options are available for the external 16C652 UART. These options are found are found under Characters devices->Serial drivers:

1. 8250/16550 and compatible serial support = Y

2. Console on 8250/16550 and compatible serial port = Y
3. Maximum number of non-legacy 8250/16550 serial ports = two (one for ST16C6552)

By default, for all architectures, the first two options are N and the last option is four.

1.5.2 Source Code Configuration Options

1.5.2.1 Chip Configuration Options

The following chip-specific configuration options for the driver are provided in `serial.h`:

1. UART1 interrupt (IRQ_UARTINT0) - This specifies the UART1 interrupt number.
2. UART2 interrupt (IRQ_UARTINT1) - This specifies the UART2 interrupt number.
3. UART1 base address (SERIAL1_BASE_ADDRESS) - This specifies the UART1 base address
4. UART2 base address (SERIAL2_BASE_ADDRESS) - This specifies the UART2 base address
5. PBC Base address (PBC_BASE_ADDRESS) - This specifies the PBC base address
6. PBC IRQ status (IRQSTAT) - PBC interrupt status register.

1.6 SC16C652/ST16C2552 UART Programming Interface

The SC16C652/ST16C2552 Universal Asynchronous Receiver/Transmitter (UART) implements all the methods that are required by the Linux serial API to interface with the UART ports. It implements and provides a set of control methods to the core UART driver present in Linux.

1.7 16C652 UART Driver Interrupt Requirements

The system requirements are shown in Table 1-2.

Table 1-2. 16C652 UART Interrupt Requirements

Parameter	Equation	Typical	Worst-Case
Rate	(BaudRate/(10))*(1/Rxtl + 1/32)	5940 /sec	118800 /sec
Latency	320/BaudRate	5.6 msec	213.33 msec

The typical values are based on a baud rate of 57600 with a receiver trigger level (Rxtl) of 1. Transmitter trigger level is not set. The worst-case is based on a baud rate of 1.152 Mbps (max supported by the UART interface) with an Rxtl of 1. The interrupt latency for typical (9600) and lowest (50) baud rates.

1.8 *Unit Test*

The following is a list of unit tests that should be performed:

1. Detection of device.
2. Opening of device.
3. Closing of device.

4. Transmission/Reception of text file
5. Transmission/Reception of binary file.
6. Test for different baud rates/stop/parity bits
7. Redirect the output of a shell command to another UART.
   — `ls >/dev/tts/0` -- the output should be printed on UART2's window
   — `ls >/dev/tts/1` -- the output should be printed on UART3's window
8. Start a shell in the background and redirect its standard input, standard output and standard error to a different UART port.

```
/bin/sh </dev/tts/1 >/dev/tts/1 2>/dev/tts/1 &
```

The following are the unit tests performed on 16C2552.

1. Boot up with external uart as console—On the board, connect external uart to a serial port on your PC. Open a Hyperterminal window that is connected to this serial port.Use redboot download kernel, and then use exec command as follows to boot up, the baud rate must be same with the set of Hypterminals:

```
exec -b 0x800000 -l 0x200000 -c "rw root=/dev/nfs nfsroot=10.193.100.211:/
home/r65130/rootdisk/mx21 console=ttyS0,115200"
```

2. Transmission/Reception of text file—Use command "`stty -F /dev/ttymxc/0 115200`" to uart1, and then use "`cat >/dev/ttymxc/0`", send text file from the external uart, will see the text file contents on the hyperterminals connected with uart1.
3. Transmission/Reception of binary file—Same as case 2 except use binary file.
4. Test for different baud rates/stop/parity bits—Use uart1 as console, use "`stty`" with different baud rates/stop/parity bits to `/dev/tts/0`, use "`cat >/dev/tts/0`" or "`cat /dev/tts/0`" to test if can receive and transmit data.
5. Redirect the output of a shell command to another UART—Use command "`stty -F /dev/ttymxc/0 115200`" to uart1, and then use `ls >/dev/ttymxc/0` -- the output should be printed on UART1's window.

Chapter 2
MXC Universal Asynchronous Receiver/Transmitter (UART) Driver

The low-level MXC Universal Asynchronous Receiver Transmitter (UART) Driver interfaces the Linux serial driver API to all the MXC UART ports. It has the following features:

- Supports interrupt-driven and SDMA-driven transmit/receive of characters.
- Supports standard Linux baud rates up to 1.5Mbps.
- Supports transmitting and receiving characters with 7-bit and 8-bit character lengths.
- Supports transmitting 1 or 2 stop bits.
- Supports `TIOCMGET` `ioctl` to read the modem control lines. Supports only the constants `TIOCM_CTS` and `TIOCM_CAR`, plus `TIOCM_RI` in `DTE` mode only.
- Supports `TIOCMSET` `ioctl` to set the modem control lines. Supports the constants `TIOCM_RTS` and `TIOCM_DTR` only.
- Supports odd and even parity.
- Supports XON/XOFF software flow control. Serial communication using software flow control is reliable when communication speeds are not too high and the probability of buffer overruns is minimal.
- Supports CTS/RTS hardware flow control (both interrupt-driven software-controlled hardware flow and hardware-driven hardware-controlled flow).
- Send and receive break characters through the standard linux serial API.
- Recognize frame and parity errors.
- Ability to ignore characters with break, parity and frame errors.
- Get and set UART port information through the `TIOCGSSERIAL` and `TIOCSSERIAL` TTY ioctls. Some programs like `setserial` and `dip` use this feature to make sure that the baud rate was set properly and to get general information on the device. While doing this the user should specify the UART type to be 52. This is defined in the `serial_core.h` header file.
- Serial IrDA support.
- Supports power management feature by suspending and resuming the UART ports
- Supports the standard TTY layer `ioctl` calls.

All the MXC UART ports can be accessed through the device files `/dev/ttymxc/0` through `/dev/ttymxc/4`, where `/dev/ttymxc/0` refers to MXC UART 1 and so on. The number of available UART ports varies from device to device.

Autobaud detection is not supported.

2.1 MXC UART Driver Hardware Operation

Refer to the Detailed Technical Specification to determine the number of UART modules available in your device. Each UART hardware port is capable of standard RS-232 serial communication and has support for IrDA 1.0. Each UART contains a 32-byte transmitter FIFO and a 32-half-words deep receiver FIFO.

They also support a variety of maskable interrupts when the data level in each FIFO reaches a programmed threshold level and when there is a change in state in the modem signals. Each UART can be programmed to be in DCE or DTE mode.

2.2 MXC UART Driver Software Operation

The Linux OS contains a core UART driver that handles a lot of the serial operations that are common across UART drivers for various platforms. The MXC low-level UART driver is responsible for supplying to this core UART driver such information as the UART port information and a set of control functions. These functions are implemented as a low-level interface between the Linux OS and the UART hardware. They cannot be called from other drivers or from a user application. The control functions used to control the hardware are passed to the core driver through a structure called `uart_ops`, and the port information is passed through a structure called `uart_port`. The low level driver is also responsible for handling the various interrupts for the UART ports, and providing console support if necessary.

Each UART can be configured to use DMA for the data transfer. These configuration options are provided in the `mxc_uart.h` header file. The user can specify the size of the DMA receive buffer. The minimum size of this buffer is 512 bytes. The size should be a multiple of 256. The driver breaks the DMA receive buffer into smaller sub-buffers of size 256 bytes and registers these buffers with the DMA system. The DMA transmit buffer size is fixed at 1024 bytes.The size is limited by the size of the Linux UART transmit buffer (1024).

The driver requests 2 DMA channels for the UARTs that need DMA transfer. On a receive transaction, the driver copies the data from the DMA receive buffer to the TTY Flip Buffer.

While using DMA to transmit, the driver copies the data from the UART transmit buffer to the DMA transmit buffer and sends this buffer to the DMA system. The user should use hardware-driven hardware flow control when using DMA data transfer. For more information, see the Linux documentation on the serial driver in the kernel source tree.

The low-level driver supports both interrupt-driven software-controlled hardware flow control and hardware-driven hardware flow control. The hardware flow control method can be configured using the options provided in the header file. The user has the capability to de-assert the CTS line using the available IOCTL calls. If the user wishes to assert the CTS line then control is transferred back to the receiver, as long as the driver has been configured to use hardware-driven hardware flow control.

2.3 MXC UART Driver Requirements

The UART driver meets the following requirements:

- Supports baud rates up to 1.5Mbps.
- Recognizes frame and parity errors only in interrupt-driven mode. The UART driver does not recognize these errors in DMA-driven mode.
- Sends and receives and appropriately handles break characters.
- Recognizes the modem control signals.
- Ignores characters with frame, parity and break errors if requested to do so.

- Implements support for software and hardware flow control (software-controlled and hardware-controlled).
- Is able to get and set the UART port information. Certain flow control count information is not available in hardware-driven hardware flow control mode.
- Implements support for Serial IrDA.
- Supports power management.
- Supports interrupt-driven and DMA-driven data transfer.

2.4 MXC UART Driver Source Code Structure

Table 2-1 lists the source files associated with the UART driver that are found in the following directory:

`drivers/serial.`

Table 2-1. UART Source And Header File List

File	Description
mxc_uart.c	MXC UART low level driver
serial_core.c	Core UART driver that is included as part of standard Linux
mac_uart_reg.h	MXC UART file for the register values

Table 2-2 lists the header files associated with the UART driver that are found in the following directory:

`include/asm-arm/arch-mxc.`

Table 2-2. UART Global Header File List

File	Description
mxc_uart.h	MXC UART header that contains UART configuration data structure definitions
board-xxxx.h	Holds some MXC UART board specific configuration options

Table 2-3 lists the source files associated with the UART driver that are found in the following directory:

`arch/asm-arm/mach-xxxx.`

Table 2-3. UART Global Header File List

File	Description
serial.c	Contains UART configuration data and calls to register the device with the platform bus

2.5 MXC UART Driver Configuration

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

2.5.1 Linux Menu Configuration Options

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

1. CONFIG_SERIAL_MXC—This configuration option is used for the MXC UART driver for the MXC UART ports. In `menuconfig`, this option is found under Characters->Serial. By default, this option is Y for all architectures.
2. CONFIG_SERIAL_MXC_CONSOLE—This configuration option chooses the MXC Internal UART to bring up the system console. This option is dependent on the "CONFIG_SERIAL_MXC" option. In the `menuconfig` this option is found under Characters->Serial. By default, this option is N for all architectures.

2.5.2 Source Code Configuration Options

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

2.5.2.1 Chip Configuration Options

The following chip-specific configuration options are provided in `mxc_uart.h`:

1. Number of UART Ports (`UART_NR`)—Number of UART ports in the platform.
2. UART Interrupts Muxed (`UARTx_MUX_INTS`)—Specifies which set of interrupts is integrated with the ARM core, the muxed ANDed interrupt lines or the individual interrupts from the UART port.
3. UART TX Interrupt / UART Muxed Interrupt Number (`UARTx_INT1`)—Specifies the transmit interrupt number, or the interrupt number of the ANDed interrupt in case the interrupts are muxed.
4. UART RX interrupt (`UARTx_INT2`)—Specifies the receiver interrupt number. In case the interrupts are muxed, set this option to a value of -1.
5. UART MINT Interrupt (`UARTx_INT3`)—Specifies the master interrupt number. If the interrupts are muxed, set this option to a value of -1.
6. UART HW Flow control (`UARTx_HW_FLOW`—Allows the user to choose either an interrupt-driven software-controlled hardware flow, or a hardware-driven hardware-controlled flow. When DMA is enabled, this option should always be specified.
7. UART CTS Threshold (`UARTx_UCR4_CTSTL`—Specifies the threshold at which the CTS pin is de-asserted by the receiver.
8. UART DMA Enable/Disable (`UARTx_DMA_ENABLE`)—Specifies whether to use DMA-driven data transfer. Setting this option to 1 enables DMA-driven data transfer.
9. UART DMA RX Buffer size (`UARTx_DMA_RXBUFSIZE`)—Specifies the size of the DMA receive buffer. The minimum size is 512 and the size should be a multiple of 256.
10. UART RX Threshold (`UARTx_UFCR_RXTL`)—Specifies the trigger level of the UART receive FIFO. This option controls the threshold at which a receive interrupt is generated. Set this option to any value between 0 and 32.
11. UART TX Threshold (`UARTx_UFCR_TXTL`)—Specifies the trigger level of the UART transmit FIFO. This option controls the threshold at which a transmit interrupt is generated. Set this option to any value between 0 and 32

12. UART Shared Peripheral (UARTx_SHARED_PERI)—Specifies whether the UART is a shared peripheral. The value should be set to the UART shared peripheral number, or to -1 if the UART is not a shared peripheral.

The x in UARTx denotes the individual UART number. The default configuration for each individual UART number is shown in Table 2-5.

2.5.2.2 Board Configuration Options

The following board-specific configuration options for the driver can be set within board.h:

1. UART Mode (UARTx_MODE)—Specifies whether the UART is configured to be in DTE or DCE mode.
2. UART IR Mode (UARTx_IR)—Specifies whether the UART port is to be used for IrDA.
3. UART Enable / Disable (UARTx_ENABLED)—Enable or disable a particular UART port. If disabled, the UART will not be registered in the file system and the user will not be able to access it.
4. MAX_UART_BAUDRATE—Specify the maximum baud rate that you wish to support on your board. Any value up to 1500000 can be specified.

The x in UARTx denotes the individual UART number. The default configuration for each individual UART number is shown in Table 2-6.

2.6 MXC UART Driver Programming Interface

The MXC UART Driver implements all the methods required by the Linux serial API to interface with the MXC UART port. The driver implements and provides a set of control methods to the Linux core UART driver.

2.7 MXC UART Driver Interrupt Requirements

The MXC UART Driver interface generates many kinds of interrupts. The highest interrupt rate is associated with the transmit and receive interrupt. The system requirements are shown in Table 2-4.

Table 2-4. UART Interrupt Requirements

Parameter	Equation	Typical	Worst-Case
Rate	(BaudRate/(10))*(1/Rxtl + 1/(32-Txtl))	5952/sec	300000/sec
Latency	320/BaudRate	5.6ms	213.33us

The baud rate is set in the mxcuart_set_termios function. The typical values are based on a baud rate of 57600 with a receiver trigger level (Rxtl) of 1 and a transmitter trigger level (Txtl) of 2. The worst-case is based on a baud rate of 1.5 Mbps (max supported by the UART interface) with an Rxtl of 1 and a Txtl of 31. There is also an undetermined number of handshaking interrupts that are generated but the rates should be an order of magnitude lower.

2.8 MXC UART Driver Unit Test

The the first four steps following are the unit tests performed on development hardware boards. Following that is the IrDA unit test.

1. Redirect the output of a shell command to another UART. Ensure both ends are using the same baud rate and other port settings.

   ```
   stty -F /dev/ttymxc/1 115200 -- set the baud rate of UART 2 to 115200
   ls >/dev/ttymxc/1 -- the output should be printed on UART2's window
   ```

2. Start a shell in the background and redirect its standard input, standard output and standard error to a different UART port

   ```
   sh </dev/ttymxc/2 >/dev/ttymxc/2 2>/dev/ttymxc/2
   ```

3. A user space test application was written to test the loop-back IOCTL.
4. Type the following command on the linux console

   ```
   cat /dev/ttymxc/0
   ```

   On the board, connect UART 1 to a serial port on your PC. Open a Hyperterminal window that is connected to this serial port. Type characters in the Hyperterminal window. These characters should be printed by the cat application in the console window. Text files should also be transferred (Transfer->Send Text File). The contents of this file should be received by the cat received through UART 1. Note that this test will not pass for serial ports which have the UART DMA Enable/Disable (`UARTx_DMA_ENABLE`) configuration option set to 1 as the port will be operating in a raw mode where the character input is not being parsed. This is expected behavior.
5. Test Serial Infrared on the boards.

   This test example needs two boards. Before trying IrDA make sure to align the IrPorts to be facing each other at a distance of about an inch.

   Below are the steps to test Serial IrDA using IrLAN on the board.

   — For i.MX31 change `/dev/ttymxc/3` to `/dev/ttymxc/1.`

   On one board:

   ```
   stty -F /dev/ttymxc/0 -echo
   modprobe irtty-sir
   insmod /lib/modules/2.6.18.1/kernel/net/irda/irlan/irlan.ko access=2
   ifconfig irlan0 10.0.0.1 netmask 255.255.255.0 broadcast 10.0.0.255
   irattach /dev/ttymxc/0 -s
   ping 10.0.0.2
   ```

   On another board:

   ```
   stty -F /dev/ttymxc/0 -echo
   modprobe irtty-sir
   insmod /lib/modules/2.6.18.1/kernel/net/irda/irlan/irlan.ko access=2
   ifconfig irlan0 10.0.0.2 netmask 255.255.255.0 broadcast 10.0.0.255
   irattach /dev/ttymxc/0 -s
   telnet 10.0.0.1
   ```

2.9 Device-Specific Information

2.9.1 UART Ports

i.MX31 - There are five UART modules in the i.MX31. The MXC UART ports can be accessed through the device files /dev/ttymxc/0 through /dev/ttymxc/4.

Table 2-5. Default UART Chip Configuration.

UART Option	i.MX31	i.MX27
UART_NR	5	6
UART1-3_MUX_INTS	INTS_MUXED	INTS_MUXED
UART4_MUX_INTS	INTS_MUXED	INTS_MUXED
UART5_MUX_INTS	INTS_MUXED	INTS_MUXED
UART6_MUX_INTS	N/A	INTS_MUXED
UART1-3_INT1	UART Interrupt	UART Interrupt
UART4_INT1	UART Interrupt	UART Interrupt
UART5_INT1	UART Interrupt	INTS_MUXED
UART6_INT1	N/A	INTS_MUXED
UART1-3_INT2	-1	-1
UART4_INT2	-1	-1
UART5_INT2	-1	-1
UART6_INT2	N/A	-1
UART1-3_INT3	-1	-1
UART4_INT3	-1	-1
UART5_INT3	-1	-1
UART6_INT3	N/A	-1
UART1_HW_FLOW	1	1
UART2_HW_FLOW	0	1
UART3_HW_FLOW	1	1
UART4_HW_FLOW	1	1
UART5_HW_FLOW	1	1
UART6_HW_FLOW	N/A	1
UART1_UCR4_CTSTL	16	16
UART2_UCR4_CTSTL	-1	16
UART3_UCR4_CTSTL	16	16
UART4_UCR4_CTSTL	16	16
UART5_UCR4_CTSTL	16	16
UART6_UCR4_CTSTL	N/A	16
UART1_DMA_ENABLE	0	0

UART Option	i.MX31	i.MX27
UART2_DMA_ENABLE	0	0
UART3_DMA_ENABLE	1	0
UART4_DMA_ENABLE	0	0
UART5_DMA_ENABLE	0	0
UART6_DMA_ENABLE	N/A	0
UART1_DMA_RXBUFS	1024	N/A
UART2_DMA_RXBUFS	512	N/A
UART3_DMA_RXBUFS	1024	N/A
UART4_DMA_RXBUFS	512	N/A
UART5_DMA_RXBUFS	512	N/A
UART6_DMA_RXBUFS	N/A	N/A
UART1_UFCR_RXTL	16	16
UART2_UFCR_RXTL	16	16
UART3_UFCR_RXTL	16	16
UART4_UFCR_RXTL	16	16
UART5_UFCR_RXTL	16	16
UART6_UFCR_RXTL	N/A	16
UART1_UFCR_TXTL	16	16
UART2_UFCR_TXTL	16	16
UART3_UFCR_TXTL	16	16
UART4_UFCR_TXTL	16	16
UART5_UFCR_TXTL	16	16
UART6_UFCR_TXTL	N/A	16
UART1_SHARED_PERI	-1	-1
UART2_SHARED_PERI	-1	-1
UART3_SHARED_PERI	SPBA_UART3	-1
UART4_SHARED_PERI	-1	-1
UART5_SHARED_PERI	-1	-1
UART6_SHARED_PERI	N/A	-1

Table 2-6. Default UART Board Configuration Options

UART Option	i.MX31	i.MX27	i.MX21
UART1-2_MODE	MODE_DCE	MODE_DCE	MODE_DCE
UART3_MODE	MODE_DCE	MODE_DTE	MODE_DTE

Table 2-6. Default UART Board Configuration Options (Continued)

UART Option	i.MX31	i.MX27	i.MX21
UART4_MODE	MODE_DTE	MODE_DTE	N/A
UART5_MODE	MODE_DTE	MODE_DTE	N/A
UART6_MODE	N/A	MODE_DTE	MODE_DTE
UART1_IR	NO_IRDA	NO_IRDA	NO_IRDA
UART2_IR	IRDA	NO_IRDA	NO_IRDA
UART3_IR	NO_IRDA	IRDA	IRDA
UART4_IR	NO_IRDA	NO_IRDA	NO_IRDA
UART5_IR	NO_IRDA	NO_IRDA	N/A
UART6_IR	N/A	NO_IRDA	N/A
UART1-3_ENABLED	1	1	1
UART4_ENABLED	0	0	1
UART5_ENABLED	1	1	N/A
UART6_ENABLED	N/A	1	N/A
MAX_UART_BAUDRATE	1500000	1500000	1500000

Chapter 3
Liquid Crystal Display Controller (LCDC) Driver

3.1 LCD Driver Overview

The MX2 Liquid Crystal Display Controller (LCDC) provides display data for external gray-scale or color LCD panels. The LCDC is capable of supporting black-and-white, gray-scale, passive-matrix color (passive color or CSTN), and active-matrix color (active color or TFT) LCD panels. The detailed hardware operation of the LCDC can be found in the processors' specification.

The LCD driver is designed under Linux frame buffer driver framework. It provides hardware access ability to support frame buffer driver.

The frame buffer device provides an abstraction for the graphics hardware. It represents the frame buffer of some video hardware and allows application software to access the graphics hardware through a well-defined interface, so the software doesn't need to know anything about the low-level (hardware registers).

The driver is enabled by selecting the framebuffer option under the graphics parameters in the kernel configuration. To supplement the Framebuffer driver, the kernel builder may also include support for fonts and a startup logo. This depends on the Virtual Terminal (VT) console for switching from serial to graphics mode.

The device is accessed through special device nodes, usually located in the `/dev` directory, i.e. `/dev/fb*`.

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

The Frame buffer driver supports different panels. Support for new panels can be added by defining new timing values for the structure `fb_videomode`. Currently supported panels are Sharp QVGA panel, NEC VGA panel as well as TV-out PAL and NTSC. The panel to be enabled during Linux booting up can be specified by appending video options into kernel command line. By default, the frame buffer driver enables the Sharp QVGA panel.

3.1.1 Hardware Operation

The framebuffer interacts with the MX2 LCDC hardware module. Refer to the LCDC section in the processors' specification for more information.

3.1.2 Software Operation

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

/dev/fb* also allows several ioctls on it, by which lots of information about the hardware can be queried and set. The color map handling works via ioctls, too. Look into <linux/fb.h> for more information on what ioctls exist and which data structures they use. Here's just a brief overview:

- You can request unchangeable information about the hardware, like name, organization of the screen memory (planes, packed pixels, etc.) and address and length of the screen memory.
- You can request and change variable information about the hardware, like visible and virtual geometry, depth, color map format, timing, and so on. If you try to change that information, the driver maybe will round up some values to meet the hardware's capabilities (or return EINVAL if that isn't possible).
- You can get and set parts of the color map. Communication is done with 16 bits per color part (red, green, blue, transparency) to support all existing hardware. The driver does all the computations needed to apply it to the hardware (round it down to less bits, maybe throw away transparency).

All this hardware abstraction makes the implementation of application programs easier and more portable. E.g. the Qt/Embedded server works completely on /dev/fb* and thus doesn't need to know, for example, how the color registers of the concrete hardware are organized. The only thing that has to be built into application programs is the screen organization (bitplanes or chunky pixels etc.), because it works on the frame buffer image data directly.

The MX2 framebuffer driver (drivers/video/mxc/mx2fb.c) interacts tightly with the generic Linux framebuffer driver (drivers/video/fbmem.c).

3.1.3 Graphic Window

Graphic window is supported by LCDC for viewfinder function in color display. The graphic window and background plane can be alpha blended. In addition, one of the pixel colors can be chosen for color keying in which the selected pixel color is made totally transparent. Memory used by graphic window can be different from the memory used by the background plane. Thus two frame buffer devices will be implemented by this driver, one for background plane and the other for graphic window.

Since graphic window is special display (For example, graphic window supports alpha blending, color keying), additional iocts are provided to support its features.

3.1.4 TV-out

The driver also provides iocts to support TV-out. The iocts implement video encoder APIs which are defined in include/linux/video_encoder.h. The TV-out can be configured by kernel configurations in graphics parameter. For more information about TV-out refer to chapter TV Encoder (TVout) Driver.

3.1.5 Architecture Diagram

The architecture diagram is shown in figure 9-1.

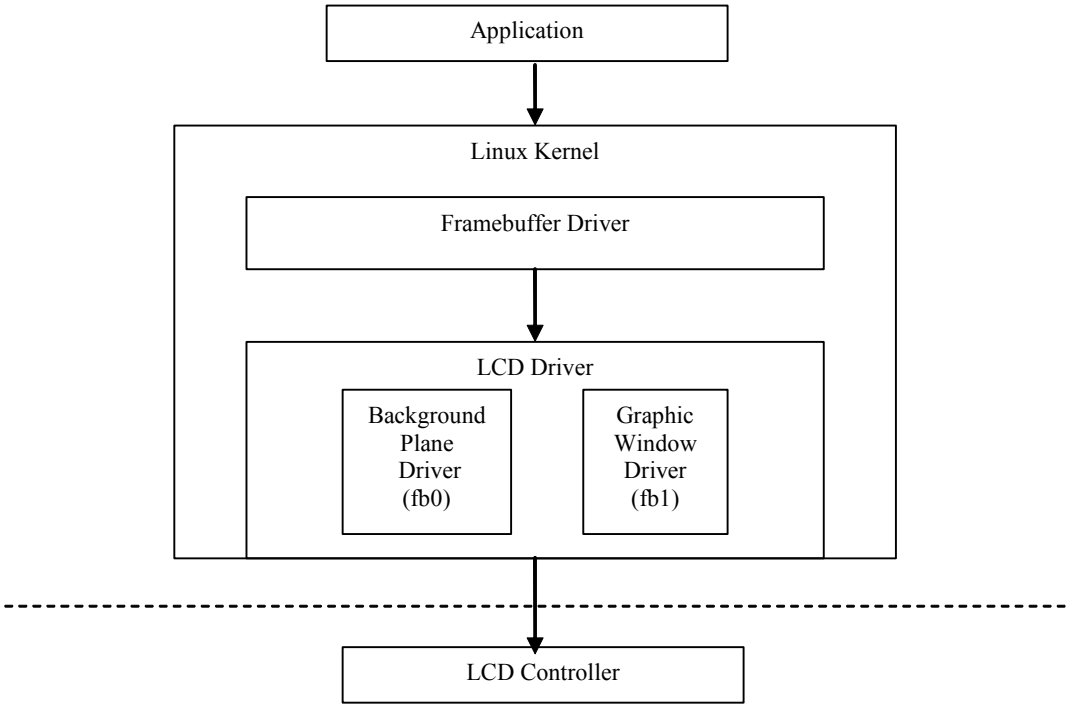

Figure 4-1. LCD Driver in the Architecture

3.2 Source Code Structure Configuration

Table 3-1 lists the source files associated with the LCD driver that are found in the directory `drivers/video/mxc/`.

Table 3-1. LCD Driver Source and Header File List

File	Description
`mx2fb.c`	Source file for MX2 LCD framebuffer driver
`mx2fb.h`	Header file for MX2 LCD framebuffer driver

3.3 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

1. CONFIG_FB_MXC: This is the configuration option for the MX2 framebuffer driver. This option is dependent on the "CONFIG_FB" option. In the `menuconfig` this option is found under "Graphics support-> MXC Framebuffer support". By default, this option is Y.
2. CONFIG_FB_MXC_SYNC_PANEL: This is the configuration option for the MX2 synchronous LCD framebuffer device. This option is dependent on "CONFIG_FB_MXC" option. In the `menuconfig` this option is found under "Graphics support-> MXC Framebuffer support->Synchronous Panel Framebuffer". By default, this option is Y.

3. CONFIG_FB_MXC_TVOUT: This is the configuration option for the MX2 TV-out framebuffer device. This option is dependent on “CONFIG_FB_MXC_SYNC_PANEL” option. In the `menuconfig` this option is found under “Graphics support-> MXC Framebuffer support->Synchronous Panel Framebuffer->TV Out Encoder”. By default, this option is Y.
4. CONFIG_FB_MXC_OVERLAY: This is the configuration option for the MX2 graphic window framebuffer device. This option is dependent on “CONFIG_FB_MXC_SYNC_PANEL” option. In the `menuconfig` this option is found under “Graphics support-> MXC Framebuffer support->Synchronous Panel Framebuffer->Framebuffer Overlay Plane”. By default, this option is N.

3.4 Unit Test

Framebuffer Tests:

- Redirecting an image directly to the background framebuffer device:

```
# cat image.bin > /dev/fb0
```

The panel to be enabled during Linux booting up can be specified by appending video options into the kernel command line.

- Enable Sharp QVGA panel during Linux booting up. The kernel command line could be like:

```
noinitrd console=ttymxc0,115200 root=/dev/nfs
nfsroot=10.193.100.211:/rootdisk/ROOTFS rw init=/linuxrc ip=dhcp
video=mxcfb:Sharp-QVGA
```

- Enable NEC VGA panel during Linux booting up. The kernel command line could be like:

```
noinitrd console=ttymxc0,115200 root=/dev/nfs
nfsroot=10.193.100.211:/rootdisk/ROOTFS rw init=/linuxrc ip=dhcp video=mxcfb:NEC-VGA
```

- Enable PAL during Linux booting up. The kernel command line could be like:

```
noinitrd console=ttymxc0,115200 root=/dev/nfs
nfsroot=10.193.100.211:/rootdisk/ROOTFS rw init=/linuxrc ip=dhcp video=mxcfb:TV-PAL
```

- Enable NTSC during Linux booting up. The kernel command line could be like:

```
noinitrd console=ttymxc0,115200 root=/dev/nfs
nfsroot=10.193.100.211:/rootdisk/ROOTFS rw init=/linuxrc ip=dhcp video=mxcfb:TV-NTSC
```

If no video options are specified in kernel command line, the Sharp QVGA panel will be enabled by default.

Chapter 4
Image Processing Unit (IPU) Drivers

The Image Processing Unit (IPU) is designed to support video and graphics processing functions in the MXC architecture, and to interface with video/still image sensors and displays.

4.1 IPU Hardware Operation

The detailed hardware operation of the IPU is detailed in the hardware documentation.

4.2 IPU Software Operation

The diagram below depicts the interaction between the different graphics/video drivers and the IPU.

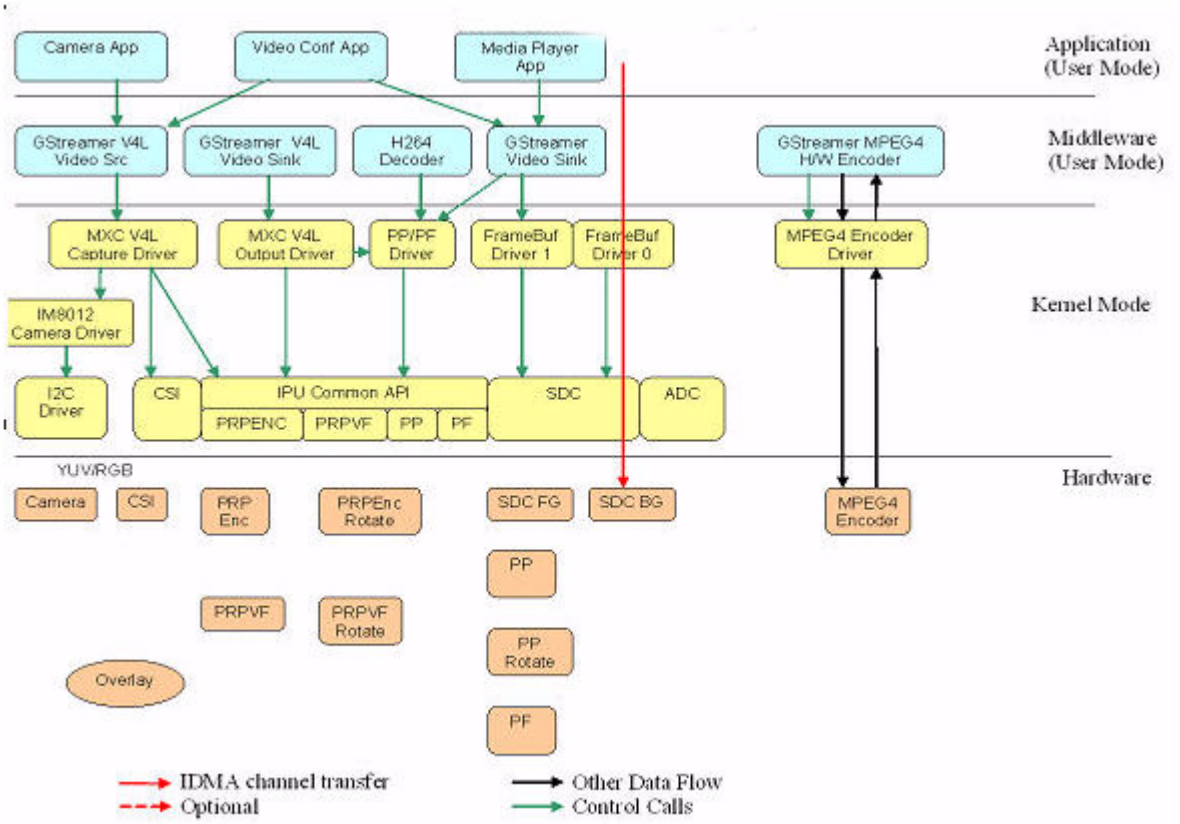

Figure 4-1. Graphics/Video Drivers Software Interaction

The IPU drivers are sub-divided as follows:

- Device drivers: They include the frame buffer driver for SDC, the frame buffer driver for Epson LCD, V4L2 capture drivers for IPU pre-processing and the V4L2 output driver for IPU post-processing. The frame buffer device drivers are found in the following directory of the Linux kernel

  `/drivers/video/mxc`

The V4L2 device drivers are found in the following directory of the Linux kernel:

```
/drivers/media/video
```

- Low-level library routines: These functions are interfaces to the IPU hardware registers. They take input from the high-level device drivers and communicate with the IPU hardware. The Low-level libraries are found in the following directory of the Linux kernel:

```
/drivers/mxc/ipu
```

4.2.1 IPU Frame Buffer Drivers Overview

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

The driver is enabled by selecting the Frame buffer option under the Graphics parameters in the kernel configuration. To supplement the Frame buffer driver, the kernel builder may also include support for fonts and a startup logo. This device depends on the Virtual Terminal (VT) console to switch from serial to graphics mode.

The device is accessed through special device nodes, usually located in the `/dev` directory, for example:

```
/dev/fb*
```

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

4.2.1.1 IPU Frame Buffer Hardware Operation

The frame buffer interacts with the IPU hardware driver module.

4.2.1.2 IPU Frame Buffer Software Operation

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

`/dev/fb*` also interacts with several ioctls, which allows you to query and set information about the hardware. The color map is also handled via ioctls. For more information on what ioctls exist and which data structures they use, see `<linux/fb.h>`. Some of the ioctl functions:

- You can request general information about the hardware, such as name, organization of the screen memory (planes, packed pixels, etc.), and address and length of the screen memory.
- You can request and change variable information about the hardware, such as visible and virtual geometry, depth, color map format, timing, and so on. If you try to change this information, the driver suggests values to meet the hardware's capabilities (the hardware returns EINVAL if that is not possible).

- You can get and set parts of the color map. Communication is 16 bits-per-pixel (values for red, green, blue, transparency) to support all existing hardware. The driver does all the calculations required in order to apply the options to the hardware (round to fewer bits, possibly discard transparency value).

All this hardware abstraction makes the implementation of application programs easier and more portable. The Qt/Embedded server works completely on `/dev/fb*` and thus is not required to know, for example, the organization of the color registers of the hardware. The only thing that must be built into the application programs is the screen organization (bitplanes or chunky pixels, and so on), because it works on the frame buffer image data directly.

The MXC frame buffer driver (`drivers/video/mxc/mxcfb.c`) interacts tightly with the generic Linux frame buffer driver (`drivers/video/fbmem.c`).

4.2.1.3 SDC Frame Buffer Driver

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

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

The Frame buffer driver supports different panels as a kernel configuration option. Support for new panels can be added by defining new values for a structure of panel settings. By default, the frame buffer driver supports the Sharp QVGA panel.

The Frame buffer interacts with the IPU Driver using custom APIs:

- Initialization of panel interface settings
- Initialization of IPU channel settings for LCD refresh
- Control of IPU SDC PWM for backlight control
- Changing the frame buffer address for double buffering support

The following features are supported:

- Support for Sharp QVGA and Sharp/NEC VGA panels
- Configurable screen resolution
- Configurable RGB 16, 24 or 32 bits per pixel frame buffer
- Configurable panel interface signal timings and polarities
- Palette/color conversion management
- Power management
- LCD Power off/on
- Backlight control

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

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

A 2nd frame buffer driver supports a second video/graphics plane.

4.2.1.4 ADC Frame Buffer Driver

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

The Frame buffer interacts with the IPU Driver using custom APIs:

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

The following features are supported:

- Support for Epson L2F60012P00 dual mode 176x220 panel.
- Configurable RGB 16, 24 or 32 bits per pixel frame buffer.
- Palette/color conversion management.
- Power management.
- LCD Power off/on.
- Backlight control.

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

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

4.2.1.5 Video for Linux 2 (V4L2) APIs

Video for Linux Two (V4L2) is a Linux standard. The API Specification is found at

`http://v4l2spec.bytesex.org/spec/`

The V4L2 capture device includes two interfaces: the capture interface and the overlay interface. The capture interface uses IPU pre-processing ENC channels to record the YCrCb video stream, while the overlay interface uses the IPU pre-processing VF channels to display the preview video to the SDC foreground panel without ARM processor interaction. The driver implements the standard V4L2 API for capture and overlay devices.

The V4L2 Output driver uses the IPU post-processing functions for video output. The driver implements the standard V4L2 API for output devices.

4.2.1.6 V4L2 Capture Device

V4L2 capture support can be selected during kernel configuration. The driver includes two layers. The top layer is the common Video for Linux driver, which contains chain buffer management, stream API and other `ioctl` interfaces. The files for this device can be found in the following file:

```
driver/media/video/mxc_v4l2_capture.c
```

The lowest layer is found in the following file:

```
driver/mxc/ipu/swlib/ipu_prp_enc.c
```

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

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

- VIDIOC_QUERYCAP
- VIDIOC_G_FMT
- VIDIOC_S_FMT
- VIDIOC_REQBUFS
- VIDIOC_QUERYBUF
- VIDIOC_QBUF
- VIDIOC_DQBUF
- VIDIOC_STREAMON
- VIDIOC_STREAMOFF
- VIDIOC_OVERLAY
- VIDIOC_G_FBUF
- VIDIOC_S_FBUF
- VIDIOC_G_CTRL
- VIDIOC_S_CTRL
- VIDIOC_CROPCAP
- VIDIOC_G_CROP
- VIDIOC_S_CROP
- VIDIOC_S_PARM
- VIDIOC_G_PARM
- VIDIOC_ENUMSTD
- VIDIOC_G_STD

- VIDIOC_S_STD
- VIDIOC_ENUMOUTPUT
- VIDIOC_G_OUTPUT
- VIDIOC_S_OUTPUT

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

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

Figure 4-2 shows a block diagram of V4L2 Capture API interaction.

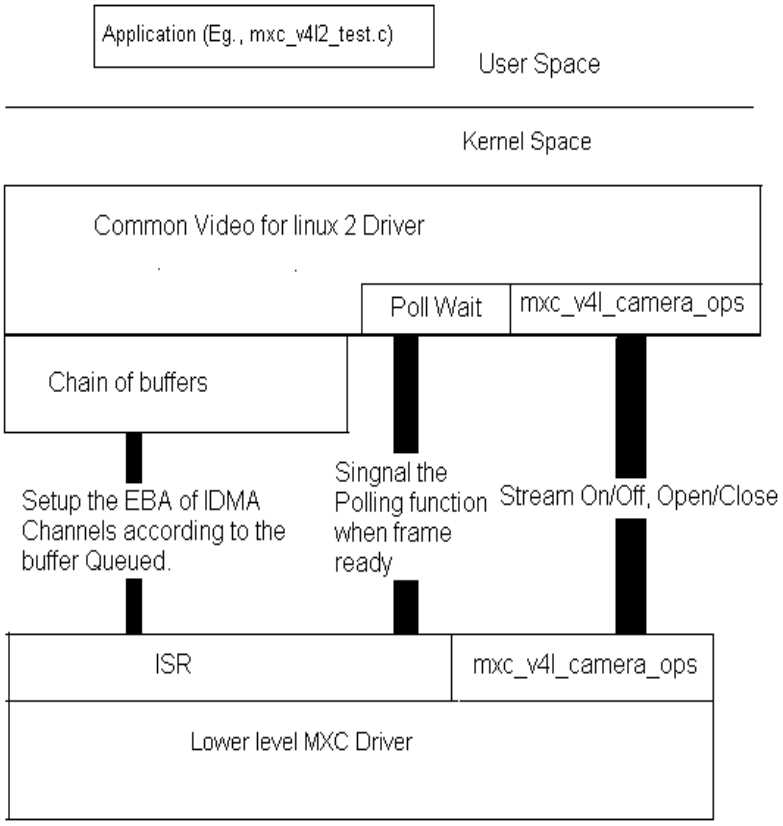

Figure 4-2. Video4Linux2 Capture API Interaction

4.2.1.7 Use of the V4L2 Capture APIs.

A sample V4L2 capture process is shown below:

1. The application sets the capture pixel format and size by `ioctl` VIDIOC_S_FMT.
2. The application sets the control information by `ioctl` VIDIOC_S_CTRL for rotation usage.
3. The application requests a buffer by using `ioctl` VIDIOC_REQBUFS. The common V4L2 driver creates a chain of buffers (currently the maximum number of frames is 3).
4. The application memory maps the buffer to its user space.
5. The application queues buffers by using the ioctl command VIDIOC_QBUF.
6. The application starts the stream by using the `ioctl` VIDIOC_STREAMON. This `ioctl` enables the IPU tasks and the IDMA channels. When the processing is completed for a frame, the driver switches to the buffer that is queued for the next frame. The driver also signals the semaphore to indicate that a buffer is ready.
7. The application takes the buffer from the queue using the `ioctl` VIDIOC_DQBUF. This `ioctl` blocks until it has been signaled by the ISR driver.
8. The application stores the buffer to a Ycrcb file.
9. The application replaces the buffer in the queue of the V4L2 driver by executing VIDIOC_QBUF again.

V4L2 still image capture:

1. The application sets the capture pixel format and size by executing the `ioctl` VIDIOC_S_FMT.
2. The application reads one frame still image with YUV422.

V4L2 overlay support use case:

1. Application set the overlay window by `ioctl` VIDIOC_S_FMT.
2. Application turn on overlay task by `ioctl` VIDIOC_OVERLAY.
3. Application turn off overlay task by `ioctl` VIDIOC_OVERLAY.

4.2.1.8 V4L2 Output Device

V4L2 output device support can be selected during kernel configuration. The driver can be found at:

```
driver/media/mxc/mxc_v4l2_output.c
```

The V4L2 features supported by the driver are:

- Direct output to the SDC foreground overlay plane (no ARM processor intervention, and synchronized to LCD refresh).
- Support for color keying alpha blending of the frame buffer and overlay planes.
- Support for linking post-processing resize and CSC, rotation, and display IPU channels for no ARM processing of intermediate steps.
- Streaming (queued) input buffer.
- Double buffering of overlay and intermediate (rotation) buffers.

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

These features are supported using custom APIs:

- Output to user buffer instead of overlay display.
- Programmable rotation.

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

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

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

- 0—Normal operation
- 1—Vertical flip
- 2—Horizontal flip
- 3—Horizontal and vertical flip
- 4—90 degree rotation
- 5—90 degree rotation and vertical flip
- 6—90 degree rotation and horizontal flip
- 7—90 degree rotation with horizontal and vertical flip

4.2.1.9 Use of the V4L2 Output APIs.

A sample V4L2 capture use case that uses the V4L2 output APIs is:

1. The application sets the capture pixel format and size using `ioctl` VIDIOC_S_FMT.
2. The application sets the control information by using `ioctl` VIDIOC_S_CTRL, for rotation.
3. The application requests a buffer by using `ioctl` VIDIOC_REQBUFS. The common V4L2 driver creates a chain of buffers (currently the maximum number of frames is 3).
4. The application memory maps the buffer to its user space.
5. The application executes the `ioctl` VIDIOC_DQBUF.
6. The application passes the data that requires post-processing to the buffer.
7. The application queues the buffer by using the ioctl command VIDIOC_QBUF.
8. The application starts the stream by executing `ioctl` VIDIOC_STREAMON.

4.2.2 MPEG4/H.264 Post Filter Driver

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

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

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

4.3 IPU Source Code Structure Configuration

Table 4-1 lists the source files associated with the IPU, Sensor, V4L2 and Panel drivers. These files are found in the following directories:

```
drivers/mxc/ipu
drivers/video/mxc
```

Different ioctls for both Display panels will be provided in future releases.

Table 4-1. IPU Source and Header File List

File	Description
`ipu_adc.c`	ADC configuration driver
`ipu_alloc.c`	IPU memory allocation driver
`ipu_common.c`	Configuration functions for ADC and SDC
`ipu_csi.c`	CMOS sensor interface functions
`ipu_ic.c`	IIVC library functions

Table 4-1. IPU Source and Header File List

File	Description
ipu_sdc.c	SDC configuration driver.
camera/s5k3aaex.c	Camera sensor driver for s5k3aaex sensor
camera/mt9v111.c	Camera sensor driver for MT9V111
driver/media/video/mxc/capture/ipu_prp_enc.c	Pre-processing encoder driver
driver/media/video/mxc/capture/ipu_prp_vf_adc.c	Pre-processing view finder (adc) driver.
driver/media/video/mxc/capture/ipu_prp_vf_sdc.c	Pre-processing view finder (sdc foreground) driver.
driver/media/video/mxc/capture/ipu_prp_vf_sdc_bg.c	Pre-processing view finder (sdc background) driver.
driver/media/video/mxc/capture/ipu_still.c	Pre-processing still image capture driver
pf/mxc_pf.c	Post filtering driver
driver/video/mxc/mxcfb.c	Framebuffer driver for SDC
driver/video/mxc/mxcfb_epson.c	Framebuffer driver for ADC
driver/video/mxc/mxcfb_epson_qvga.c	Framebuffer driver for ADC QVGA
driver/media/video/mxc/capture/mxc_v4l2_capture.c	V4L2 capture device driver
driver/media/video/mxc/output/mxc_v4l2_output.c	V4L2 output device driver

Table 4-2 lists the global header files associated with the IPU and Panel drivers. These files are found in the following directory:

drivers/mxc/ipu and drivers/vedio/mxc

Table 4-2. IPU Global Header File List

File	Description
ipu_alloc.h	Header file for IPU memory allocation driver
ipu.h	Header file for IPU drivers
ipu_param_mem.h	Helper functions for IPU parameter memory access
ipu_prv.h	Header file for Pre-processing drivers
ipu_regs.h	IPU register definitions
camera/mt9v111.h	Header file for MT9V111 sensor driver
camera/s5k3aaex.h	Header file for s5k3aaex sensor driver
pf/mxc_pf.h	Header file for Post filtering driver
driver/video/mxc/mxcfb.h	Header file for framebuffer driver for SDC

Table 4-2. IPU Global Header File List

File	Description
driver/video/mxc/mxcfb_epson.h	Header file for framebuffer driver for ADC
driver/media/video/mxc/capture/ipu_prp_sw.h	Header file for IPU PRP use case driver.
driver/media/video/mxc/capture/mxc_v4l2_capture.h	Header file for V4L2 capture device driver
driver/media/video/mxc/output/mxc_v4l2_output.h	Header file for V4L2 output device driver

4.4 IPU Linux Menu Configuration Options

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

- CONFIG_MXC_IPU—Includes support for the Image Processing Unit. In menuconfig, this option is found under 'MXC support drivers->MXC IPU->MXC Image Processing Unit'. By default, this option is Y for all architectures.
- CONFIG_MXC_CAMERA_MICRON_111—Option for both the Micron mt9v111 sensor driver and the use case driver. This option is dependent on the MXC_IPU option. In menuconfig, this option is found under 'MXC support drivers->MXC IPU Camera configuration->IPU Camera support mt9v111'. Only one sensor should be installed at one time. By default, this option is valid for all MXC EVB.
- CONFIG_MXC_IPU_PRP_VF_SDC—Option for the IPU:

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

  use case driver for dumb sensor or

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

  for smart sensors. In menuconfig, this option is found under 'MXC support drivers->MXC IPU->MXC IPU PRP Feature support->Pre-Processor VF SDC library'. By default, this option is Y for all.
- CONFIG_MXC_IPU_PRP_VF_ADC—Option for the IPU:

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

  use case driver for the rotation use case or

  ```
  CSI->IC->ADC
  ```

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

  ```
  MXC support drivers->MXC IPU->MXC IPU Feature support->Pre-Processor ADC VF library'
  ```

  By default, this option is Y for all.
- CONFIG_MXC_IPU_PRP_ENC—Option for the IPU:

  ```
  CSI -> IC -> MEM MEM -> IC (PRP ENC) -> MEM
  ```
- use case driver for dumb sensors or

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

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

```
MXC support drivers->MXC IPU->MXC IPU Feature support->Pre-processor Encoder library
```

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

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

```
MXC support drivers->MXC IPU->MXC MPEG4/H.264 Post Filter Driver.
```

By default, this option is N for all.

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

```
VIDEO_DEV && MXC_IPU && MXC_IPU_PRP_VF_SDC && MXC_IPU_PRP_ENC
```

option. In `menuconfig`, this option is found under:

```
Multimedia devices->Video For Linux->MXC Video for Linux Camera"
```

By default, this option is Y for all.

- CONFIG_VIDEO_MXC_OUTPUT—This is configuration option for V4L2 output Driver. This option is dependent on "VIDEO_DEV && MXC_IPU" option. In `menuconfig`, this option is found under 'Multimedia devices->Video For Linux->MXC Video for Linux Output". By default, this option is Y for all.
- CONFIG_FB—This is the configuration option to include frame buffer support in the Linux kernel. In `menuconfig`, this option is found under:

```
Graphics->Support for Frame buffer devices'
```

By default, this option is Y for all architectures.

- CONFIG_FB_MXC—This is the configuration option for the MXC Frame buffer driver. This option is dependent on the "CONFIG_FB" option. In `menuconfig`, this option is found under:

```
'Graphics->MXC Frame buffer support'
```

By default, this option is Y for all architectures.

- CONFIG_FB_MXC_16BPP—This is the configuration option that chooses 16 bits-per-pixel as the screen depth. This option is dependent on the "CONFIG_FB_MXC" option. In `menuconfig`, this option is found under

```
Graphics->Frame buffer Bit Depth
```

By default, this option is N for all architectures.

- CONFIG_FB_MXC_24BPP—This is the configuration option to choose 24 bits-per-pixel as the screen depth. This option is dependent on the "CONFIG_FB_MXC" option. In `menuconfig`, this option is found under:

```
Graphics->Frame buffer Bit Depth
```

By default, this option is Y for all architectures.

- CONFIG_FB_MXC_32BPP—This is the configuration option that chooses 32 bits-per-pixel as the screen depth. This option is dependent on the "CONFIG_FB_MXC" option. In `menuconfig`, this option is found under

```
Graphics->Frame buffer Bit Depth
```

By default, this option is N for all architectures.

4.5 IPU Programming Interface

4.6 Unit Test

4.6.1 Framebuffer Tests:

The IPU has been tested with the SDC and ADC framebuffer.

1. Execute a GUI application (Qtopia with Qt/Embedded). The output is sent the framebuffer device `/dev/fb0`
2. Write a simple test application to blit some pixels to the device by allocating memory space through `mmap`
3. Redirect an image directly to the framebuffer device:

```
# cat image.bin > /dev/fb0
```

4.6.2 Video4Linux API test

There is a test application called mxc_v4l2_test.c under the Linux2.6/misc. directory. If you do a make menuconfig, you can select the test by Test programs->Available tests->MXC_V4l2_test, make sure you configure the right LCD type and byte per pixel information, you should change those setting according with the kernel configuration under Graphics support->MXC Framebuffer Support->Framebuffer Bit depth and MXC Support drivers->MXC IPU->MXC IPU Display Configuration.

Before running the v4l2 capture test application, you should be able see that the `/dev/v4l/video0` been created.

```
# mxc_v4l2_capture.out -w 352 -h 288 -r 0 -c 50 -fr 30 test.yuv
Capture the camera and store the 50 frames of YUV420 (QQVGA size)to a file called
test.yuv and set the frame rate to 30 fps. Look at mxc_v4l2_capture.out -help to see
usage.
# mxc_v4l2_still.out -w 640 -h 480 -f YUV422P
Do a still image capture of the camera and store the YUV422P (VGA size)to a file
called ./still.yuv.
# mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 160 -oh 160 -ot 20 -ol 20 -r
0 -t 50 -d 0 -fg -fr 30
Direct preview the camera to SDC foreground, and set frame rate to 30 fps, window of
interest is 640 X 480 with starting offset(0,0), the preview size is 160 X 160 with
starting offset (20,20). mxc_v4l2_overlay.out -help to see the usage.
# mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 160 -oh 160 -ot 20 -ol 20 -r
4 -t 50 -d 0-fr 30
Direct preview(90 degree rotation) the camera to SDC background, and set frame rate
to 30 fps.
# mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 120 -oh 120 -ot 40 -ol 40 -r
0 -t 50 -d 1 -fg -fr 30
Direct preview the camera to ADC Epson panel, and set frame rate to 30 fps.
# mxc_v4l2_overlay.out -iw 640 -ih 480 -it 0 -il 0 -ow 120 -oh 120 -ot 40 -ol 40 -r
4 -t 50 -d 1 -fg -fr 30
Direct preview(90 degree rotation) the camera to ADC Epson, and set frame rate to 30
fps.
# mxc_v4l2_output.out -iw 352 -ih 288 -ow 176 -oh 144 -r 0 -fr 20 test.yuv
Read the YUV420 stream file on test.yuv created by mxc_v4l2_capture test. do color
space conversion and resize, then display on the framebuffer.
```

NOTE

The PRP channels require the stride line must be multiple of 8 pixels, for example with no rotation, width need to be 8 pixel aligned; and with 90 rotation, the height need to be 8 bytes aligned. Downsize can not be exceed 8:1, for example a VGA sensor,

the smallest downsize will be 80X60.

Chapter 5
PMIC Protocol Driver

This chapter describes the Power Management Integrated Circuit (PMIC) protocol device driver for Linux that provides the low-level read/write access to the PMIC's hardware control registers.

One key objective of the PMIC protocol driver and the other PMIC-related drivers is to provide a complete API interface to all supported PMIC chips, despite differences in hardware design and implementation. This is necessary in order to minimize the effort to design, implement, test, and support PMIC device drivers.

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

This chapter describes the requirements, design, implementation, and client API that is provided for accessing PMIC hardware. Additional information about the PMIC device driver APIs, especially programming-related details, can also be found in the Doxygen-generated HTML documentation that is provided with the Linux BSP distribution. As shown in Figure 5-1, the PMIC protocol driver handles all low-level communications between many other Linux device drivers and the PMIC hardware. The PMIC protocol driver uses one of the available SPI buses to communicate with the PMIC chip.

NOTE

The PMIC protocol driver is intended only for use with the MCU core and the Linux OS. An equivalent PMIC driver for the DSP core within a dual core platform is beyond the scope of this document.

Full technical documentation for the PMIC chips can be found in the respective Detailed Technical Specification (DTS) documents.

5.1 Key PMIC Features and Capabilities

The Power Management Integrated Circuit (PMIC) protocol typically provides hardware to support the following functions for Freescale's MXC-based platforms:

- Audio playback and recording
- Power supply control, battery charging, and power management support
- Analog-to-digital conversion (including touchpanel support)
- External RS-232 and USB OTG connectivity
- LED and LCD backlight control

- Real-time clock (RTC) support
- Event notification through the use of hardware interrupts

These functions are all selected and configured through the PMIC control registers, which are accessible through two separate SPI interfaces. The Primary SPI interface initially has full read/write access to the PMIC control registers, while the Secondary SPI interface initially has only read access but can be granted selective read/write access. When used with dual core platforms, the PMIC can be controlled by both the MCU and the DSP through their respective SPI interfaces. Depending upon the actual system requirements, either the MCU or the DSP can be designated as the Primary Processor and connected to the PMIC through the Primary SPI bus. The other processor would then be designated as the Secondary Processor and be connected to the PMIC through the Secondary SPI bus. For single core platforms, only the Primary SPI interface to the PMIC is typically used.

Figure 5-1 shows the main functional blocks provided by the PMIC.

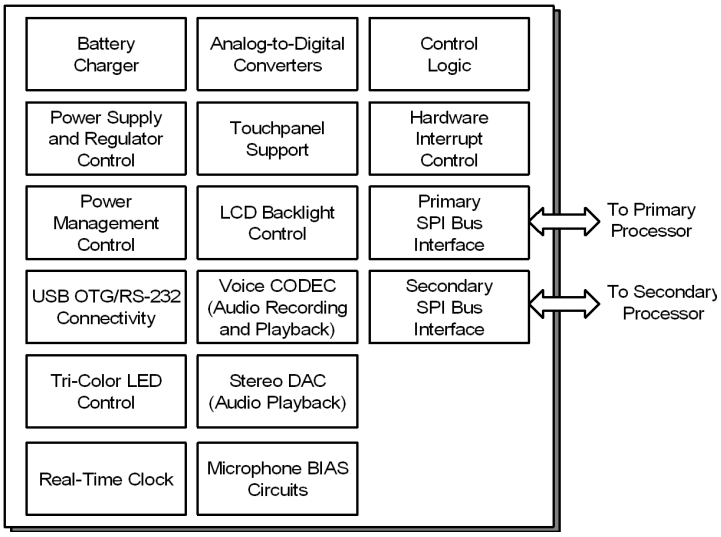

Figure 5-1. PMIC Block Diagram

Note that not all of the functions can be used at the same time because of hardware constraints. For example, some of the I/O pins are shared between the USB OTG and RS-232 transceivers. Therefore, USB OTG and RS-232 connectivity cannot be used at the same time, although it is certainly possible to switch between the two modes. For the sake of simplicity, only the SPI bus interfaces are shown in Figure 5-2 and all of the other PMIC data buses and external I/O connections have been omitted.

Table 5-1 provides a brief description of the PMIC functional blocks for which Linux device drivers have already been implemented. Additional information about the device drivers for each of these PMIC functional blocks may be found in the following chapters.

Table 5-1. A Brief Summary of All Available PMIC Client Device Drivers

PMIC Device Driver	Functions
Power Management Driver	• Battery charger interface for wall charging and USB charging. • Regulators with internal and external pass devices. • Power up and power down control.
Analog-to-Digital Conversion (ADC) Driver	• 10-bit ADC for battery monitoring and other readout functions. • Touch screen interface.
Audio Driver	• Audio input amplifier selection and gain control. • Microphone bias circuit control. • Audio output amplifier selection and gain control. • Audio output hardware mixing and mono adder control. • 13-bit Voice CODEC supporting playback and recording at either 8 kHz or 16 kHz sampling rates. • 13-bit Stereo DAC supporting playback at multiple sample rates.
RTC Driver	• Real-time clock support (MC13783 PMIC only).
Backlight and LED Driver	• Manages the LCD backlight level and each of the Red, Green, and Blue LEDs.
Connectivity Driver	• USB OTG and RS-232 transceiver control. • USB OTG device insert/removal detection and notification. • USB OTG connection negotiation and voltage level control.
Battery Driver	• Configures the battery control/monitoring interface.

5.1.1 PMIC Register Access and Arbitration

The main purpose of the PMIC protocol driver is to provide the necessary read/write access to the PMIC control registers using the SPI bus interfaces in order to support all of the higher-level PMIC client drivers that are shown in Figure 5-1 on page 5-2 and which were briefly described in Table 5-1 above. There are two possible techniques for accessing the PMIC control registers: Exclusive Sharing or Logic Sharing.

For each PMIC control register, choose either of these two techniques.

1. **Exclusive Sharing**—One processor has exclusive control. The processor connected to the primary SPI bus interface determines which processor has control by setting the appropriate arbitration control bits. Only the designated processor can modify the register. By default, only the primary SPI has read/write access to the PMIC control registers, while the secondary SPI has only read access. However, some of the PMIC control registers and settings cannot be accessed at all from the secondary SPI, regardless of the arbitration bit settings. See the appropriate PMIC Detailed Technical Specifications (DTS) document for complete information about primary versus secondary SPI bus access to the control registers.
2. **Logic Sharing**—Control is determined by analyzing logical expressions. Values of both the primary and secondary control register settings are logically ANDed or ORed to create the final resource control value. Logic Sharing of a resource, through either a single bit or a multi-bit vector, can also be selected by setting the appropriate arbitration bit values through the primary SPI interface.

Immediately following a Power up or Reset event, the processor that is connected to the Primary SPI interface can modify the PMIC control registers to configure the desired access mode for control registers.

The specific registers that need to be updated and the appropriate arbitration bit values can be found in the Detailed Technical Specifications document for the PMIC.

PMIC register access and arbitration settings are not issues on platforms where Linux is running on the primary processor. However, where Linux is running on the secondary processor (on a dual-core platform), additional steps must be taken to provide the required level of access to the PMIC registers from the secondary SPI interface. The following options may be used to resolve this issue:

- Modify the platform or the PMIC hardware to swap the primary and secondary processor connections.
- Implement additional software on the primary processor (which is not running Linux in this case) to grant the secondary processor the required access rights to the PMIC control registers.

The second option is typically the preferred solution. However, in situations where additional software development on the primary processor (usually DSP core) is not practical in the short-term, then a possible interim solution is to modify the PMIC hardware so that both the primary and secondary SPI interfaces are connected to a secondary processor running Linux (for example, by connecting both CSPI1 and CSPI2 from the ARM core to the PMIC). A single function can be implemented that will be called during the Linux boot process and use the primary SPI interface to reconfigure the PMIC arbitration bits as required. This allows the rest of the Linux system to operate properly using only the secondary SPI interface, after the boot process has been completed. Note that this is strictly an interim solution for getting Linux to run properly on the secondary processor with full PMIC functionality. This hardware change to the PMIC SPI interfaces completely disconnects the DSP core from the PMIC and, therefore, cannot be used as a true solution to the arbitration problem. Ultimately, implementing the appropriate software on the primary processor to reconfigure the PMIC arbitration settings is the only appropriate solution when Linux is running on the secondary processor.

5.1.2 Interrupt Notification

Events are reported to either the primary or secondary processor through the use of a PMIC-generated hardware interrupt. A single interrupt signal can indicate one or more events. The PMIC protocol driver first receives the interrupt signal and then checks the PMIC's interrupt status register to determine exactly which events are being signalled. Finally any client-registered callback functions are called to complete the handling of the event. If no callback functions are currently registered, then the event is ignored.

Table 5-2 lists all events that the MC13783 PMIC protocol driver supports.

Table 5-2. MC13783 PMIC Hardware Interrupt Events

Event	Description
ADC has finished requested conversions	ON1B event
Touchscreen wake up	ON2B event
ADC reading above high limit	ON3B event
ADC reading below low limit	System reset
Charger attach	SW1A low setting stabilized

Table 5-2. MC13783 PMIC Hardware Interrupt Events

Event	Description
Charger over voltage detection	SW1A high setting stabilized
Charger path reverse current	SW1B low setting stabilized
Charger path short circuit	SW1B high setting stabilized
BP regulator in regulation	SW2A low setting stabilized
Dual path selection	SW2A high setting stabilized
End of trickle charge	SW2B low setting stabilized
End of life / low battery detect	SW2B high setting stabilized
USB 4V detect	Thermal warning
USB 2V detect	Power cut event
USB 1V detect	Warm start event
Microphone bias 2 detect	Memory hold event
Headset attach	Clock source change
Stereo headset detect	Semaphore cleared
Thermal shutdown Asp	ICTEST state
Short circuit on `Ahs` outputs	CHRGMOD state
1 Hz time tick	USBMOD state
Time of day alarm	BOOT state
Wake up event	SW1A and SW1B joined

5.2 Driver Requirements

The PMIC protocol driver module (also called the “core” driver in the Linux source tree) is responsible for providing two types of services for all of the PMIC client driver components:

1. Control Services
2. Event Notification Services

The PMIC protocol driver may be built as a Linux loadable kernel module and manually loaded following system boot. However, the protocol driver is typically configured to be built into the Linux kernel image itself, because the PMIC card is not intended to be dynamically added or removed once the system has been powered on. Also, some of the Linux power management functions require that the PMIC protocol driver be properly loaded and fully operational.

5.2.1 Control Services

The key control services provided by the protocol device driver are:

1. The ability to configure the SPI bus driver to communicate with the PMIC.
2. The ability to read the current value of any PMIC hardware control register, by initiating the appropriate SPI bus transaction.
3. The ability to write new values to any PMIC hardware control register, by initiating the appropriate SPI bus transaction.
4. As the SPI bus transactions are asynchronous in nature, the PMIC protocol driver must not disable interrupts or be operating in an atomic context when making calls to the SPI driver.

Note that both the read and write capabilities may be affected by the Primary and Secondary SPI bus arbitration settings.

5.2.2 Event Notification Services

The PMIC protocol device driver must support the following event notification services:

1. Register a default interrupt handler to handle all PMIC-related hardware interrupt events.
2. Allow other PMIC client drivers to subscribe and unsubscribe to one or more PMIC events and to specify an appropriate “callback” function.
3. Call all previously registered callback functions when the corresponding PMIC event has been received.
4. The ability to properly set the PMIC interrupt event mask register to selectively control which hardware events are enabled or disabled.
5. The ability to query the PMIC interrupt status register to determine which events are being signalled by the current hardware interrupt.

5.2.3 Other Requirements

In addition to the specific services-related requirements given above, the PMIC protocol driver must also satisfy the following additional requirements:

1. Be able to properly reconfigure the PMIC arbitration settings (if required) to support the functionality that is expected by the rest of the Linux system.
2. Conform to the Linux coding standards.

5.3 Driver Software Operation

The PMIC protocol driver controls the PMIC by reading and writing the PMIC hardware control registers. Both read and write access to the PMIC hardware control registers is done via the SPI driver. The PMIC protocol driver requires the SPI driver to perform all of the following functions:

- Create the proper data packets for transmission on the SPI bus. This includes putting the proper destination address for accessing a specific PMIC control register.
- Send the data packet and verify it’s transmission status.

- Receive and decode any data packets that were sent by the PMIC.
- Return any data received from the PMIC hardware back to the PMIC protocol driver.

Figure 5-2 shows the relationship between the PMIC protocol driver and all of the other related device drivers in the system as well as the interaction between them.

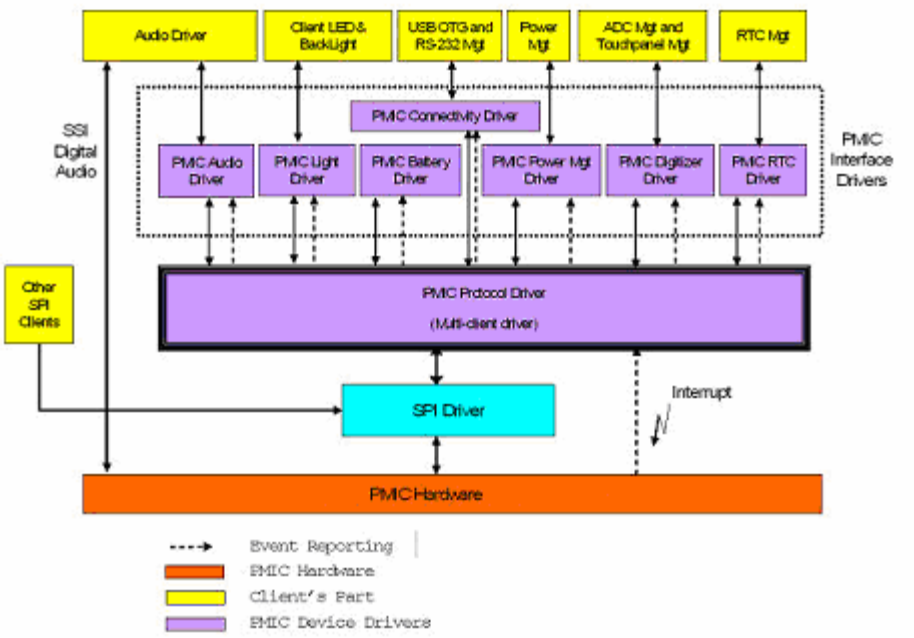

Figure 5-2. PMIC Device Driver

The hardware interrupt signal that can be generated by the PMIC is first received and handled by the PMIC protocol driver. The PMIC protocol driver will then determine exactly which events are being signalled by the PMIC by examining the PMIC's interrupt status register. Finally, all PMIC interface drivers that have previously registered for the currently active events are signalled via their respective callback functions.

5.4 Driver Architecture

Figure 5-3 shows the overall architecture and external interfaces for the PMIC protocol driver.

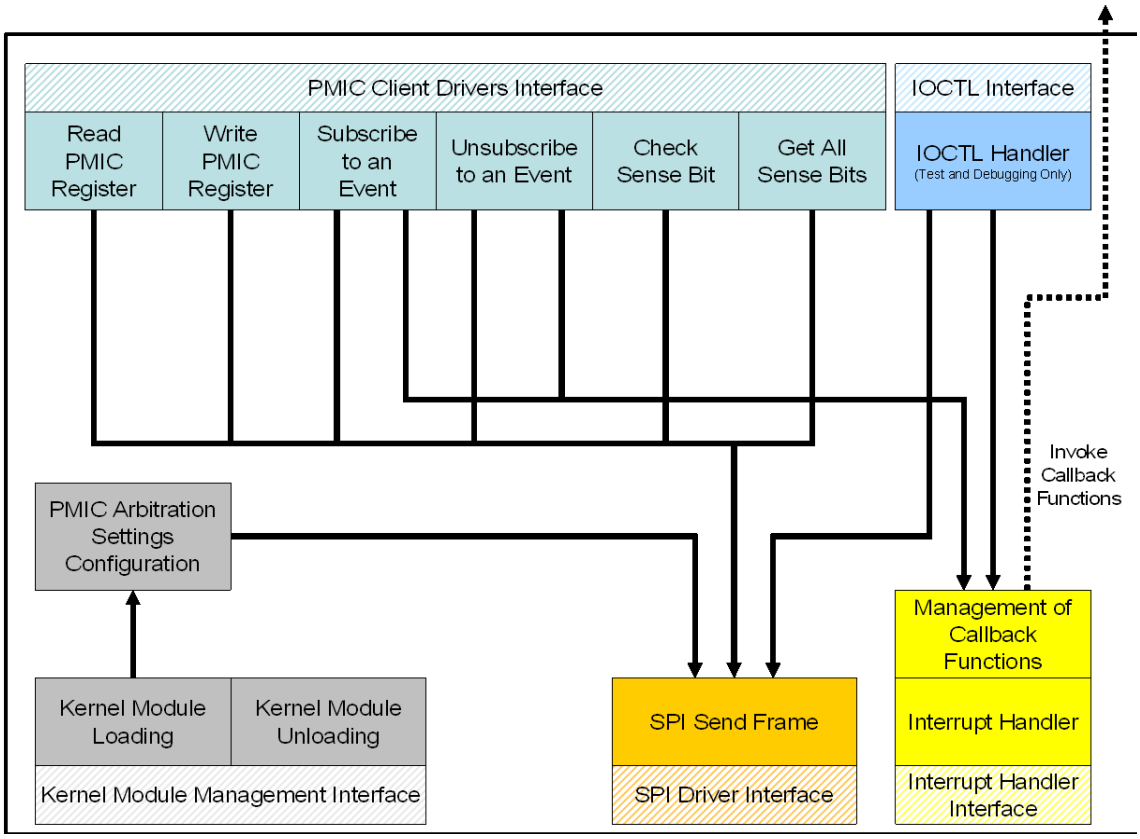

Figure 5-3. PMIC Protocol Driver Architecture and Interfaces

The key components are:

- Read/Write interfaces for the PMIC control registers, subscribing/unsubscribing to PMIC events, and checking on one or more of the PMIC sense bits.
- PMIC device interface supporting the Linux IOCTL interface, for use with `/dev/pmic` device.
- Interface for sending and receiving SPI data packets to and from the PMIC.
- API for hardware interrupt notifications and which will then invoke the appropriate event callback functions.
- API supporting the Linux kernel module, loading/unloading operations and device driver initialization requirements.
- Internal function, called only during device driver initialization, that reconfigures the arbitration bits on the PMIC, if necessary, to support proper operation from the secondary processor.

Each of these main device driver components will be described in greater detail, including any implementation-specific issues, in the following section.

5.5 Driver Implementation Details

This section describes implementation-specific details associated with the PMIC protocol driver. The device driver source files should also be consulted in order to fully understand the implementation of the PMIC protocol driver. The CSPI driver documentation and sources should also be consulted if required.

5.5.1 Driver Initialization

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

- Create either a `/dev/pmic` character device entry, depending on which version of the driver is actually being loaded, and register the new device with the kernel.
- Perform any required PMIC arbitration fixes (see Section 5.1.1, "PMIC Register Access and Arbitration)
- Initialize the PMIC registers to a known state (optionally done here; or can be done by the individual PMIC client drivers on a component-by-component basis).
- Initialize all driver-specific global variables.
- Enable the PMIC hardware interrupt line and bind it to the top half interrupt handler (see Section 5.5.4.1, "Top Half Interrupt Handler).

5.5.2 Driver Unloading

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

- Remove the `/dev/pmic` device entry and tell the kernel to deregister this device.
- Disable the PMIC hardware interrupt line to prevent any further interrupts from occurring.

5.5.3 Event Notification List

The PMIC protocol driver uses a static array of `list_head` to manage the event notification list. The subscript of the array corresponds to a specific event ID, and each array element is actually the head of a linked list. Each element of the linked list contains all the information needed to invoke a callback function. Initially the array of list_head is initialized to indicate that all of the linked lists are currently empty and that no callback functions are currently registered.

Whenever an event callback function is to be registered, a new linked list element consisting of a structure with the following fields is allocated:

- A pointer to a callback function that takes a single (void *) argument and which does not return anything.
- A (void *) field that holds that argument that is to be used when invoking the callback function.
- A pointer to the next callback data structure for the same event.

This structure element is then initialized with the proper values and added to the appropriate linked list in the array of event notification lists.

When a PMIC-generated hardware interrupt arrives, the interrupt handler will start by examining the PMIC's interrupt status register to determine the currently active events. The corresponding elements in the array of event notification lists are then examined to see if any callback functions have been registered and, if so, they are all invoked in the same order that they were registered.

Deregistering a callback function simply involves removing the callback data structure by adjusting the linked list pointers and then deallocating the memory for the callback data structure.

The only important thing to keep in mind with the handling of the event notification list is that it must always be kept in a consistent state and that any possible race conditions must be prevented. This basically means that all of the following scenarios must be properly handled:

1. Registration and deregistration of callback functions must always be performed in a critical section, so that the array of pointers and the associated linked lists are always kept in a consistent state. This also avoids any possible memory leaks during allocation and deallocation of the memory required for the linked list elements. As part of the callback registration and deregistration process, calls to the SPI driver must be made, in order to update the PMIC's interrupt mask register. When calls are made to the SPI driver, interrupts must be enabled, which eliminates atomic contexts. Therefore, the critical section must be implemented using only a mutex and not a spinlock.
2. Callback function registration, deregistration, and the interrupt handler must all use a critical section when accessing the array of pointers and the linked list of callback data structures. Since the interrupt handler is involved here, spinlocks must be used to implement the critical section. Fortunately, the interrupt handler itself does not require making any calls to the SPI driver, so running in an atomic context does not cause any problems.

What these two requirements ultimately mean is that a mutex must be used to guard against race conditions between callback registration and deregistration operations. Furthermore, within the mutex critical section, a spinlock must be used to guard against race conditions when the contents of anything in the event notification list (either the array of pointers or the associated linked lists) are used or modified. However, the spinlock can be released as soon as modifying the event notification list is no longer required, and the mutex can be used to perform any operation that is not directly associated with or impacted by the interrupt handler.

5.5.4 Interrupt Handler

The PMIC interrupt handler is divided into two parts. The "top half" is called directly by the Linux kernel when the hardware interrupt is first raised, and all interrupts are disabled while the "top half" interrupt handler is executing. The "top half" acknowledges and handles the interrupt.

However, if handling the interrupt also requires significant processing or other, possibly time consuming operations, then all such operations should be deferred to a separate "lower half" interrupt handler that can be executed at a lower priority and with hardware interrupts reenabled.

5.5.4.1 Top Half Interrupt Handler

The top half interrupt handler in the PMIC protocol driver performs the following operations:

1. Acknowledge and clear the hardware interrupt condition.
2. Schedule a work queue task to complete the handling of the interrupt event.

Note that PMIC-related interrupts typically don't have any hard real-time requirements. Therefore, it is perfectly acceptable to defer much of the interrupt handling to a separate work queue task.

5.5.4.2 The Lower Half Interrupt Handler

The lower half interrupt handler for the PMIC protocol driver is implemented as a work queue. Scheduling is done by the top half whenever a hardware interrupt is received. The lower half handler does the work that is required to handle the PMIC interrupt. The exact steps are as follows:

1. Read the current value of the PMIC's interrupt status register to determine the list of currently active events.
2. Clear the PMIC interrupts that will be handled by the PMIC device drivers. Note that if no callback functions have been registered yet for an event, the PMIC protocol driver will just silently ignore the event.
3. Invoke any callback functions that have been registered for the currently active events.

As already noted in the previous section, the interrupt handler must use a spinlock to implement a critical section around any code that accesses the event notification list. This is needed to ensure that the event notification list remains in a consistent state while the interrupt handler is running.

5.5.5 Event Handlers

Event handlers are callback functions that a device driver may use in order to be notified by the PMIC interrupt handler that a particular event has just occurred. The device driver that is registering a callback function may also specify a single (void *) argument that will be returned later when the callback is invoked. This argument can be used to identify a specific instance of the callback function or be used to access any context-specific data. No return value is expected from the event handler.

5.5.6 Register Access

The PMIC protocol driver exports APIs that allow other device drivers to read and write to PMIC control registers. The PMIC control registers are accessed using one of the two available SPI interfaces. Either the Primary or Secondary SPI interface is used, depending upon the specific design for the hardware connections between the platform and the PMIC. As previously described in Section 5.1.1, "PMIC Register Access and Arbitration, there are significant operational differences between register access by the primary and secondary SPI bus interfaces. However, the PMIC protocol driver is implemented in such a way that all these differences are taken care within the PMIC protocol driver. Externally, the PMIC protocol driver simply provides APIs to read and write to the PMIC control registers.

A separate IOCTL-based interface using the `/dev/pmic` device to read and write to the PMIC control registers has also been implemented as a separate test module. However, this interface is intended only for debugging and testing, and is not intended for general use.

5.6 Driver Source Code Structure

The source files for the PMIC protocol driver are contained in the drivers directory

```
linux\drivers\mxc\pmic\core.
```

Table 5-3 provides a brief description of each of the device driver source files.

Note that in addition to the driver-specific source files, there also exists a `Kconfig` file that is used to define the device driver's build configuration (see Section 5.7, "Driver Configuration) and a `Makefile` that is used during the Linux kernel image build process.

Table 5-3. PMIC Protocol Driver Sources File List

File	Description
`pmic_core_spi.c`	Main function of the module, register access function
`pmic_config.h`	Define global configuration definitions or macros used by client drivers.
`pmic_event.c`	Event notification function.
`pmic_external.c`	This files contains client API implementation, define SPI interface.
`pmic-dev.c`	This provides `/dev` interface to the user-space programs.
`pmic.h`	Declaration of all the functions whose implementation differs from PMIC chip to PMIC chip.

5.7 Driver Configuration

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

5.7.1 Linux Menu Configuration Options

The following Linux kernel configuration options are provided for the PMIC protocol driver:

- Device Drivers-> MXC Support Drivers->MXC PMIC Support ->PMIC Protocol Support (SPI Interface) - Choose this to have PMIC protocol driver support. By default, this option is Y for all architectures.

```
Device Drivers-> MXC Support Drivers->MXC PMIC Support ->MXC PMIC device Interface - Choose
this to provide /dev interface to PMIC. This makes it possible to have user-space programs use
or control PMIC and for notification of PMIC events to user space.
```

- Device Drivers-> MXC Support Drivers->MXC PMIC Support -> MC13783Client Drivers - Used by all MC13783 clients - Used to enable the PMIC client drivers.
- Device Drivers-> MXC Support Drivers->MXC PMIC Support ->MXC_PMIC_FIXARB - This option includes the software workaround for reconfiguring the PMIC's arbitration bit settings in order to enable secondary processor access.

5.8 Driver Unit Tests

The built unit test applications are available at

```
/unit_tests/mxc_pmic_test/
```

The built unit test modules are available at

```
/unit_tests/modules/
```

Use the `pmic_testapp` program to test the MC13783-specific version of the protocol driver.

5.8.1 MC13783 PMIC Protocol Driver Read/Write Unit Test

The RW unit test checks the read/write functionality of the MC13783 protocol driver on a register.

```
Type: pmic_testapp -T RW
```

If the test has run correctly the result is:

```
pmic_testapp    1  PASS  :  pmic_testapp test worked as expected
```

5.8.2 PMIC Subscribe/Unsubscribe to an Event Unit Test

The SU unit test checks the ability to subscribe to an event using the MC13783 PMIC.

```
Type : pmic_testapp -T SU
```

If the test has run correctly the result is:

```
pmic_testapp    1  PASS  :  pmic_testapp test worked as expected
```

5.8.3 PMIC Subscribe, Interrupt, Unsubscribe Unit Test

The S_IT_U unit test checks the interrupt handling capability of the MC13783 protocol driver using the TSI (touchscreen) event.

```
Type : pmic_testapp -T S_IT_U
```

If the result is correct the result is:

```
Press PWR button on the KeyBoard or ATLAS
you should see IT callback info
Press Enter to continue after pressing the button
*********************************
*** IT MC13783 CALLBACK FUNCTION ***
*** TEST 2                      ***
*********************************

Did you see the callback info  [Y/N] y
Test subscribe/unsubscribe 2 event = 27

Press PWR button on the KeyBoard or ALTAS
you should see IT callback info twice.
Press Enter to continue after pressing the button
*********************************
*** IT MC13783 CALLBACK FUNCTION ***
*** TEST 2                      ***
*********************************
*********************************
*** IT MC13783 CALLBACK FUNCTION ***
*** TEST 2                      ***
```

```
*********************************
Did you see the callback info twice [Y/N] y
pmic_testapp    1  PASS  : pmic_testapp test worked as expected
```

5.8.4 PMIC Open/Close Unit Test

The OC unit test checks the ability to Open and Close API of the MC13783 PMIC.

```
Type : pmic_testapp -T OC
```

If the test has run correctly the result is:

```
pmic_testapp    1  PASS  :  pmic_testapp test worked as expected
```

5.8.5 PMIC Concurrent Access Unit Test

The CA unit test checks the ability for Concurrent Access API of MC13783 PMIC.

```
Type : pmic_testapp -T CA
```

If the test has run correctly the result is:

```
pmic_testapp    1  PASS  :  pmic_testapp test worked as expected
```

Chapter 6
PMIC Audio Driver

This chapter describes the PMIC audio device driver for Linux. The PMIC Audio Driver provides low-level control of the PMIC audio playback and recording devices.

The PMIC audio device driver uses the PMIC protocol driver (Chapter 5, "PMIC Protocol Driver) to control the audio playback and recording components of the PMIC.

6.1 PMIC Audio Driver Features

Figure 6-1 shows the key audio-related components that are provided by the MC13783 Power and Audio Management IC.

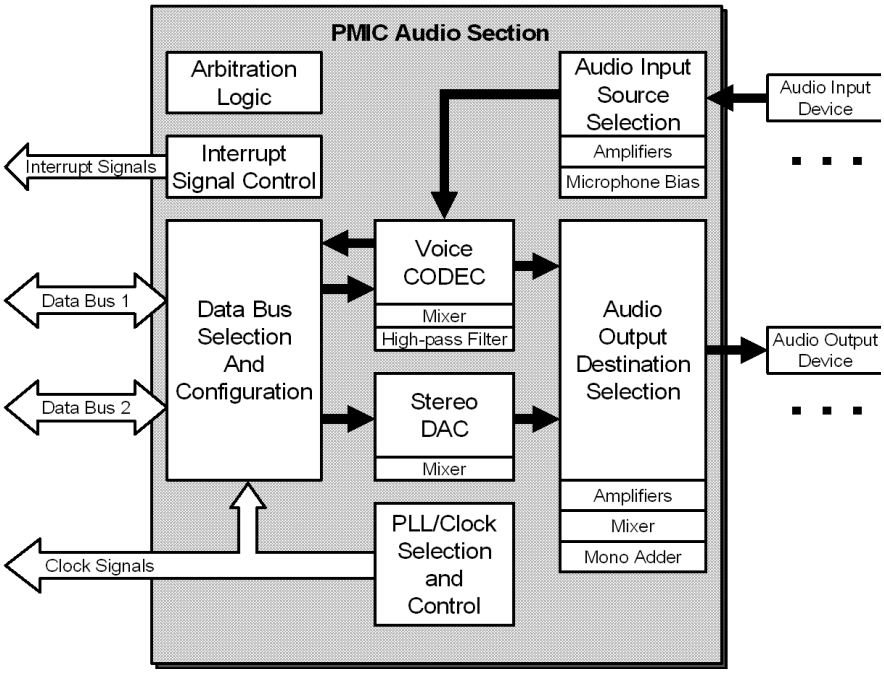

Figure 6-1. PMIC Audio Hardware Components

Even though each specific power management IC may have some unique capabilities and features, they all share the following common components and general capabilities:

- Stereo DAC —Can provide both left and right channel audio output with sampling rates from 8 kHz to 96 kHz. The Stereo DAC also has an optional internal mixer that can be used to mix together two separate input stereo audio streams to produce a single stereo output stream.
- Voice or Telephone CODEC—Provides both mono playback and recording capabilities with either an 8 kHz or 16 kHz sampling rate. The Voice CODEC on certain power management ICs may also support stereo recording but this feature is platform-specific. The Voice CODEC also includes an optional high-pass filter that can be used to filter the input or output audio streams.
- Data Bus Selection and Configuration section—Controls the connections between the audio data buses and the Voice CODEC and Stereo DAC components. The data buses can be configured to

operate either in a standard MSB-aligned mode, network mode, or I^2C mode. Each data bus can be routed to either the Voice CODEC or the Stereo DAC and both data buses can be active simultaneously along with the Voice CODEC and Stereo DAC.

- Audio output control section—Determines which devices or audio output connectors will be used as well as the gain settings on the various output amplifiers. The output control section may also include an additional mixer for mixing together the outputs from the Voice CODEC and the Stereo DAC and a mono adder for converting the Stereo DAC output to a single mono channel. However, whether these components are actually available and, if so, their exact capabilities are specific to each power management IC.
- Audio input control section—Used to select an appropriate audio recording signal source as well to configure the various input amplifiers and microphone bias circuits. The output of this section is fed directly to the Voice CODEC which then performs the analog-to-digital conversion at either an 8 kHz or 16 kHz sampling rate.
- PLL or clock control section—Can internally generate the clock signals to drive the data buses and thereby act as a bus master. Alternatively, the internal clock generator can be disabled and the power management IC operated as a slave device using an external clock source. It is recommended that the power management IC always be configured as the bus master in order to ensure that the correct clock frequencies needed to support the various audio playback and recording sampling rates are generated. This avoids having to rely on external clock sources that may have to be shared with other system devices and which may not be operating at exactly the correct frequency thereby possibly causing distortion in both audio playback and recording.
- Interrupt signal control section—Determines which hardware interrupts are enabled. The exact number and type of interrupts that may be generated is specific to each power management IC but they may include events such as the insertion of a microphone or headset.
- Arbitration control block—Determines the level of access to the audio-related hardware registers that is provided to both the primary and secondary processors. Various combinations of read-write or read-only access can be configured as required. However, the audio API does not include any access to the arbitration control block because this component is expected to be properly configured during device power-up and there is currently no operating scenario which would require reconfiguring the arbitration settings while the device is running.

As shown in Figure 6-1 above, the audio components of the power management IC connect with the processor core and other peripheral devices through the data buses, interrupt signals, and clock signals. What is not shown in Figure 6-1 are the SPI bus interfaces that are used to access the hardware control registers (including the audio-related registers) on the power management IC. The SPI bus interface and associated APIs are described in more detail in Chapter 5, “PMIC Protocol Driver.

Finally, the external audio devices, such as headsets, loudspeakers, microphones, etc. are connected to the Voice CODEC or Stereo DAC through the appropriate audio jacks and plugs. The exact type and placement of these jacks and plugs is implementation and device design-dependent. Therefore, while the current implementation does allow access to all of the available audio input and output ports provided by the power management IC, the actual device schematics or design documentation must be used to determine which ports are available and what type of connector is being used. The result of trying to use a port or external audio device which is not available or disconnected is undefined.

The power management ICs all share a similar set of components and features in terms of audio recording and playback functions. However, each power management IC also has its own unique set of features and capabilities beyond what has been described thus far. The documentation for the specific IC should be consulted in order to fully understand all of the audio-related components, features, and functions provided by a specific power management IC.

The audio API does include the ability to make use of both common and device-specific audio features as required. For example, it is possible to perform mono audio recording using the Voice CODEC on the MC13783 power management IC. However, the API also provides stereo recording through the MC13783 Voice CODEC, since that is supported on the MC13783 power management IC.

6.2 Driver Requirements

The PMIC audio driver provides full access to all of the features that are supported by the PMIC hardware. However, the API must also be identical for all ICs. Attempting to use a feature or select a configuration option that is not supported by the PMIC that is being used returns PMIC_NOT_SUPPORTED. Successful operations always return PMIC_SUCCESS, while any supported operations that failed due an error condition return PMIC_ERROR.

The PMIC audio driver must also satisfy the requirements that are stated in the WMSG Linux coding conventions document.

6.2.1 Audio Device Handle Management

The PMIC audio device driver must provide an API to support the following operations:

- Obtain a device handle for accessing the Stereo DAC, Voice CODEC, or external stereo input.
- Release a previously acquired device handle.

Higher-level device drivers that wish to access the PMIC audio components must first request and receive a valid device handle. This ensures that there will never be a conflict over access to and control of a particular audio component. Separate device handles have been defined for the Stereo DAC, the Voice CODEC, and the external stereo input.

6.2.2 Digital Audio Bus Selection and Configuration

After successfully acquiring the appropriate device handle, another set of APIs must be provided to allow for the selection and configuration of the digital audio bus:

- Select the digital audio data bus to be used.
- Configure the operating mode, timeslot selection, and timing signal parameters for the selected digital audio data bus.

Note that both the Voice CODEC and the Stereo DAC can be connected to either of the two available digital audio buses but only to one bus at a time. Furthermore, a single digital audio bus cannot be simultaneously connected to both the Voice CODEC and the Stereo DAC.

The digital audio bus must be able to operate in either master or slave modes at all of the permissible sampling rates.

6.2.3 Stereo DAC and Voice CODEC Control and Configuration

An API interface must also be provided for directly controlling the Voice CODEC and the Stereo DAC audio components. The required functionality includes the following:

- Enable/disable the audio device
- Enable/disable the available hardware mixing devices
- Perform a digital filter reset

6.2.4 Audio Input Section Control and Configuration

An API must be provided to configure the PMIC's audio input section to support using the Voice CODEC to record an audio stream. The required functionality includes the following:

- Select the desired audio input source and recording mode (stereo or mono)
- Enable/disable the audio input source
- Select the desired input amplifier gain level
- Enable/disable the appropriate microphone bias circuit

6.2.5 Audio Output Section Control and Configuration

An API must be provided to configure the PMIC's audio output section to support playback using either the Voice CODEC or the Stereo DAC. The required functionality includes the following:

- Select the desired audio source for output (e.g., Voice CODEC, Stereo DAC, or external stereo input)
- Select the desired output amplifier gain and balance levels
- Enable/disable the available hardware mixing devices
- Enable/disable the phantom ground circuit

6.2.6 Resetting the PMIC Audio Components

An API must be provided to allow partial or complete resetting of the PMIC audio components. This will provide a means to ensure that audio components are in a consistent state and to recover from any errors that might occur. The required functionality includes the following:

- Reset only the Voice CODEC or Stereo DAC settings to their respective power-on settings
- Reset all PMIC audio-related settings in all registers to their respective power-on settings

6.2.7 Audio-Related Interrupts and Event Notification

An API must be provided to allow other device drivers (e.g., the OSS sound driver) to register for and to receive notification of audio-related events. The PMIC audio driver will first receive and handle all audio-related interrupts as required but it must also allow higher-level drivers and applications access to the event notification and any associated data so that they too can respond as required. The required functionality includes the following:

- Register an event callback function
- Deregister an event callback function
- Enable/disable headset detection
- Enable/disable microphone bias detection notification (MC13783 PMIC only)

6.2.8 Additional Audio-related Configuration Options

Finally, an API must be provided to support some additional audio driver-related functions that do not necessarily fit into any of the functional categories that have already been given. The required functionality includes the following:

- Ability to query for which PMIC chip and driver is currently being used
- Ability to control the power consumption of the audio components by completely or selectively powering up and powering down specific audio circuits and devices
- Enable/disable the anti-pop circuitry
- Provide a fully decoded PMIC audio control register dump (for debugging/testing purposes only) Driver

6.3 Software Operation

The PMIC audio driver basically makes calls to the PMIC protocol driver to reconfigure the PMIC's control registers to the desired setting. All higher-level audio configuration and operation requests are converted to the appropriate PMIC control register settings and then the PMIC's hardware state is updated via the SPI bus interface.

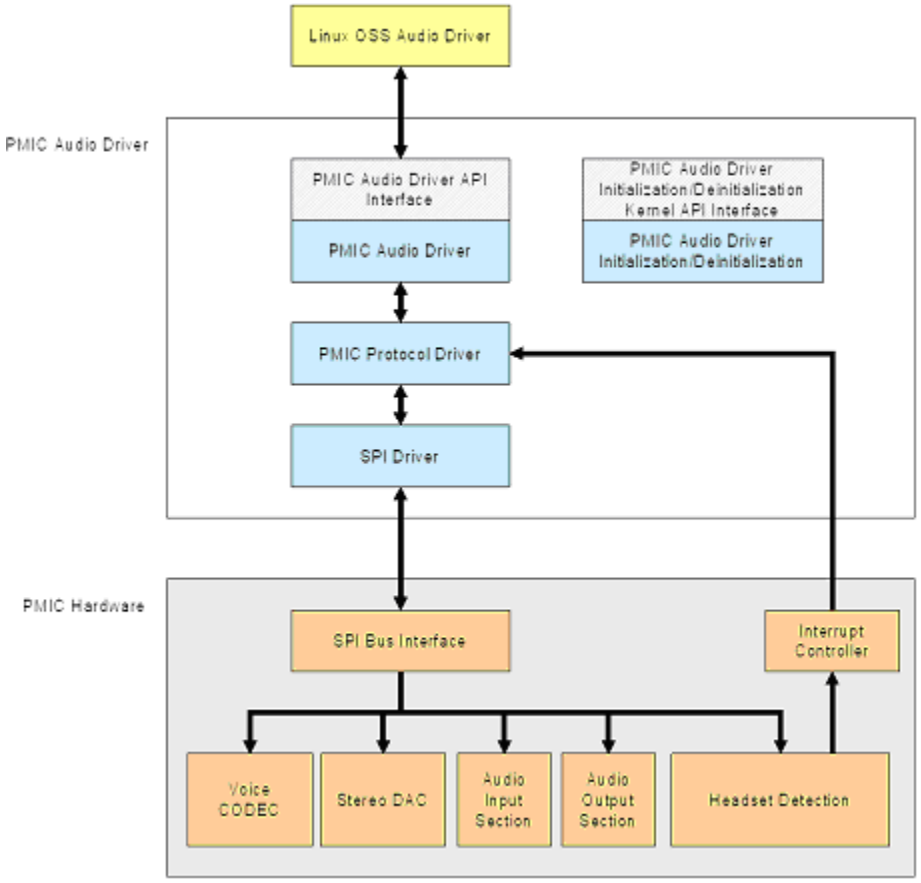

Figure 6-2. PMIC Audio Driver Architecture

6.4 Driver Architecture

Figure 6-2 shows the basic architecture of the PMIC audio driver and it's interfaces to the higher-level Linux OSS sound driver as well as the underlying PMIC hardware.

6.5 Driver Implementation Details

A structure defines the fields within each of the a PMIC's audio-related control registers. Each element of the structure defines the size and offset of the register field. This enables the use of simple macros to access each register field.

Note that the PMIC's hardware registers are not exported outside the device driver. There is no need to provide external low-level access to the PMIC's registers. This also helps to ensure the maintenance of complete control over the PMIC hardware state.

Another structure keeps track of the current PMIC hardware state. This data structure always mirrors exactly how the hardware has been configured, and avoids the possibility of conflicting or invalid configurations. It is also possible to easily return the current state of the PMIC audio hardware without using extra SPI bus transactions to directly query the hardware.

6.5.1 Driver Initialization

Nothing special needs to be done during the initialization phase for this device driver. The higher-level OSS sound driver makes calls to this driver only through the exported API and no other access method needs to be supported.

6.5.2 Driver Deinitialization

When deinitializing this driver, we must make sure that any still opened device handles are properly closed and that the PMIC hardware is restored to it's default power on state. This will help to ensure that we never leave the PMIC in an inconsistent state or that it will signal an interrupt event when there is nobody registered to properly handle it.

6.6 Driver Source Code Structure

Table 6-1 lists the MC13783-specific source files that are contained in the device driver directory

```
linux/drivers/mxc/pmic/mc13783.
```

Table 6-1. MC13783 Audio Driver Source Files

File	Description
pmic_audio.c	Implementation of the MC13783 audio PMIC client driver.
pmic_audio.h	Header file for the MC13783 audio client driver.

The header file for PMIC audio drivers is

```
linux/include/asm-arm/arch-mxc/pmic_audio.h
```

6.7 Driver Configuration

This module has Linux menu configuration options.

6.7.1 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

1. Device Drivers-> MXC Support Drivers->MXC PMIC Support -> MC13783 Client Drivers-> MC13783 Audio support - This is the configuration option to choose the MC13783-specific audio driver.

6.8 Driver Unit Tests

ALSA test applications either in the native mode or in the OSS emulation mode should presently suffice to verify the PMIC Audio driver APIs.

Chapter 7
PMIC Digitizer Driver

This chapter describes the PMIC Digitizer Driver for Linux that provides low-level access to the PMIC's analog-to-digital converters (ADC).

The PMIC digitizer driver controls the analog-to-digital converter (ADC) components of the PMIC. This capability includes taking measurements of the X-Y coordinates and contact pressure from an attached touchpanel. This device driver uses the PMIC protocol driver (See Chapter 'PMIC Protocol Driver") to access the PMIC hardware control registers that are associated with the ADC.

7.1 PMIC Digitizer Driver Features and Capabilities

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

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

Note that some of these functions (e.g., the battery monitoring and USB ID functions) are handled separately by other PMIC device drivers.

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

A hardware interrupt can be generated following the completion of an ADC conversion. A hardware interrupt can also be generated if the ADC conversion results are outside of previously defined high and low level thresholds.

Some PMIC chips also provide a pulse generator that is synchronized with the ADC conversion. The pulse generator can enable or drive external circuits in support of the ADC conversion process.

The PMIC ADC components are subject to arbitration rules as documented in the documentation for each PMIC. These arbitration rules determine how requests from both primary and secondary SPI interfaces are handled.

SPI bus arbitration configuration and control is not part of this driver, because the platform has configured arbitration settings as part of the normal system boot procedure. There is no need to dynamically reconfigure the arbitration settings after the system has been booted.

7.2 Driver Requirements

The PMIC digitizer driver is a client of the PMIC protocol driver. The PMIC protocol driver provides hardware control register reads and writes through the SPI bus interface, and also register/deregister event

notification callback functions. The PMIC protocol driver requires access to ADC-specific event notifications.

The following are the requirements for supporting a touchpanel device:

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

Note that this digitizer driver is not responsible for any additional ADC-related activities such as battery level or USB ID handling. Such functions are handled by other PMIC-related device drivers.

Also, as previously indicated, this device driver is not responsible for SPI bus arbitration configuration. The appropriate arbitration settings that are required in order for this device driver to work properly are expected to have been set during the system boot process.

7.3 Driver Software Operation

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

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

7.4 Driver Architecture

Figure 7-1 shows the basic architecture for the PMIC digitizer driver. The PMIC protocol driver and the platform's SPI driver provides the necessary interface to read and write the PMIC's hardware control registers.

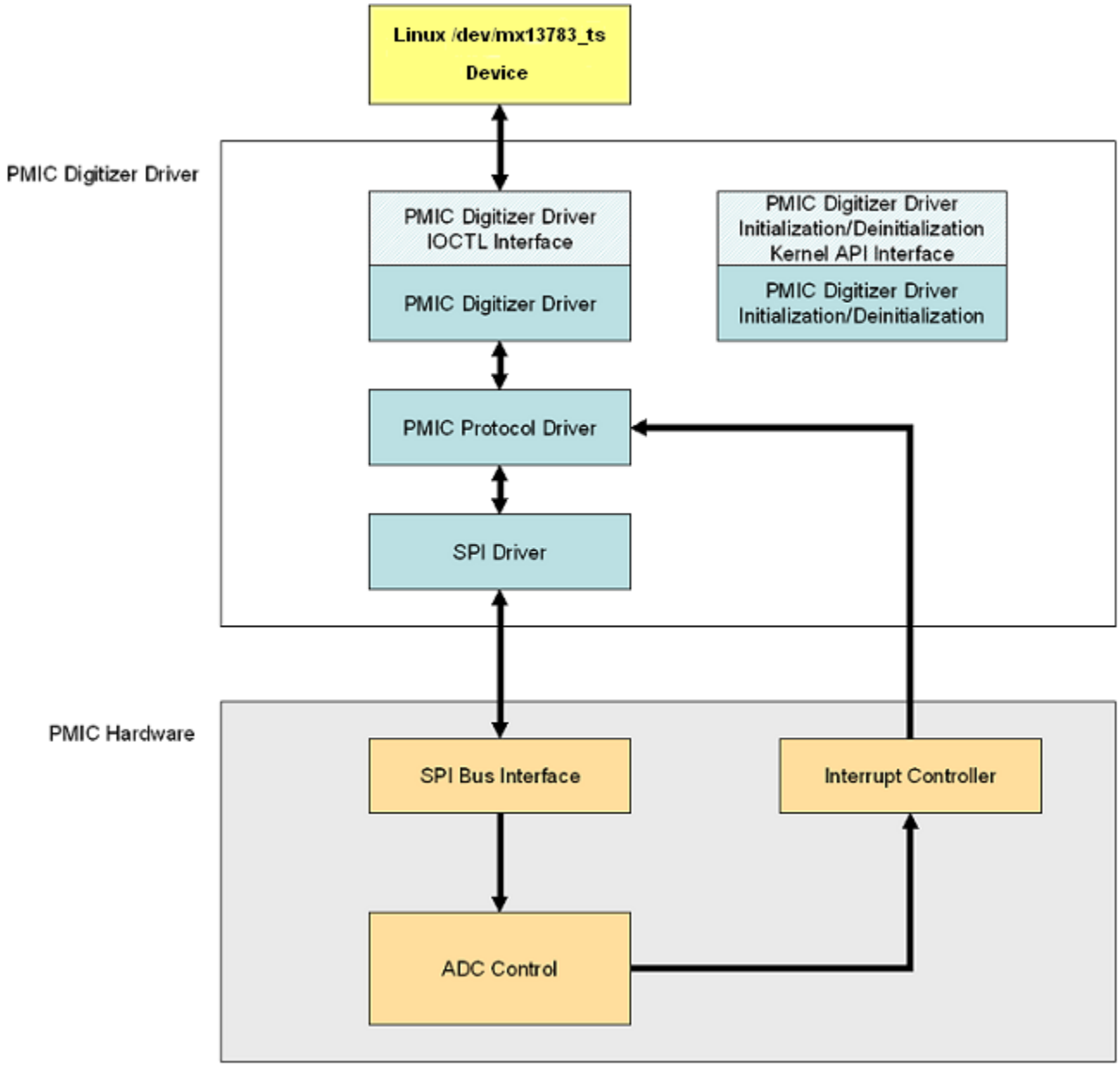

Figure 7-1. PMIC Digitizer Driver Architecture

The PMIC's interrupt controller generates interrupts for the following events:

- ADC end-of-conversion
- High/low level threshold exceeded

7.5 Driver Implementation Details

The PMIC ADC conversion can take a significant amount of time.

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

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

7.5.1 Driver Initialization

The PMIC digitizer driver must also create the appropriate `/dev` character device entry, to allow applications to obtain the touchpanel (X,Y) coordinates and pressure measurements. The touchpanel device is only required to support a read operation.

The following devices are created

- For MC13783— `/dev/mc13783_ts` device

A device-independent softlink, `/dev/ts`, that references the PMIC-specific touchpanel device name is also created, but this is part of a Linux boot script and is not handled by this device driver.

7.5.2 Driver Removal

The PMIC digitizer driver must remove the `/dev` device entry that was created when the driver was loaded.

7.6 Driver Source Code Structure

Table 7-1 lists the source files for the MC13783-specific version of this driver. These are contained in the following directory:

```
linux/drivers/mxc/pmic/mc13783.
```

Table 7-1. MC13783 Digitizer Driver Source Files

File	Description
pmic_adc.c	Implementation of the MC13783 ADC client driver.
pmic_adc_defs.h	Hardware definitions and internal functions for the ADC client driver.

The header file for PMIC adc drivers is

```
linux/include/asm-arm/arch-mxc/pmic_adc.h
```

7.7 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

1. Device Drivers-> MXC Support Drivers->MXC PMIC Support -> MC13783 Client Drivers-> MC13783 ADC support—Chooses the MC13783 (MC13783) -specific digitizer driver.

7.8 Driver Unit Tests

Use the "`pmic_testapp_adc`" program to test the MC13783-specific version of this device driver.

7.8.1 PMIC Read ADC Value Tests

This test returns the last correct touch screen value of MC13783 ADC.

1. Type the following:

```
pmic_testapp_adc -T CONV
```

The system returns:

```
pmic_testapp_adc    0  INFO  :  Testing if 46792 test case is OK

========== TESTING PMIC adc DRIVER ==========
Select a channel [0-15] :
=>0
Test convert ADC functions
Channel 0: 746
Convert 8x channel 0 - 0: 747
Convert 8x channel 0 - 1: 747
Convert 8x channel 0 - 2: 749
Convert 8x channel 0 - 3: 750
Convert 8x channel 0 - 4: 750
Convert 8x channel 0 - 5: 752
Convert 8x channel 0 - 6: 752
Convert 8x channel 0 - 7: 752
MULTICHANNEL From channel 0 - 0: 748
MULTICHANNEL From channel 0 - 1: 210
MULTICHANNEL From channel 0 - 2: 624
MULTICHANNEL From channel 0 - 3: 26
MULTICHANNEL From channel 0 - 4: 1023
MULTICHANNEL From channel 0 - 5: 1023
MULTICHANNEL From channel 0 - 6: 0
MULTICHANNEL From channel 0 - 7: 274
pmic_testapp_adc    1  PASS  :  46792 test case worked as expected
```

7.8.2 PMIC monitoring tests

This test returns monitoring touch screen value of MC13783 ADC.

1. Type the following:

```
pmic_testapp_adc -T MON
```

The system returns:

```
pmic_testapp_adc    0  INFO  :  Testing if 46792 test case is OK

Test monitoring ADC functions
```

```
pmic_testapp_adc    1  PASS  :  46792 test case worked as expected
```

Chapter 8
PMIC Power Management Driver

The PMIC power management device driver for Linux provides the low-level control of the power supply regulators, selection of voltage levels, and enabling/disabling of various low-power modes.

This device driver makes use of the PMIC protocol driver (See the 'PMIC Protocol Driver" Chapter) to access the PMIC hardware control registers.

8.1 PMIC Features

Using the PMIC chip in a product potentially provides a complete power control and power management strategy. The PMIC chips have built-in switching power supplies and linear voltage regulators that can be configured to power the rest of the platform. These power supplies may also be selectively enabled/disabled, and the voltage levels may be dynamically adjusted to control power consumption.

In addition, there is an internal state machine that can provide automatic power-cut functions and transparent transitions between various low-power operating modes. Full shutdown and automatic restart based on possible external events is also supported.

The IC documentation should be consulted for full details about what power supplies are provided, how they can be configured, and how the internal power control logic is implemented.

8.2 Driver Requirements

The PMIC power management driver is a client of the PMIC protocol driver. It provides services for power management control of the PMIC component.

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

8.3 Driver Software Operation

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

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

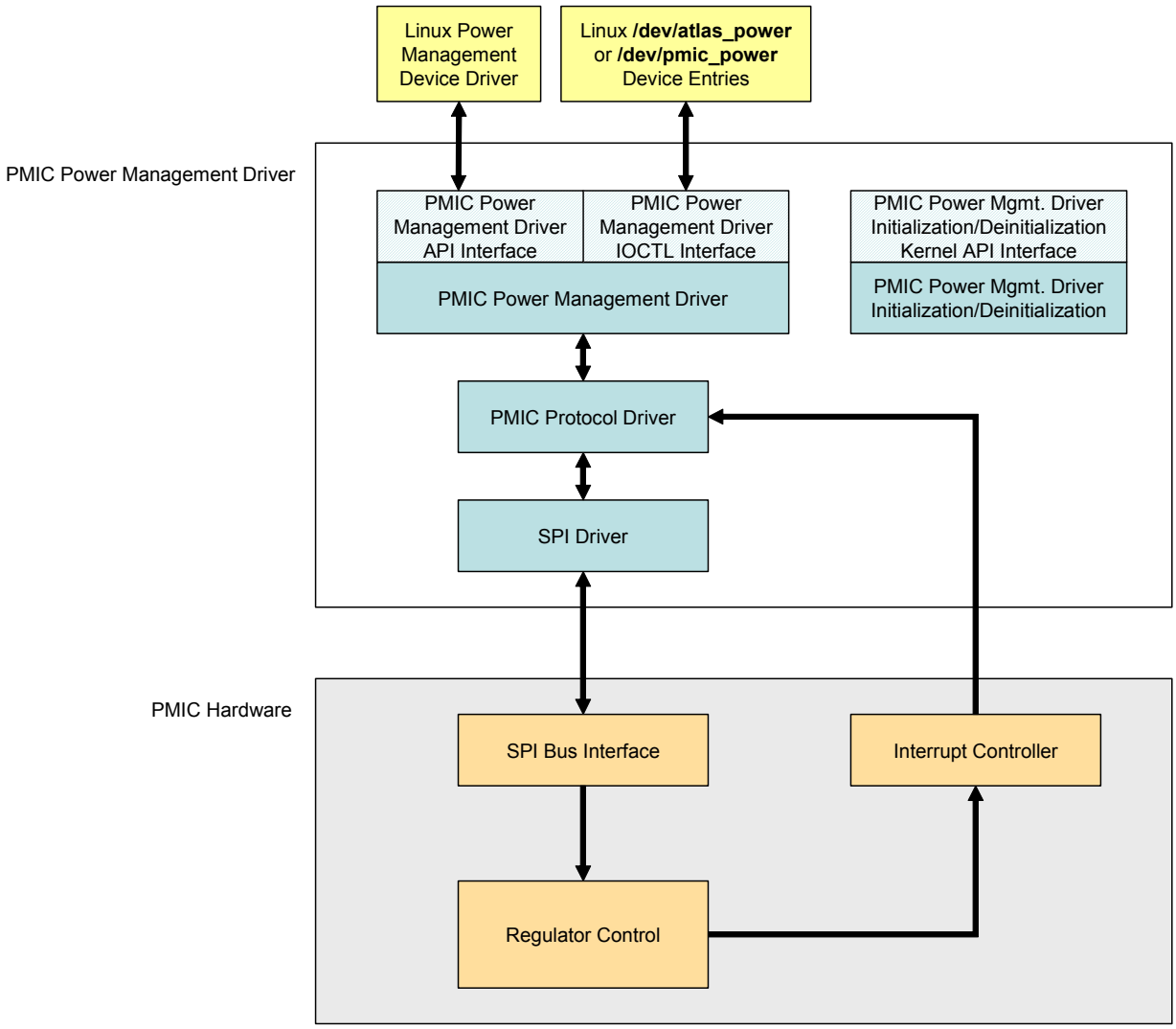

Figure 8-1. PMIC Power Management Driver Architecture

8.4 Driver Architecture

Figure 8-1 shows the basic architecture of the PMIC power management driver, as well as its higher-level interfaces and the connections to the underlying PMIC hardware.

8.5 Driver Implementation Details

Access to the PMIC power management driver is provided through a set of exported APIs. The exported APIs are meant for use by other kernel-mode device drivers. The IOCTL interface is provided in a separate test module. All IOCTL calls are translated into corresponding internal API calls to perform the requested operation.

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

8.6 Driver Source Code Structure

Table 8-1 lists the MC13783-specific source files for the power management driver. These are contained in the device driver directory

```
linux/drivers/mxc/pmic/mc13783.
```

Table 8-1. MC13783 Power Management Driver Source Files

File	Description
pmic_power.c	Implementation of the MC13783 power management client driver.
pmic_power_defs.h	Internal header for MC13783 power management client driver.

The header file for PMIC Power drivers is

```
linux/include/asm-arm/arch-mxc/pmic_power.h
```

8.7 Driver Configuration

This module has Linux menu configuration options.

8.7.1 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- Device Drivers-> MXC Support Drivers->MXC PMIC Support -> MC13783 Client Drivers-> MC13783 Power support—Chooses the MC13783-specific power management driver.

8.8 Driver Unit Tests

Use the "`pmic_testapp_power`" program to test the MC13783-specific version of this device driver. To run this test application, the test module `mxc_pmic_power_testmod.ko` must be inserted.

8.8.1 PMIC Regulator ON/OFF Tests

This test checks the Regulator ON/OFF API function of the PMIC power.

For the PMIC ON/OFF test, type this line:

```
./pmic_testapp_power -T 1
```

The following result appears for a successful test:

```
Testing if REGULATOR ON is OK
REGULATOR 4 : ON
REGULATOR 5 : ON
REGULATOR 6 : ON
REGULATOR 7 : ON
REGULATOR 8 : ON
REGULATOR 9 : ON
REGULATOR 10 : ON
REGULATOR 11 : ON
```

```
REGULATOR 12 : ON
REGULATOR 13 : ON
REGULATOR 14 : ON
REGULATOR 15 : ON
REGULATOR 16 : ON
REGULATOR 17 : ON
REGULATOR 18 : ON
REGULATOR 19 : ON
REGULATOR 20 : ON
REGULATOR 21 : ON
REGULATOR 22 : ON
Test PASSED
Testing if REGULATOR OFF is OK
Switching OFF is restricted to limited of Regulator
as its' dangerous to switch-off regulators randomly.
REGULATOR 4 : OFF
REGULATOR 5 : OFF
REGULATOR 6 : OFF
REGULATOR 16 : OFF
REGULATOR 18 : OFF
Test PASSED
```

8.8.2 PMIC Switcher and Regulator Configuration Tests

This test checks switches/Regulator Configuration API function of PMIC power.

Note: Make sure the config file mc13783_power.cfg/sc55112_power.cfg is in the working directory. It is available in misc/test/mxc_pmic_test/pmic_testapp_power/ .

For the PMIC Switcher/Regulator Configuration test, type this line:

```
./pmic_testapp_power -T 2
```

The following result appears for a successful test:

```
Testing if SWITCHES/REGULATOR configurations are OK
REGULATOR 0 : Test PASSED
REGULATOR 0 : Test PASSED
REGULATOR 1 : Test PASSED
REGULATOR 1 : Test PASSED
REGULATOR 2 : Test PASSED
REGULATOR 2 : Test PASSED
REGULATOR 3 : Test PASSED
REGULATOR 3 : Test PASSED
REGULATOR 4 : Test PASSED
REGULATOR 5 : Test PASSED
REGULATOR 6 : Test PASSED
REGULATOR 7 : Test PASSED
REGULATOR 8 : Test PASSED
REGULATOR 8 : Test PASSED
REGULATOR 9 : Test PASSED
REGULATOR 9 : Test PASSED
REGULATOR 10 : Test PASSED
REGULATOR 10 : Test PASSED
REGULATOR 11 : Test PASSED
REGULATOR 11 : Test PASSED
REGULATOR 12 : Test PASSED
REGULATOR 12 : Test PASSED
REGULATOR 13 : Test PASSED
```

```
REGULATOR 13 : Test PASSED
REGULATOR 14 : Test PASSED
REGULATOR 14 : Test PASSED
REGULATOR 15 : Test PASSED
REGULATOR 15 : Test PASSED
REGULATOR 16 : Test PASSED
REGULATOR 16 : Test PASSED
REGULATOR 17 : Test PASSED
REGULATOR 19 : Test PASSED
REGULATOR 19 : Test PASSED
REGULATOR 20 : Test PASSED
REGULATOR 20 : Test PASSED
REGULATOR 21 : Test PASSED
REGULATOR 21 : Test PASSED
REGULATOR 22 : Test PASSED
REGULATOR 22 : Test PASSED
TEST CASE PASSED
```

8.8.3 PMIC Miscellaneous Feature Tests

This test checks regulator API function of MC13783 power.

For the PMIC Switcher test, type this line:

```
/pmic_testapp_power -T 3
```

The following result appears for a successful test:

```
Testing if Miscellaneous features are OK
This testcase is not valid for SC55112
Testing if Configuration of System Reset Buttons are OK
Test PASSED
Testing if Auto Reset is configured
Test PASSED
Testing if ESIM control voltage values are configured\
Test PASSED
Testing if Regen polarity is configured
Test PASSED
Testing if battery detect enable is configured
Test PASSED
Testing if AUTO_VBKUP2 enable is configured
Test PASSED
Testing if Power Control Configurations are OK
Test PASSED
TEST CASE PASSED
```

Chapter 9
PMIC Connectivity Driver

The MC13783 PMIC Connectivity Driver for Linux provides support for external connectivity of the following types:

- RS-232
- USB OTG
- CEA936

The MC13783 PMIC Connectivity Driver is based on the MC13783 DTS 3.0 specification and the Freescale MC13783 board.

This device driver makes use of the PMIC protocol driver (See the “PMIC Protocol Driver” chapter) to access the PMIC hardware control registers.

9.1 PMIC Features

The PMIC includes transceivers to support both RS-232 and USB On-the-Go (OTG) external connectivity. The MC13783 PMIC also includes support for the CEA936 specification. Due to the limited number of available pin connections, only one of these external connectivity modes can be used at any one time. In the case of the USB OTG transceiver, the specific connections that are made between the PMIC transceiver and the host platform may also affect which USB OTG operating modes are supported.

The PMIC documentation should also be consulted for details about the configuration and use of the RS-232 and USB OTG transceivers.

9.2 Driver Requirements

The PMIC connectivity driver is a client of the PMIC protocol driver and uses it to provide access to the PMIC’s hardware control registers. The PMIC connectivity driver, in turn, must provide a suitable API interface for the Linux UART and USB OTG drivers to support both RS-232 and USB OTG connectivity. The required functionality includes the following:

- Acquisition and release of a connectivity device handle—The current owner of the device handle is granted exclusive access to the PMIC’s transceivers as long as the handle is being held. Any attempts to access or use the connectivity hardware without first successfully acquiring the device handle result in the return of PMIC_ERROR.
- Selection of one of the supported operating modes—For example, RS-232, USB OTG, or CEA-936. Attempting to use an unsupported mode results in a NOT_SUPPORTED return.

NOTE

The list of supported modes may differ from PMIC to PMIC.

- Registration and removal of an event handler callback function—Any attempts to register a callback for an unsupported event results in a NOT_SUPPORTED return.
- Invocation of all registered callback functions—When the matching event has been signalled by the PMIC protocol driver.

- Configuration of the PMIC USB transceiver—As required to communicate with the platform's USB controller. This includes, for example, configuring the operating speed and the transceiver's power supply.
- Configuration of the PMIC USB transceiver—As required to support the additional USB OTG requirements. This includes, for example, setting the Data Line Pulse duration and performing a Host Negotiation Protocol sequence.
- Configuration of the PMIC RS-232 transceiver—As required to support an RS-232 connection.
- Configure the PMIC hardware to support the CEA-936 operating mode— Attempting to use the CEA-936 mode when it is not supported by the underlying PMIC hardware results in a NOT_SUPPORTED return.

NOTE

Currently, this mode is only supported by the MC13783 PMIC.

- Ability to explicitly reset the PMIC's connectivity-related hardware components to their default or power-on state—This function is useful to reinitialize the connectivity hardware to a known state.
- Automatic RS-232 to USB OTG mode switch—As supported by the PMIC, whenever a USB device is attached while idle in RS-232 mode.

The specific hardware register settings that are required to configure the PMIC connectivity components are found in the documentation for each PMIC chip.

In addition to the functional requirements listed above, the PMIC connectivity device driver must also conform to the WMSG Linux coding conventions.

9.3 Driver Software Operation

Most of the operations that must be performed by the PMIC connectivity driver involve setting the appropriate values in the PMIC hardware control registers using the APIs that are provided by the PMIC protocol driver. Specific settings for each PMIC can be found within the documentation for each PMIC chip.

Event handler callback functions, if any, are registered using PMIC protocol driver APIs. The PMIC protocol driver interrupt handler automatically invokes all registered callback functions whenever the associated event is signaled by the PMIC hardware.

Note that this device driver is not responsible for handling the actual RS-232 or USB OTG data transfer operations.

NOTE

This device driver is not responsible for handling the actual RS-232 or USB OTG data transfer operations.

Higher-level UART or USB OTG drivers handle the transfers, as well as the configuration of the UART and USB OTG controllers. The PMIC chips only provide the transceiver components, an event notification capability, and the connections to any external connectors. The PMIC connectivity driver's role is restricted to transceiver configuration and event notification.

9.4 Driver Architecture

Figure 9-1 shows the basic architecture of the PMIC connectivity driver, as well as its relationship to the Linux UART and USB OTG drivers and the PMIC hardware components. The UART driver configures the RS-232 transceiver, while the USB OTG driver handles the USB OTG transceiver. Only one of these transceivers can be active at any one time.

The only events that are typically reported to the PMIC connectivity driver involve the detection of USB-related state changes. In particular, the voltage level that is measured on the USB signal lines is used to indicate the insertion or removal of a device, the requested operating speed, and whether it operates as a host or a peripheral (for USB OTG devices). Details about the signalling and correct handling of the various USB-related events can be found in the USB specification and the USB OTG supplement.

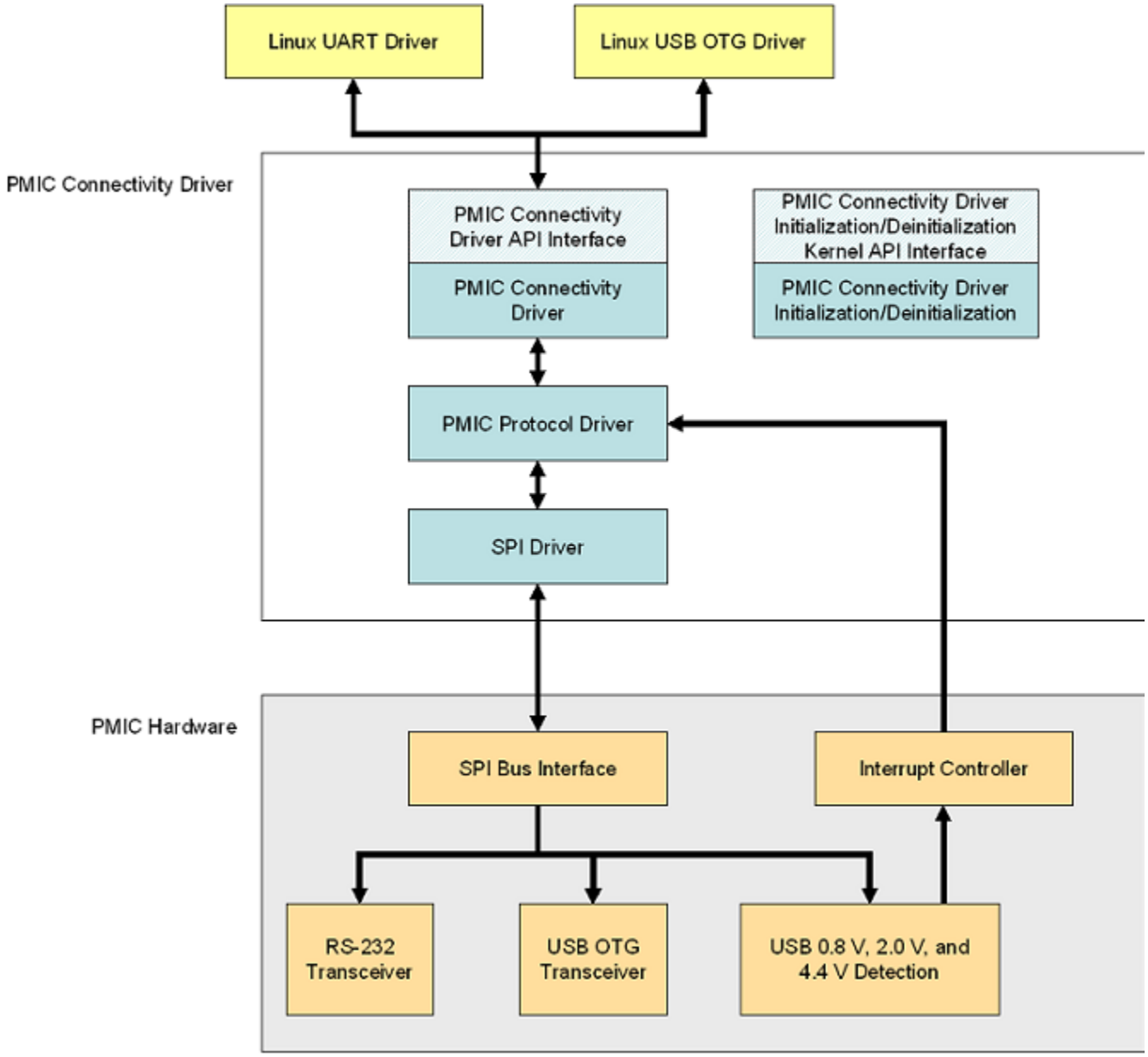

Figure 9-1. PMIC Connectivity Driver Architecture

9.5 Driver Implementation Details

The PMIC connectivity device driver uses a single data structure to track the current state of the device handle and all available PMIC configuration options. All APIs that modify the PMIC hardware also update the data structure, so the device driver state always matches that of the hardware.

A mutex is used to ensure that the state of the device driver and the PMIC hardware are always kept in a consistent state. A mutex is used whenever the system is not operating in an atomic or interrupt context within this driver. In the limited places where the system is operating in an atomic or interrupt context, a

spinlock is also acquired. The spinlock is released at the earliest possible time in order to minimize interrupt handling latencies.

A tasklet is used to perform most of the event handling operations, so as to minimize the time actually spent in the low-level interrupt handling routine.

9.5.1 Driver Initialization

The device driver initialization sequence continues as normal with no special provision for the PMIC commectivity device driver.

NOTE

No device name is created within the /dev directory, because this driver does not support any IOCTL interfaces.

9.5.2 Driver Removal

If the device handle is still being held when this driver is removed, then the handle must be forcibly closed and the PMIC connectivity components must be restored to default power-on state, before the deinitialization sequence is completed.

9.6 Driver Source Code Structure

Table 9-1 lists the MC13783-specific source files for this device driver, as contained in the device driver directory

`linux/drivers/mxc/pmic/mc13783.`

Table 9-1. MC13783 Connectivity Driver Source Files

File	Description
pmic_convity.c	Function for MC13783 USB/RS232 connectivity client.

9.7 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- Device Drivers-> MXC Support Drivers->MXC PMIC Support -> MC13783 Client Drivers-> MC13783 Connectivity support—Chooses the MC13783–specific connectivity driver.

9.8 Driver Unit Tests

The mc13783 specific tests in `misc/module_test/pmic_convity_test`. This test suite can be configured and built in the same manner as the rest of the existing module tests. The result of a successful build is a collection of utestXX_YYYY.ko loadable kernel modules where XX represents the test number and YYYY matches the name of a connectivity device driver API.

9.8.1 MC13783 PMIC Connectivity Tests

After it has been successfully built, the tests in the test suite can be executed by typing

```
insmod utest9_set_callback.ko
PMIC Connectivity Unit Test loading

Testing pmic_convity_set_callback ...WAITING FOR PLUG IN / REMOVAL

DETECTED  EVENT: - TEST PASSED
```

Chapter 10
PMIC Battery Driver

The PMIC battery device driver for Linux provides support for controlling the PMIC battery interface circuits. This device driver makes use of the PMIC protocol driver (See Chapter 'PMIC Protocol Driver") to access the PMIC hardware control registers.

10.1 PMIC Features

PMIC chips include circuits to automatically detect the presence of a charger and to recharge the system battery. Additional circuits are provided to detect and prevent overcharging. Additional capabilities include:

- Support for USB chargers
- Support for a coin cell charger

Battery voltage levels can also be monitored using the analog-to-digital converter and low voltage or battery end-of-life conditions can be signalled through the use of hardware interrupts.

10.2 Driver Requirements

This module is a client of the PMIC protocol driver and uses it to provide access to the PMIC's hardware control registers. The PMIC battery driver, in turn, must provide an IOCTL interface that applications can use to control and monitor the state of the battery and charger circuits. The required functionality includes the following:

- API for battery charger control including selecting the appropriate charger path
- Configure the charging mode (e.g., the charge current level)
- Configure the battery voltage and current level monitoring and end-of-life functions.

The specific hardware register settings that are required to configure the battery control circuits can be found in the documentation for each PMIC chip.

In addition to the functional requirements listed above, the PMIC battery device driver must also conform to the WMSG Linux coding conventions.

10.3 Driver Software Operation

The PMIC battery driver provides an IOCTL interface through the `/dev/pmic_battery` device. Applications use this driver to access the PMIC battery control registers and circuits. The battery driver actually uses the PMIC protocol driver's APIs to perform the necessary hardware control register read/write operations.

The PMIC protocol driver's APIs are also used to register/deregister event handler callback functions. Event handlers can be registered for any of the supported battery-related event notifications, for example, a battery end-of-life condition, detection of a charger being attached, or a charger overvoltage condition.

10.4 Driver Architecture

Figure 10-1 shows the basic architecture of the PMIC battery device driver along with the associated PMIC hardware components.

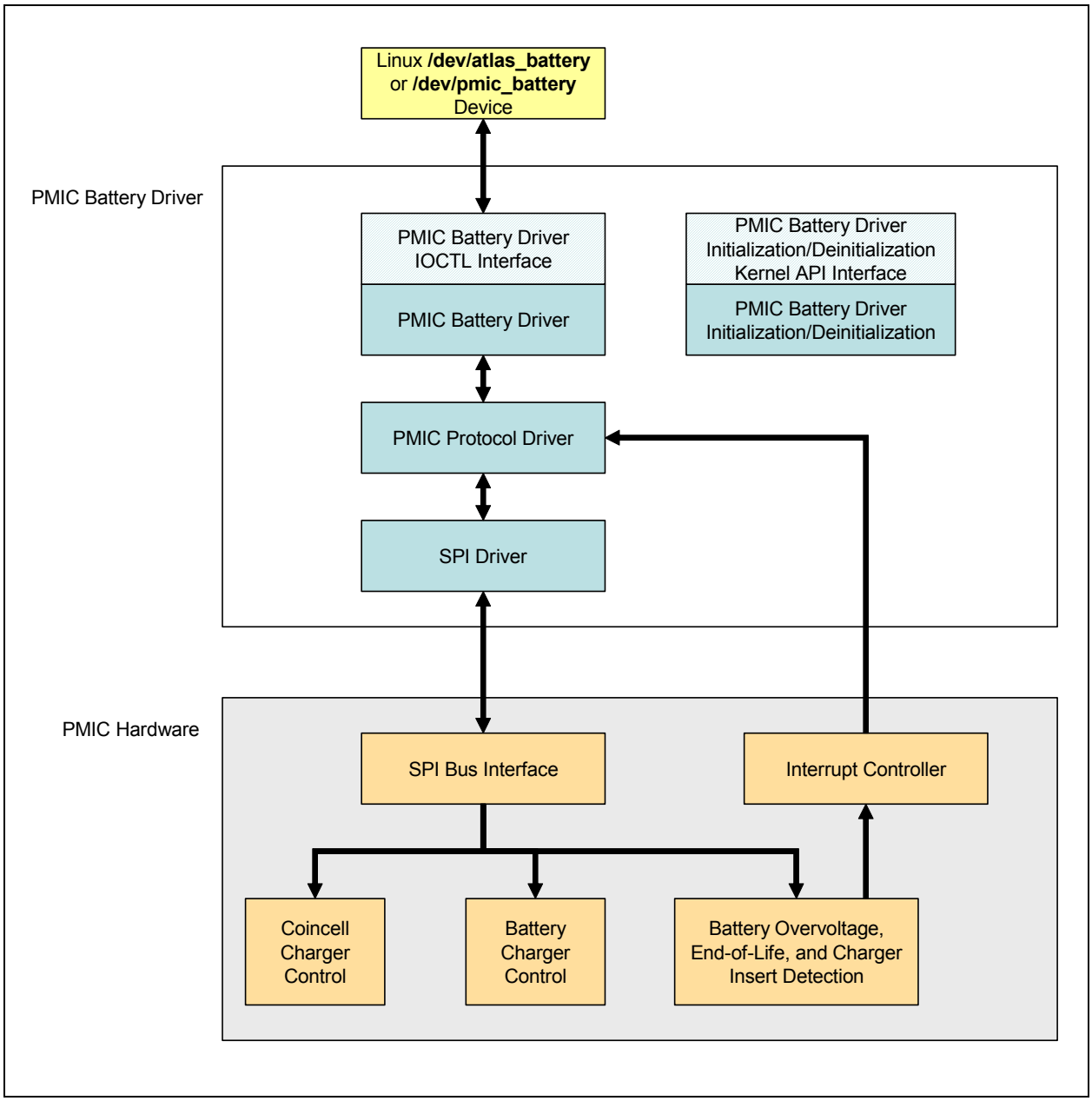

Figure 10-1. PMIC Battery Device Driver Architecture

10.5 Driver Implementation Details

The implementation of the PMIC battery driver is relatively straightforward and involves providing the appropriate IOCTL interface to support the `/dev/pmic_battery` device. Internally, each IOCTL call is translated to the appropriate PMIC hardware control register operations, which are then performed with the aid of the PMIC protocol and SPI drivers.

Event handler callback functions are registered directly with the PMIC protocol driver. The registered event handler is invoked when the corresponding event is detected and the hardware interrupt is received by the PMIC protocol driver.

10.5.1 Driver Initialization

During initialization, register a `/dev/pmic_battery` device to allow application-level access to the device driver using the IOCTL interface.

10.5.2 Driver Deinitialization

The previously registered `/dev/pmic_battery` device entry must be removed when the device driver is unloaded.

10.6 Driver Source Code Structure

Table 10-1 lists the source files for this driver that are contained in the following directory:

`linux/drivers/mxc/pmic/mc13783.`

Table 10-1. MC13783 Battery Driver Source Files

File	Description
pmic_battery.c	Implementation of the PMIC battery client driver.
pmic_battery_defs.h	Define hardware registers for the PMIC battery driver.

The header file for PMIC battery drivers is

`linux/include/asm-arm/arch-mxc/pmic_battery.h`

10.7 Linux Menu Configuration Options

The following Linux kernel configurations are provided for the PMIC Battery device driver:

- Device Drivers-> MXC Support Drivers->MXC PMIC Support -> MC13783 Client Drivers-> MC13783 Battery API support—Chooses the mc13783-specific version of the battery device driver.

10.7.1 Charger Management Tests

This test checks charger management API function of PMIC battery.

```
Type : ./pmic_testapp_battery -T 0
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test main charger control function of PMIC Battery
pmic_battery_testapp    0  INFO  :  VOLTAGE = 0  CURRENT = 0
pmic_battery_testapp    0  INFO  :  VOLTAGE = 0  CURRENT = 1
.....
pmic_battery_testapp    0  INFO  :  VOLTAGE = 7  CURRENT = 6
pmic_battery_testapp    0  INFO  :  VOLTAGE = 7  CURRENT = 7
pmic_battery_testapp    0  INFO  :  Test disable charger control function of PMIC
Battery driver
pmic_battery_testapp    0  INFO  :  Test COIN cell charger control function of PMIC
Battery
pmic_battery_testapp    0  INFO  :  VOLTAGE = 0
....
pmic_battery_testapp    0  INFO  :  VOLTAGE = 3
pmic_battery_testapp    0  INFO  :  Test disable charger control function of PMIC
Battery driver
pmic_battery_testapp    0  INFO  :  Test TRICKLE charger control function of PMIC
Battery
pmic_battery_testapp    0  INFO  :  CURRENT = 0
......
pmic_battery_testapp    0  INFO  :  CURRENT = 7
pmic_battery_testapp    0  INFO  :  Test disable charger control function of PMIC
Battery driver
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_0 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_0 test case working as
expected
```

10.7.2 EOL Comparator Tests

This test checks charger management API function of PMIC battery.

```
Type : ./pmic_testapp_battery -T 1
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test enable eol function of PMIC Battery
pmic_battery_testapp    0  INFO  :  THRESHOLD = 0
pmic_battery_testapp    0  INFO  :  THRESHOLD = 1
pmic_battery_testapp    0  INFO  :  THRESHOLD = 2
pmic_battery_testapp    0  INFO  :  THRESHOLD = 3
pmic_battery_testapp    0  INFO  :  Test disable eol function of PMIC Battery
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_1 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_1 test case working as
expected
```

10.7.3 LED Management Tests

This test checks LED management API function of PMIC battery.

```
Type : ./pmic_testapp_battery -T 2
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test led control function of PMIC Battery
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_2 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_2 test case working as
expected
```

10.7.4 Reverse Supply and Unregulated Modes Tests

This test checks Reverse Supply and Unregulated Modes API function of PMIC battery.

```
Type : ./pmic_testapp_battery -T 3
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test set reverse supply function of PMIC Battery
pmic_battery_testapp    0  INFO  :  Test set reverse supply function of PMIC Battery
pmic_battery_testapp    0  INFO  :  Test unregulated function of PMIC Battery enable
pmic_battery_testapp    0  INFO  :  Test unregulated function of PMIC Battery disable
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_3 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_3 test case working as
expected
```

10.7.5 Set Out Control Tests

This test checks Set Out Control API function of PMIC battery.

```
Type : ./pmic_testapp_battery -T 4
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test set out control function of PMIC Battery in
CONTROL_BPFET_LOW
pmic_battery_testapp    0  INFO  :  Test set out control function of PMIC Battery in
CONTROL_BPFET_HIGH
pmic_battery_testapp    0  INFO  :  Test set out control function of PMIC Battery in
CONTROL_HARDWARE
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_4 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_4 test case working as
expected
```

10.7.6 Set Over Voltage Threshold

This test checks Set Over Voltage Threshold API function of PMIC battery.

```
Type : ./pmic_testapp_battery -T 5
```

If the result is correct you can see:

```
pmic_battery_testapp    0  INFO  :  Test set threshold function of PMIC Battery
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_5 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_5 test case working as
expected
```

10.7.7 Get Charger Current

This test checks Get Charger Current API function of PMIC battery.

```
Type : ./pmic_testapp_battery -T 6
```

If the result is correct you can see:

```
charger current : 1023.
pmic_battery_testapp    0  INFO  :  Testing pmic_battery_testapp_6 test case is OK
pmic_battery_testapp    1  PASS  :  pmic_battery_testapp_6 test case working as
expected
```

Chapter 11
PMIC Light Driver

The MC13783 PMIC Light Driver for Linux provides access to the PMIC's backlight and LED control circuits.

This device driver makes use of the PMIC protocol driver (See Chapter "PMIC Protocol Driver") to access the PMIC hardware control registers.

11.1 PMIC Features

The PMIC chip includes circuits to control the following external components:

- Backlight (for LCD or keypads)
- Color LEDs

The current level and duty cycle can be controlled as required to satisfy a wide variety of operating requirements. The color LEDs can also be configured to Flash in a number of different patterns.

Complete information about the backlight and LED controls can be found in the documentation for each PMIC.

11.2 Driver Requirements

The PMIC light driver must provide access to all of the PMIC backlight and LED control circuits. This includes configuring the current levels, duty cycle, and Flashing modes. Note that the actual external devices that are attached to the PMIC differ from platform to platform. Therefore, while the light driver must provide access to all of the PMIC supported features, it cannot make any assumptions about the actual nature of the external devices (e.g., the color of the LEDs that are attached) and whether they actually exist or not.

Note that the PMIC light driver interface may include functions that are not supported by all PMIC chips. Attempting to use a configuration that is not supported by the current PMIC hardware returns NOT_SUPPORTED.

In addition to the technical requirements that are described in the following sections, the PMIC light device driver must also conform to the WMSG Linux coding conventions.

11.2.1 Backlight Control Functions

The PMIC backlight circuits are intended to support the control of the backlight level for an LCD display and/or the keypad. The device driver must support the following operations:

- Enable/disable the backlight
- Set/get the backlight current level
- Set/get the backlight duty cycle
- Set/get the backlight cycle time
- Configure the backlight ramp up and ramp down settings

- Configure the backlight strobe settings

11.2.2 LED Control Functions

The LED control circuits supplement the backlight circuits by providing the ability to control additional light sources for signaling purposes and for other special effects. The LED channels are labelled as being R, G, and B because one typical application would be to attach red, green, and blue LEDs, respectively, to each channel. However, this is not a required configuration, and other types of LEDs may be used with these circuits. The device driver must support the following operations:

- Enable/disable each individual colored LED circuit
- Select either colored LED or funlight operating modes
- Set/get the colored LED current level
- Set/get the colored LED blink pattern
- Set/get the funlight current level
- Set/get the funlight duty cycle
- Set/get the funlight cycle time
- Configure the funlight ramp settings
- Configure the funlight strobe settings
- Enable/disable audio modulation

11.3 Driver Software Operation

The operation of the PMIC light driver is fairly simple, and only involves configuring the PMIC hardware control registers as required. Access to the PMIC hardware control registers uses the PMIC protocol driver, which in turn, uses the SPI driver.

As no standard Linux device driver exists to control backlight and external LEDs, applications must use the documented IOCTL interface to access the PMIC light driver. Note, however, that not all of the available backlight and LED control functions are supported by a specific PMIC chip. The device driver returns NOT_SUPPORTED if an attempt is made to use a configuration or function that is not supported by the underlying PMIC hardware.

The PMIC-specific control register settings that are required to configure the various backlight and LED control circuits can be found in the documentation for each PMIC.

NOTE

No interrupt or notification events are associated with the PMIC light driver.

11.4 Driver Architecture

Figure 11-1 shows the basic PMIC light driver architecture along with the PMIC hardware components that are being used.

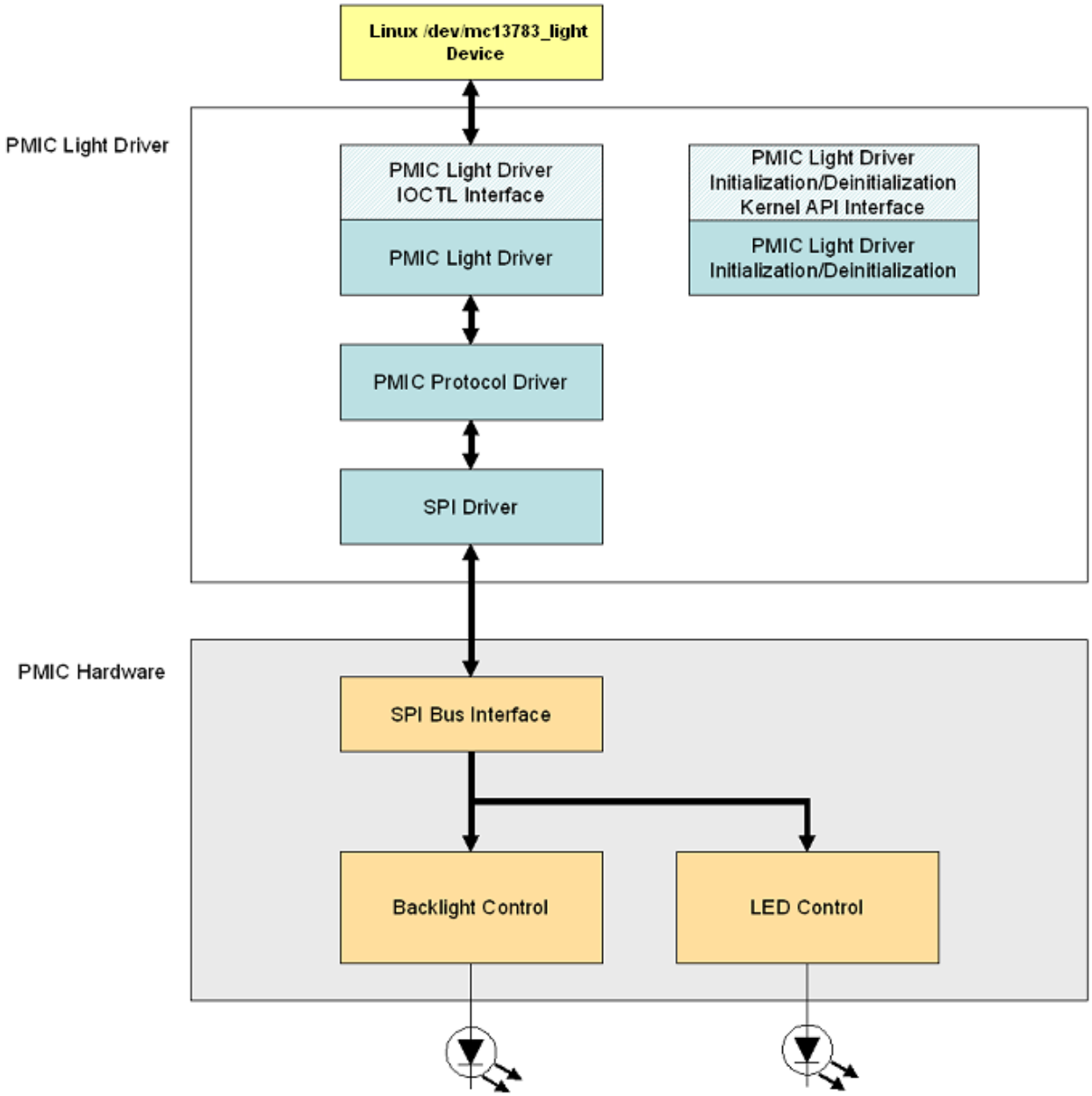

Figure 11-1. PMIC Light Driver Architecture

11.5 Driver Implementation Details

Configuring the PMIC light driver includes configuring parameters such as duty cycle, current level, ramp-up/ramp-down profiles, and so on, for the various backlight and LED circuits. The appropriate control register settings are found in the documentation for the PMIC chip.

11.5.1 Driver Initialization

To initialize this driver, open the `/dev/pmic_light` device to allow application-level access to the device driver using the IOCTL interface.

11.5.2 Driver Deinitialization

When you unload the device driver, remove the `/dev/pmic_light` device.

11.6 Driver Source Code Structure

Table 11-1 lists the source files for the MC13783-specific version of this driver that are contained in the device driver directory

```
linux/drivers/mxc/pmic/mc13783.
```

Table 11-1. MC13783 Light Driver Source Files

File	Description
pmic_light.c	Implementation of the MC13783 light client driver.
pmic_light_defs.h	Definitions for the MC13783 light client driver.

The header file for PMIC Light drivers is

```
linux/include/asm-arm/arch-mxc/pmic_light.h
```

11.7 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

1. Device Drivers-> MXC Support Drivers->MXC PMIC Support -> MC13783 Client Drivers->MC13783 Light support—Chooses the MC13783–specific version of the light driver.

11.8 Driver Unit Tests

Use the "`pmic_testapp_light`" program to test the PMIC-specific version of this driver.

```
./pmic_testapp_light
Master enable for BL and TC LED Bias
========== TESTING PMIC Light DRIVER ==========
Enter any of the following options :
1. Enable Backlight
2. Ramp up Backlight
3. Ramp down Backlight
4. Disable Backlight
5. Automated Backlight test
6. Start TC LED pattern
7. Automated FUN test
8. Start TC LED Indicator
9. Automated TCLED INDICATOR test
0. Exit
```

11.8.1 PMIC LED Tests

This test checks led control API function of PMIC light.

Select 6 to run the LED test:

```
=>6
Choose a bank:
1. Bank1
2. Bank2
3. Bank3
2
Choose a pattern - no effect on MC13783:
1. Red
2. Green
3. Blue
3
Choose a pattern:
1. BLENDED_RAMPS_SLOW
2. BLENDED_RAMPS_FAST
3. SAW_RAMPS_SLOW
4. SAW_RAMPS_FAST
5. BLENDED_BOWTIE_SLOW
6. BLENDED_BOWTIE_FAST
7. STROBE_SLOW
8. STROBE_FAST
9. CHASING_LIGHT_RGB_SLOW
10. CHASING_LIGHT_RGB_FAST
11. CHASING_LIGHT_BGR_SLOW
12. CHASING_LIGHT_BGR_FAST
12
Test Passed
```

11.8.2 PMIC Backlight Tests

This test checks backlight control API function of PMIC light.

Select 5 to run the Backlight test:

```
=>5
Backlight test:
Test PASSED
```

11.8.3 PMIC Fun Tests

This test checks fun pattern control API function of PMIC light.

Select 7 to run the Fun test:

```
=>7
Read/Write Tests
Run different pattern
Run the same pattern on the three banks
Test PASSED
```

Chapter 12 PMIC RTC

The PMIC RTC for Linux provides access to the PMIC's RTC control circuits.

This device driver makes use of the PMIC protocol driver (See Chapter "PMIC Protocol Driver") to access the PMIC hardware control registers.

12.1 PMIC Features

The PMIC chip is used for this topic:

- Real time clock control.
- Wait alarm event.

12.2 Driver Requirements

The PMIC RTC driver is a client of the PMIC protocol driver. It provides services for real time clock control of PMIC component.

12.3 Driver Software Operation

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

12.4 Driver Architecture

Figure 12-1 shows the basic PMIC RTC driver architecture along with the PMIC hardware components that are being used.

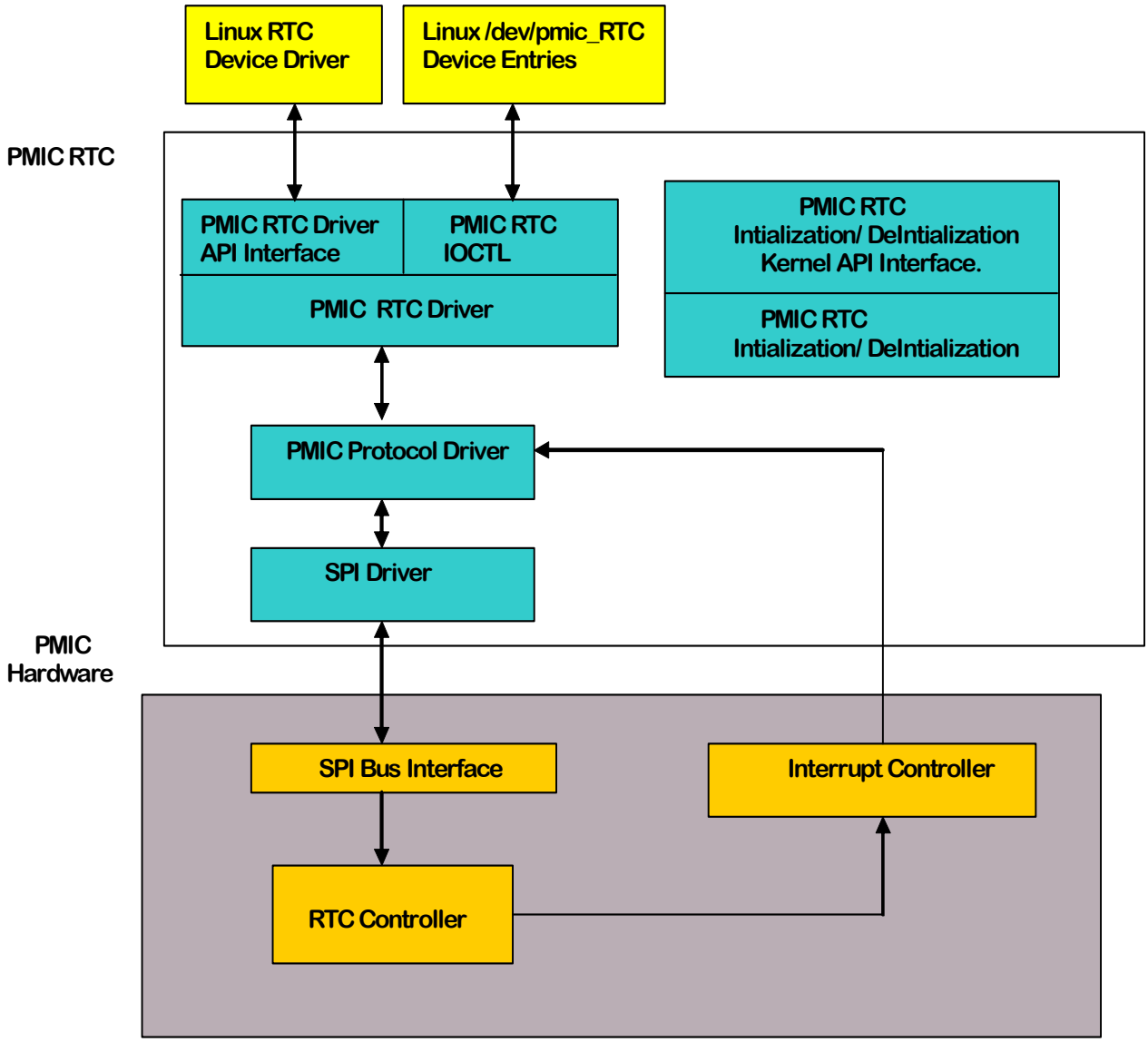

Figure 12-1. PMIC RTC Driver Architecture

12.5 Driver Implementation Details

Configuring the PMIC RTC driver includes configuring parameters such as

- Set time of day and day value

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

12.5.1 Driver Initialization

To initialize this driver, open the `/dev/pmic_rtc` device to allow application-level access to the device driver using the IOCTL interface.

12.5.2 Driver Deinitialization

When you unload the device driver, remove the `/dev/pmic_rtc` device.

12.6 Driver Source Code Structure

Table 12-1 lists the source files for the MC13783-specific version of this driver that are contained in the device driver directory

```
linux/drivers/mxc/pmic/mc13783.
```

Table 12-1. MC13783 RTC Driver Source Files

File	Description
pmic_rtc.c	Implementation of the MC13783 RTC client driver.
pmic_rtc_defs.h	Definitions for the MC13783 RTC client driver.

The header file for PMIC RTC drivers is

```
linux/include/asm-arm/arch-mxc/pmic_rtc.h
```

12.7 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

1. Device Drivers-> MXC Support Drivers->MXC PMIC Support -> MC13783 Client Drivers->MC13783 RTC support - This is the configuration option to choose the MC13783-specific version of the RTC driver.

12.8 Driver Unit Tests

Use the "`pmic_testapp_rtc`" program to test this module.

12.8.1 PMIC RTC Tests

This test checks all API function of PMIC RTC.

```
Type : ./pmic_testapp_rtc -T TEST
```

If the result is correct you can see:

```
PMIC_TestApp_rtc    0  INFO  :  Testing if PMIC_TestApp_rtc_TEST test case is OK
PMIC_TestApp_rtc    0  INFO  :  Get PMIC time
PMIC_TestApp_rtc    0  INFO  :  4412 seconds between the current time and midnight,
January 1, 1970 UTC
PMIC_TestApp_rtc    0  INFO  :  The PMIC time is : 01/01/70 01:13:32
PMIC_TestApp_rtc    0  INFO  :  Thursday, January 1, 1970 0e day of the year
PMIC_TestApp_rtc    1  PASS  :  PMIC_TestApp_rtc_TEST test case worked as expected
```

12.8.2 Time and Date Read and Write RTC Tests

This test checks all API function of Time and Date Read and Write PMIC RTC.

```
Type : ./pmic_testapp_rtc -T TIME
```

If the result is correct you can see:

```
PMIC_TestApp_rtc    0  INFO  :  Testing if PMIC_TestApp_rtc_TIME test case is OK
PMIC_TestApp_rtc    0  INFO  :  367 seconds between the current time and midnight,
January 1, 1970 UTC
PMIC_TestApp_rtc    0  INFO  :  The date and time is : 01/01/70 00:06:07
PMIC_TestApp_rtc    0  INFO  :  Thursday, January 1, 1970 0e day of the year
PMIC_TestApp_rtc    0  INFO  :  Set PMIC time
PMIC_TestApp_rtc    0  INFO  :  Get PMIC time
PMIC_TestApp_rtc    0  INFO  :  Return value of PMIC time is 368
PMIC_TestApp_rtc    1  PASS  :  PMIC_TestApp_rtc_TIME test case worked as expected.
```

12.8.3 Alarm Read and Write RTC Tests

This test checks all API function of Alarm Read and Write RTC.

```
Type : ./pmic_testapp_rtc -T ALARM
```

If the result is correct you can see:

```
PMIC_TestApp_rtc    0  INFO  :  Testing if PMIC_TestApp_rtc_ALARM test case is OK
PMIC_TestApp_rtc    0  INFO  :  372 seconds between the current time and midnight,
January 1, 1970 UTC
PMIC_TestApp_rtc    0  INFO  :  The date and time is : 01/01/70 00:06:12
PMIC_TestApp_rtc    0  INFO  :  Thursday, January 1, 1970 0e day of the year
PMIC_TestApp_rtc    0  INFO  :  Set PMIC Alarm time
PMIC_TestApp_rtc    0  INFO  :  Get PMIC Alarm time
PMIC_TestApp_rtc    0  INFO  :  Return value of PMIC Alarm time is 372
PMIC_TestApp_rtc    1  PASS  :  PMIC_TestApp_rtc_ALARM test case worked as expected
```

12.8.4 Alarm IT Test

This test checks all API function of Alarm IT.

```
Type : ./pmic_testapp_rtc -T WAIT_ALARM
```

If the result is correct you can see:

```
PMIC_TestApp_rtc    0  INFO  :  Testing if PMIC_TestApp_rtc_WAIT_ALARM test case is OK
PMIC_TestApp_rtc    0  INFO  :  Test PMIC Alarm event
```

```
PMIC_TestApp_rtc    0  INFO  :  Set PMIC time to local time
PMIC_TestApp_rtc    0  INFO  :  Set PMIC Alarm time to local time +10 second
PMIC_TestApp_rtc    0  INFO  :  Wait Alarm event...

WAIT ALARM...
****************************************
***** PMIC RTC 'Alarm IT CallBack' ******
****************************************
*** Alarm IT Pmic ***
ALARM DONE
PMIC_TestApp_rtc    0  INFO  :  *** Alarm event DONE ***
PMIC_TestApp_rtc    1  PASS  :  PMIC_TestApp_rtc_WAIT_ALARM test case worked as
expected
```

Chapter 13
Low-Level Keypad Driver

The MXC Low-Level Keypad Driver interfaces with the keypad port (KPP) in the MXC application processors. The KPP interface in these processors is provided by the hardware. The MXC Low-Level Keypad Driver is implemented as a standard Linux 2.6 keyboard driver, modified for the MXC application processors.

The MXC Low-Level Keypad Driver supports the following features:

- Interrupt-driven scan code generation for keypress and release on a keypad matrix.
- The keypad is supported as a standard input device.

The MXC Low-Level Keypad Driver can be accessed through the following device file:

```
/dev/input/event0.
```

13.1 Hardware Operation

The MXC application processors keypad device supports a keypad matrix with as many as 8 rows and 8 columns. Any pins that are not being used for the keypad are available as general purpose input/output pins.

The keypad port interfaces with a keypad matrix. On a keypress, the intersecting row and column lines are shorted together. The keypad has two mode of operation, Run mode and Low Power mode. In Run mode, the KPP detects any key press event. In Low Power mode, when there is no MCU clock, the keypad also detects any keypress event.

13.2 Software Operation

The MXC Low-Level Keypad Driver generates scancodes for keypress and release on the keypad matrix. The operation is as follows:

1. When a key is pressed on the keypad, the keypad interrupt handler is called.
2. In the keypad interrupt handler, the `mxc_kpp_scan_matrix` function is called to scan for keypresses and releases.
3. The keypad scan timer function is called every 10ms to scan for any keypress or release on the keypad.
4. The scancode for the keypress or release is generated by the `mxc_kpp_scan_matrix` function.
5. The generated scancodes are converted to input device keycodes using the `mxckpd_keycodes` array.

   Every keypress or release follows the debounce state machine which is shown in <Default_XREF_style>Figure 13-1. The `mxc_kpp_scan_matrix` function is called for every keypress and release interrupt.

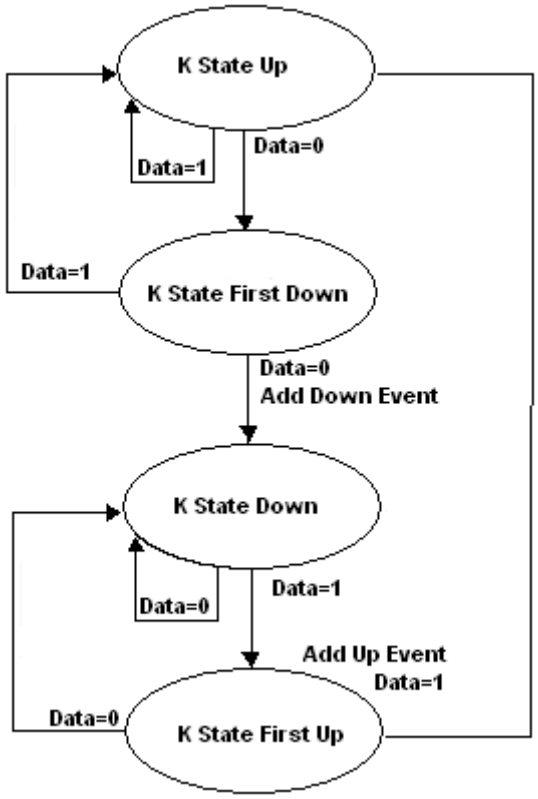

Figure 13-1. Keypad Driver State Machine

The MXC Low-Level Keypad Driver registers the input device structure within the `__init` function by calling `input_register_device(&mxckbd_dev)`.

The driver sets input bit fields and conveys to other parts of the input systems all the events that can be generated by this input device. The MXC Low-Level Keypad Driver can generate only EV_KEY type events. This can be indicated by using `__set_bit(EV_KEY, mxckbd_dev.evbit)`.

The keypress keycodes are reported by calling `input_event()`. The reported key press/release events are passed to the event interface (`/dev/input/event0`). This event interface is created when the `evdev` executable, located in `linux/input`, is compiled. The event interface is a generic input event interface. It passes the events generated in the kernel to the user space with timestamps. Blocking reads and non-blocking reads and also `select()` can be done on `/dev/input/event0`.

The structure of `input_event` is as follows:

```
struct input_event {
        struct timeval time;
        unsigned short type;
        unsigned short code;
        unsigned int value;
};
```

where:

- ‘time’ is the timestamp and returns the time at which event happened.
- ‘code’ MXC keycode for keypress or release.
- ‘value’ equals ‘0’ for key release and ‘1’ for key press.

The functions mentioned in this section are implemented as a low-level interface between the Linux OS and the KPP hardware. They cannot be called from other drivers, or from a user application.

The keypress and release scancodes can be derived using the following formula,

```
scancode (press)   = (row * 8) + col;
scancode (release) = (row * 8) + col + 128;
```

The following table describes key connection, key scancodes and key mapcodes of the keys in the keypad.

Table 13-1. Key Connections in Keypad

Key	Row	Column	Scancode	Keycode
SEL	0	0	0	161
LEFT	0	1	1	103
DOWN	0	2	2	106
RIGHT	0	3	3	108
UP	0	4	4	105
KEY2	0	5	5	88
END	0	6	6	107
BACK	0	7	7	158
KEY1	1	0	8	59
SEND	1	1	9	145
HOME	1	2	10	102
APP1	1	3	11	64
VOL_UP	1	4	12	115
APP2	1	5	13	66
APP3	1	6	14	67
APP4	1	7	15	68
KEY_3	2	0	16	4
KEY_2	2	1	17	3
KEY_1	2	2	18	2
KEY_4	2	3	19	5
VOL_DOWN	2	4	20	114
KEY_7	2	5	21	8

Table 13-1. Key Connections in Keypad (Continued)

Key	Row	Column	Scancode	Keycode
KEY_5	2	6	22	6
KEY_6	2	7	23	7
KEY_9	3	0	24	10
Number_sign	3	1	25	87
KEY_8	3	2	26	9
ASTERISK	3	3	27	53
PLUS	3	4	28	78
RECORD	3	5	29	167
KEY_Q	3	6	30	16
KEY_W	3	7	31	17
KEY_A	4	0	32	30
KEY_S	4	1	33	31
KEY_D	4	2	34	32
KEY_E	4	3	35	18
KEY_F	4	4	36	33
KEY_R	4	5	37	19
KEY_T	4	6	38	20
KEY_Y	4	7	39	21
KEY_TAB	5	0	40	15
SYMB	5	1	41	65
CAPS	5	2	42	58
KEY_Z	5	3	43	44
KEY_X	5	4	44	45
KEY_C	5	5	45	46
KEY_V	5	6	46	47
KEY_G	5	7	47	34
KEY_B	6	0	48	48
KEY_H	6	1	49	35
KEY_N	6	2	50	49
KEY_M	6	3	51	50
KEY_J	6	4	52	36
KEY_K	6	5	53	37
KEY_U	6	6	54	22

Table 13-1. Key Connections in Keypad (Continued)

Key	Row	Column	Scancode	Keycode
KEY_I	6	7	55	23
SPACE	7	0	56	57
ON/OFF	7	1	57	64
PERIOD	7	2	58	52
ENTER	7	3	59	28
KEY_L	7	4	60	38
KEY_BS	7	5	61	14
KEY_P	7	6	62	25
KEY_O	7	7	63	24

Refer to the Device-Specific Information section for additional mapcodes and scancodes.

13.3 Requirements

The Keypad driver meets the following requirements:

- The keypad driver returns the input keycode for every key that is pressed or released.
- The keypad driver implements support for an interrupt driver for keypress or release.
- The keypad driver implements support for blocking and non-blocking reads.
- The keypad driver is implemented as a standard input device.

13.4 Source Code Structure

Table 13-2 lists the source files found in the following directories:

```
linux/drivers/input/keyboard
```

Table 13-2. Keypad Source File List

File	Description
mxc_keyb.c	keypad lower level file

Table 13-3 lists the header files associated with the keypad driver, which are found in the following directory:

```
linux/drivers/input/keyboard
```

Table 13-3. Keypad Header File List

File	Description
mxc_keyb.h	keypad driver header file

13.5 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_KEYBOARD_MXC—MXC Keypad driver used for the MXC Keypad port (KPP). In the `menuconfig` this option is found under Device Drivers->Input device support->MXC Internal INPUT keypad Driver.
- CONFIG_INPUT_KEYBOARD —Create the event interface. Enabling this option creates the device node `/dev/input/event0`. In `menuconfig`, this option is found under Device Drivers->Input device support->Event interface.

There is also a source code configuration option. The following source code configurations are provided through ioctl call for this module:

— Matrix config: The keypad matrix can be configured for up to 8 rows and 8 columns. The keypad matrix configuration can be done by changing the MAXROW and MAXCOL macros defined in `include/asm-arm/arch-mxc/board-<platform>.h`.

— Debounce delay: User can configure the debounce delay by changing the variable `KScanRate` defined in `mxc_keyb.c`

13.6 Programming Interface

All MXC Low-Level Keypad Driver defines, typedefs, global variables and functions are implemented in the `mxc_keyb.c`, mxc_keyb.h.

13.7 Interrupt Requirements

Table 13-4 describes the keypad interrupt timer requirements.

Table 13-4. Keypad Interrupt and Timer Requirements

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

13.8 Example Usage

The test program follows these steps in order:

1. Open the event interface. The event interface in turn opens the MXC Low-Level Keypad Driver.
2. Read for keypress and release.
3. Handle the keypress and release events.
4. If the test is complete, key the “END” key to terminate the keypad testing.
5. Close the device.

```
Example: ./mxc_keyb_test.out
```

13.9 Unit Test

The following test cases are required to test for proper functionality of the MXC Low-Level Keypad Driver.

14.0.1 Test for keycode return, when a single key pressed on keypad.

14.0.2 Test for keycode return, when 3 keys are pressed on keypad mode.

13.10 Example Usage

The test program does the following actions:

1. Opens event interface. The event interface in turn open MXC keypad driver
2. Does a read to test the key press and key release.
3. If the testing keypad is over key, press the “END” key to terminate the keypad testing.
4. Closes the device.
5. After running the below keypad application will print 2 messages for key press and key release. For example if “1” is pressed on the keypad, the message appears as below,
6. KPP TEST APP: Pressed key is 1
7. KPP TEST APP: Pressed key is 1

```
Example: ./mxc_keyb_test.out
```

13.11 Unit Test

The following are the unit test cases required to test for proper functionality of the keypad driver.

- Test for keycode return, when a single key pressed on keypad.
- Test for keycode return, when single key is released on keycode.
- Test for keycode code return, when 3 keys pressed on keypad mode.
- Press a key for a long time. Only the pressed key will be displayed, the pressed key will not be repeated by the test application.

Chapter 14
NOR Flash MTD Driver

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

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

14.1 Hardware Operation

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

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

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

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

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

14.2 Software Operation

In a Flash-based embedded Linux system, a number of Linux technologies work together to implement a file system. Figure 14-1 illustrates the relationships between some of the standard components.

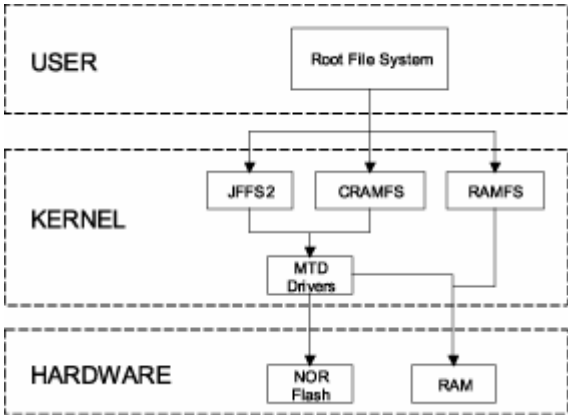

Figure 14-1. Components of a File System in a Flash-Based System

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

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

14.3 Requirements

This NOR MTD implementation meets the following requirements:

- The NOR MTD module provides necessary information for the upper layer MTD driver.
- The NOR MTD module conforms to the Linux coding standard as documented in the appendix.

14.4 Source Code Structure

From a porting perspective, only one file is added to provide functions to locate valid MTD partition table and its initialization. The files, shown in Table 14-1, are found in the following directory:

```
drivers/mtd/maps
```

Table 14-1. NOR MTD File List

File	Description
mx3ads_flash.c	MX3 ADS functions to do MTD partition

14.5 Linux Menu Configuration Options

The following Linux kernel configurations are turned on for the Linux MTD module:

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

Besides, for Intel Strata Flash, select:

```
CONFIG_MTD_CFI_INTELEXT=y
```

For Spansion Flash, select:

```
CONFIG_MTD_CFI_AMDSTD=y
```

In the `menuconfig`, these two options are under `MTD->RAM/ROM/Flash chip drivers->Support for Intel/Sharp Flash chips` and `MTD->RAM/ROM/Flash chip drivers->Support for AMD/Fujitsu Flash chips` respectively.

14.6 Programming Interface

Only two static functions are provided in this module:

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

14.7 Unit Test

This module is tested by booting Linux with the cramfs root filesystem. In addition, for a read/write test, mount a JFFS2 file system on a valid MTD partition and then do some file operations such as copy, remove, etc. Those operations should be successful.

Chapter 15
NAND Flash MTD Driver

15.1 Overview

The NAND Flash MTD (Memory Technology Devices) driver is for the NAND Flash Controller (NFC) on the MXC series processor. For the NAND MTD driver to work, only the hardware specific layer has to be implemented. The rest of the functionality such as Flash read/write/erase is automatically taken care of by the generic layer provided by the Linux MTD subsystem for NAND devices.

15.1.1 Hardware Operation

NAND Flash is a non-volatile storage device used for embedded systems. It does not support random access of memory as in case of RAM or NOR Flash. Reading or writing to NAND Flash has to be through the NFC in the MXC processors. It uses a multiplexed I/O Interface with some additional control pins. It is a sequential access device appropriate for mass storage applications. Code stored on NAND Flash can't be executed from there. It must be loaded into RAM memory and executed from there.

Memory card on the CPU_BOARD hardware contains the Samsung K9F5608 NAND Flash which is 512 bytes 8 bit NAND Flash and K9F1G08U0A NAND Flash which is 2048 bytes 8 bit NAND Flash. The manufacture id of K9F5608 is 0xEC and device id is 0x35, the manufacture id of K9F1G08U0A is 0xEC and device id is 0xf1. For more information on the NAND Flash please refer to the datasheet of K9F5608 and K9F1G08U0A NAND Flash provided by Samsung.

The NFC in the MXC processors implements the interface to standard NAND Flash devices. It provides access to both 8-bit and 16-bit NAND Flash. The NAND Flash Control block of NFC generates all the control signals that control the NAND Flash.

15.1.2 Software Operation

Memory Technology Devices (MTDs) in Linux cover all memory devices such as RAM, ROM and different kinds of NOR and NAND Flash devices. The MTD subsystem provides a unified and uniform access to the various memory devices.

There are three layers of NAND MTD driver

MTD driver

NAND: generic NAND driver

Hardware specific driver

The MTD driver just provides a mount point for the file system. It can support various file systems such as CRAMFS and JFFS2 and YAFFS.

The hardware specific driver interfaces with the integrated NFC on the MXC processors. It implements the lowest level operations on the external NAND Flash chip such as read and write. It defines the static

partitions and registers it to the kernel. This partition information is used by the upper filesystem layer. It initializes the `nand_chip` structure to be used by the generic layer.

The generic layer provides all functions, which are necessary to identify, read, write and erase NAND Flash. It supports bad block management, because blocks in a NAND Flash are not guaranteed to be good. The upper layer of the filesystem uses this feature of bad block management to manage the data on the NAND Flash.

NAND MTD driver is part of the kernel image.

For detailed information on NAND MTD driver architecture and NAND API documentation refer to http://www.linux-mtd.infradead.org/

15.2 Requirements

This NAND Flash MTD driver implementation should meet the following requirements:

The NAND MTD driver shall provide necessary hardware-specific information to the generic layer of NAND MTD driver.

The NAND MTD driver shall provide software ECC (Error Correction Code) support.

The NAND MTD driver shall support for both 16-bit and 8-bit NAND Flash

The NAND MTD driver shall conform to the Linux coding standard.

15.3 Source Code Structure

The following files need to be added for the NAND MTD driver. The files `mxcnd.c` and `mxcnd.h` are under the `drivers/mtd/nand` directory.

Table 16-1. NAND MTD File List

File	Description
mxc_nd.c	Hardware-specific layer for NAND MTD driver
mxc_nd.h	Register declaration for NAND Flash Controller

15.4 Configuration

The NAND MTD driver has the following Linux menu configuration options.

15.4.1 Linux Menu Configuration Options

The following Linux kernel configuration is provided for this module:

CONFIG_MTD_NAND_MXC - This is the configuration option for the NAND MTD driver for the MXC processors. In the `menuconfig` this option is found under Memory Technology Devices (MTD)->NAND Flash Device Drivers->MXC NAND Support.

15.5 Programming Interface

The generic NAND driver `nand_base.c` provides all functions that are necessary to identify, read, write and erase NAND Flash. The hardware-dependent functions are provided by the hardware driver `mxcnd.c`. It provides mainly the hardware access informations and functions for the generic NAND driver.

15.6 Unit Test

After the system boots up with the NAND MTD driver, it could be tested as follows:

JFFS2:

1. Create a raw file containing a `Jffs2` filesystem
   ```
   $ mkfs.jffs2 -d /etc -o etc.jffs2 -e 0x4000 --pad=0x400000
   ```
2. Erase one of the NAND Flash partitions using `flash_eraseall`
   ```
   $ flash_eraseall /dev/mtd/6
   ```
3. Write the Jffs2 image on to the NAND raw device
   ```
   $ nandwrite /dev/mtd/6 etc.jffs2
   ```
4. Mount the NAND partition and read the files
   ```
   $ mkdir -p /tmp/mtd/6
   $ mount -t jffs2 /dev/mtdblock/6 /tmp/mtd/6
   $ cd /tmp/mtd/6
   $ find .                                    # list all the files
   $ find . -type f -exec cat {} \; >/dev/null  # read all the files into /dev/null
   ```

In addition this module is tested by booting Linux with jffs2 root filesystem. In addition, for read/write test, mount a JFFS2 file system on a valid MTD partition and then do some file operations such as copy, remove, etc. Those operations should be successful. The files copied into the JFFS2 mount directory exist even after shutting down and restarting the platform.

NOTE: If either of the erase or write to NAND Flash fails, the possible reason may be that the NAND Flash on the SDR memory card is write protected. Put a jumper JP2 on the SDR Memory card. This jumper will disable the write protection of the NAND Flash.

NOTE: For booting Linux with jffs2 as root filesystem the command line argument should be changed. If jffs2 rootfilesystem is present on mtdblock 2 then the following command line should be used

```
      exec -b 0x100000 -l 0x200000 -c "noinitrd console=ttyS0,115200
root=/dev/mtdblock2 rw rootfstype=jffs2 ip=off".
```

Chapter 16
CS8900A Ethernet Driver

The MXC CS8900A Ethernet Driver interfaces CS8900A-specific functions with the standard Linux kernel network module. The CS8900-based ISA Ethernet Adapters from Cirrus Logic follow IEEE 802.3 standards, and support half or full-duplex operation in ISA bus computers on 10 Mbps Ethernet networks. The adapters are designed for operation in 16-bit ISA or EISA bus expansion slots, and are available in10BaseT-only or 3-media configurations (10BaseT, 10Base2, and AUI for 10Base-5 or fiber networks).

The MXC CS8900A Ethernet Driver has the following features:

- The efficient PacketPage Architecture can operate in I/O and memory space, and as a DMA slave.
- Supports full duplex operation.
- Supports on-chip RAM buffers for transmission and reception of frames.
- Supports programmable transmit features like automatic retransmission on collision and automatic CRC generation.
- Supports programmable receive features
  - — Support for DMA transfer
  - — Early interrupts for frame preprocessing
  - — Automatic rejection of erroneous frames
- EEPROM support for configuration.
- Supports setting of multicast-list.
- Supports MAC address setting.
- Supports obtaining statistics from the device such as transmit collisions.

This network adapter can be accessed through the `ifconfig` command with interface name (`eth0`). The probe function of this driver is declared in `drivers/net/Space.c` to probe for the device and to initialize it during boot.

16.1 Hardware Operation

The CS8900A is an Ethernet controller interfaces the system to the LAN network. It is interfaced to multimedia application processors through the Peripheral Bus Controller (PBC).

A brief overview of the device functionality is provided here. For details, see the Crystal LAN ISA Ethernet Controller data sheet.

The CS8900-based ISA Ethernet Adapters are I/O mapped devices on MXC boards. The CS8900A is a Fast Ethernet LAN controller that provides a 10 Mb/s data rate. It provides a direct interface to the ISA Local Bus. The CS8900A interfaces to the host processor by using on-chip Command and Status Registers (`CSRs`) and has internal RAM for buffer management.

The CS8900A architecture accesses its internal registers through Packetpage. The multimedia application processors have 4 kbytes of integrated RAM known as Packetpage memory. This memory is used for temporary storage of transmit and receive frames and is also used to access internal registers. This memory is divided into the following sections:

Table 16-1. Ethernet Memory Map

PacketPage Address	Contents
0000h-0045h	Bus Interface registers
0100h-013Fh	Status and control registers
0140h-014Fh	Initiate transmit registers
0150h-015Dh	Address filter registers
0400h	Receive frame location.
0A00h	Transmit frame location.

- **Transmissio**n—There are two phases of transmission. In the first phase, an Ethernet frame is moved to the chip internal buffer by issuing the transmit command. In the second phase, the frame is converted to an Ethernet packet and sent on to the network. For more information, see the Crystal LAN ISA Ethernet Controller data sheet.
- **Reception—**After receiving a packet from the analog interface, Pre-processing occurs, then temporary buffering and finally the packet is transferred to host memory. For more information, see the Crystal LAN ISA Ethernet Controller data sheet.
- **Interrupt management**—The interrupt service register (ISR) receives events through the ISQ (Interrupt service queue). The ISQ register is used to provide the host with interrupt information. Whenever an enabled interrupt is triggered, the CS8900A sets the appropriate bits in one of the five registers, and maps the contents of that register to the ISQ register. Three of the registers mapped to the ISQ are event registers: RxEvent, TxEvent and BufEvent. For more information, see the Crystal LAN ISA Ethernet Controller data sheet. Figure 16-1 shows the interrupt handling in the driver.

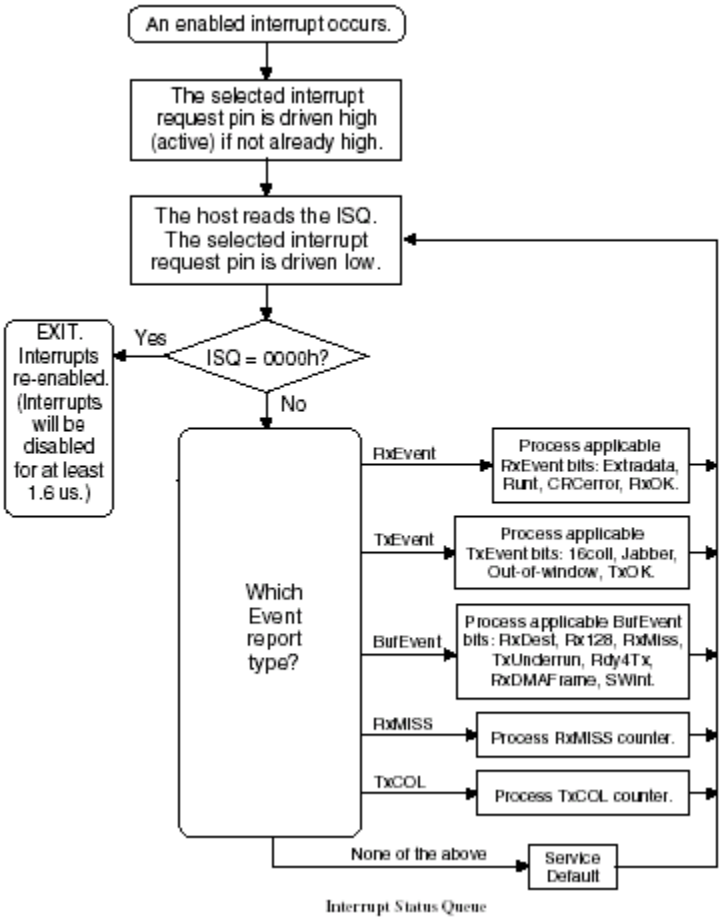

Figure 16-1. Interrupt Handling in the MXC CS8900A Ethernet Driver

For more information about IP packets on the internet, see [PKTSIZE] http://www.nlanr.net/NA/Learn/packetsizes.html:

16.2 Software Operation

The MXC CS8900A Ethernet Driver has the functions listed below.

- Module initialization—Initializes the module with the device specific structure.
- Driver entry points—Provides standard entry points for transmission, such as `net_send_packet` and for reception of Ethernet packets through the ISR, such as `net_interrupt`.
- Interrupt servicing routine—Supports events such as TxEvent, RxEvent and BufEvent.
- Miscellaneous routines—Different routines come under this category, such as `net_timeout`() for waking up network stack and `net_get_stats` to obtain statistics for the device.

16.3 Requirements

The Ethernet driver meets the following requirements:

17.0.3 The module provides all the entry points to interface with the Linux kernel 2.6 net module.

17.0.4 This Ethernet driver implements the default data configuration function to set the MAC address and interface media used in case of EEPROM failure.

17.0.5 This module follows Linux kernel coding style by Linus Torvalds. This is included in Linux distributions as the file Documentation/CodingStyle.

16.4 Source Code Structure

Table 16-2 lists the source files contained in the following directory:

```
linux/drivers/net
```

Table 16-2. Ethernet File List

File	Description
cs89x0.h	Header file defining registers.
cs89x0.c	Linux driver for Ethernet LAN controller.

For more information about the generic Linux driver, see the following source file:

```
linux-2.6.3/drivers/net/cs89x0.c
```

16.5 Linux Menu Configuration Options

The following Linux kernel configuration is provided for this module:

- CONFIG_cs89x0—Ethernet driver used for the CS8900A chip. In `menuconfig`, this option is found under Networking support-->Ethernet (10 or 100Mbit)-->CS89x0 support.

16.6 Programming Interface

Table 16-2 above shows the source files for the MXC CS8900A Ethernet Driver.The following sections show modifications that were required in the original Ethernet driver source for porting it to the MXC multimedia application processors.

16.6.1 Defines

Device-specific defines are added to the header file (`cs89x0.h`) and they provide common board configuration options.

In cs89x0.h it defines the Base address and IRQ used for CS8900A Ethernet controller on MXC multimedia application processors EVBs.

```
/* System IRQ used by CS8900A for interrupt generation taken from platform.h.*/
#define CS8900AIRQ                                              INT_CS8900A

/* I/O Base Address used access CS8900A internal registers.*/
#define CS8900A_BASE_ADDRESS  (PBC_BASE_ADDRESS + PBC_CS8900A_IOBASE+ 0x300).
```

16.6.1.1 Changes made in the source file cs89x0.c to port to linux kernel 2.6 for MXC boards.

If CS8900A is defined, the I/O memory address for probing the device on the bus is added.

```
static unsigned int netcard_portlist[] __initdata ={CS8900A_BASE_ADDRESS, 0}.
static unsigned int cs8900_irq_map[]={CS8900AIRQ, 0, 0, 0}.
```

16.6.1.2 setup_default_data

The setup_default_data function is added for the MXC multimedia application processors:

Syntax:

```
static void set_default_data(struct net_device *dev)
```

Description:

In case of EEPROM failure this routine is called to configure the MAC address and the interface media to be used.

MAC Address used ={0x0800, 0x6CA1, 0x679E}.

Parameters:

dev Pointer to network device structure.

16.7 Interrupt Requirements

Table 16-3 shows the Ethernet interrupt requirements for the MXC CS8900A Ethernet Driver.

Table 16-3. Ethernet Interrupt Requirements for MXC CS8900A

Parameter	Equation	Typical	Worst-Case

Rate	This parameter is calculated for packet Reception:
	`MBPS/(PKTsize * 8)`
	
	Here: `MBPS = 10 Mbps`
	`PKTSize(typical) = 250 Bytes (See reference [PKTSIZE])`
	`PKTSize(WorstCase) = 64 Bytes`
		5000/sec	19531/sec

Latency	(Inrenal_buffer_size * 8)/Mbps (4096*8)/10000000	3.3msec	–

16.8 Unit Test

The following items are unit test cases required to test for proper functionality of Ethernet driver for linux kernel 2.6. Following tests can be carried out using test application i.e, client server programs to send or receive test packet.

- ping OK
- ftp OK (busybox 1.00)

- dhcp OK
- telnet OK
- telnetd FAILED
- nfs OK

For more information about the material covered in this chapter, see [CS8900A]- CS8900A - Crystal LAN ISA Ethernet Controller data sheet. Also consult the following source file:

```
linux-2.6.18.1/drivers/net/cs89x0.c
```

16.9 Unit Test

The following items are unit test cases that are required to test for proper functionality of the MXC CS8900A Ethernet Driver for Linux kernel 2.6. The following tests can be carried out using a test application, that is, client server programs to send or receive test packets.

1. [UT-ETHER-1]: Test for all the entry points.
2. [UT-ETHER-1.1]: Test for `net_send_packet` entry point for transmission of packet.
3. [UT-ETHER-1.2]: Test for `net_interrupt` entry point for reception of packet.
4. [UT-ETHER-2]: Test for configuration of MAC address.

Table 16-4 gives requirements for Ethernet unit tests.

Table 16-4. Ethernet Unit Test Requiremen

Requirement ID	Test Cases
[R-ETHER-1]	[UT-ETHER-1], [UT-ETHER-1.1], [UT-ETHER-1.2]
[R-ETHER-2]	[UT-ETHER-2]

Chapter 17
Fast IrDA (FIRI) Driver

The MXC Fast IrDA (FIRI) Driver provides an interface with the Fast Infra-Red Interface (FIRI) module for MXC multimedia application processors. The Fast Infra-Red Interface (FIRI) module provides short range wireless connectivity using infra-red communication. Section 17.1, "Hardware Operation" explains the FIRI port interface in MXC processors, and gives register and memory map details. The software operation of FIRI driver for MXC processors is similar to SA1100 FIR driver in the Linux 2.6.x kernel source.

The MXC Fast IrDA (FIRI) Driver supports the following features:

- Compliant with IrDA 1.1 for Medium Infra-Red(MIR) and Fast Infrared(FIR).
- Supports 0.576 Mbps and 1.152 Mbps for MIR mode
- Supports 4 Mbps for FIR mode.
- 16-bit and 32-bit CRC generation and error detection.
- Interrupt-driven transmit and receive of data.
- SDMA capability.
- Full physical layer implementation.
- Device Destination Detection Hardware Support.
- SIP generation for collision avoidance.
- Power management.

17.1 Hardware Operation

A FIRI device is capable of establishing the following interfaces: 0.576 Mbit/sec, 1.152 Mbit/sec (or 4 Mbit/sec half-duplex), point-to-point link via LED, and IR detector. In MXC multimedia application processors it is connected to the IP Skyblue interface.

FIRI supports the following protocols:

- 0.576Mbit/sec, 1.152 Mbit/sec Medium InfraRed (MIR) physical layer protocol
- 4 Mbit/sec Fast InfraRed(FIR) physical layer protocol, and the half duplex link defined by IrDA version 1.4.
- Serial InfraRed (SIR) protocol, which supports a data rate of 115.2kbps or lower, is implemented in the UART module (see MXC UART Driver).

The FIRI interface signals are multiplexed with UART counterpart signals by using a GPIO configuration for a complete InfraRed Interface supporting SIR, MIR and FIR modes.

FIRI contains a 128-byte transmitter FIFO and a 128-byte receiver FIFO. The Transmit or Receive control registers allow the FIRI transfer rate to change to MIR or FIR, and the DMA trigger level can also be modified.

Table 17-1 provides a summary of the registers in the FIRI device.

Table 17-1. FIRI Module Memory Map

Address	Use	Access
base + 0x000	FIRI Transmit Control Register (FIRITCR)	R/W
base + 0x004	FIRI Transmit Count Register (FIRITCTR)	R/W
base + 0x008	FIRI Receive Control Register (FIRIRCR)	R/W
base + 0x00c	FIRI Transmit Status Register (FIRITSR)	R/Write one to clear
base + 0x010	FIRI Receive Status Register (FIRIRSR)	R/Write one to clear
base + 0x014	Transmitter FIFO	W
base + 0x018	Receiver FIFO	R
base + 0x01c	FIRI Control Register (FIRICR)	R/W

17.2 Software Operation

The FIRI driver is classified as a network driver. The Linux OS contains an IrDA network subsystem which implements the protocol stack for Infrared communication.The FIRI driver is interfaced to the IrDA network subsystem of the Linux kernel. The FIRI driver uses Smart DMA(SDMA) for data transfer.

The MXC FIRI driver implements standard entry points for init, exit, open, close, transmission, and reception. The driver implements the following functions:

- The init function `mxc_fir_init()`—Performs initialization and registration of the device specific structure.
- The exit function `mxc_fir_exit()`—Performs deregistration (release).
- The probe function `mxc-irda_probe()` —Performs initialization and registration of the network specific structure with the network bus protocol subsystem. The driver gets memory and IRQ resources, allocates memory for the network specific structure, initializes a buffer for IrDA, sets the supported speed, and finally registers it into the network subsystem.
- The start function `mxc_fir_start()`—The driver configures the IOMUX to enable the FIRI port, initializes the hardware, requests for IRQ, reserves DMA buffers, and allocates SDMA channels along with transfer completion routines `mxc_fir_rxdma_irq()` for reception and `mxc_fir_txdma_irq()` transmission of data. The driver opens an IrLAP layer instance which attaches this driver to the IrDA protocol stack. This function is called after the `ifconfig` command.
- The remove function `mxc_irda_remove()`—Releases resources and performs unregistration.
- The stop function `mxc_fir_stop()`—The driver and its attached resources return to their default state. DMA buffers are flushed and released, and the IRQ is released.
- The transmit function `mxc_fir_hard_xmit()`—When the user attempts to write data to the device, this function is called, and the driver writes data to the TX FIFO. If there is no room in the TX FIFO, the driver waits for the transmitter to complete the current data transfer. At the driver level, the SDMA channel is configured to transfer the filled SK buffers. Upon DMA transfer completion, the SDMA interrupt routine calls the FIRI callback routine

`mxc_fir_txdma_irq()`, which sends more data. The packet to be transmitted can contain a request to change the interface speed.

- The receive function ISR `mxc_fir_irq()`—When the FIRI device receives a data packet, an interrupt is generated and the kernel calls this function. After DMA transfer activation, the driver reads data from the RX FIFO. At the driver level, received data is read from RX FIFO by the SDMA channel and stored in SK buffers. After successful reception of a frame, the data is passed to upper layer protocols by the `netif_rx() API`. This function also looks for any receive errors, like framing or CRC errors. If there is an error in the RX FIFO the packet is ignored.

These functions are implemented as a low-level interface between the Linux OS and the FIRI hardware. They cannot be called from other drivers or from a user application.

17.3 Requirements

18.0.6 The FIRI driver provides all the entry points to interface with the Linux IrDA network subsystem.

18.0.7 The FIRI driver supports baud rates of MIR (0.576 Mbps, 1.152 Mbps) and FIR(4 Mbps).

18.0.8 The FIRI driver recognizes frame and CRC errors.

18.0.9 The FIRI driver supports power management.

18.0.10 The FIRI driver conforms to the WMSG Linux coding standards.

17.4 Source Code Structure

Table 17-2 lists the source files contained in the source directory:

```
drivers/net/irda
```

Table 17-2. FIRI File List

File	Description
mxc_fir.h	Header file defining registers.
mxc_fir.c	MXC FIRI driver.

17.5 Configuration

The FIRI port is accessed by the following command:

```
$ifconfig irda0 up
```

Note: the IP address should not be specified. IrLAP (Link Access Protocol) generates an unique address, which is used for communicating with another IrDA device. Obtain the IrLAP-generated address using irdadump. (see Section 17.8, “Unit Test” on page 17-4.)

17.5.1 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_IRDA—Build support for the IrDA protocols. In menuconfig, this option is found under Networking support -->IrDA (infrared) support --->IrDA subsystem support. By default, this option is N for all architectures.

5. CONFIG_MXC_FIR—MXC FIR driver used for the MXC FIR port. In menuconfig, this option is found under Networking support -->IrDA (infrared) support --->Infrared-port device drivers --->Freescale MXC FIR. By default, this option is N for all architectures.

17.6 Programming Interface

This driver implements all the functions that are required by the Linux IrDA network subsystem to interface with the MXC FIRI port.

17.7 Interrupt Requirements

The FIR interface generates many kinds of interrupts. The highest interrupt rate is associated with the transmit and receive interrupts. The system requirements are shown in Table 17-3

Table 17-3. FIRI Interrupt Requirements

Parameter	Equation	Typical	Worst-Case

Rate	This parameter is calculated for packet Reception:
	`BaudRate/(PacketSize * 8)`
	
	`BaudRate = 4 Mbps`
	`PacketSize(Typical) = 2048 Bytes`
	`PacketSize(Worst-Case) = 512 Bytes`
		244 /sec	976 /sec

Latency	(FIFOSize * 8)/BaudRate
	
	`FIFOSize = 128 Bytes`
	`BaudRate(Typical) = 0.576 Mbps`
	`BaudRate(Worst-Case) = 4 Mbps`
		1.778 usec	0.256 usec

17.8 Unit Test

This test example needs two boards. Before trying IrDA make sure to align the IrPorts to be facing each other at a distance of about an inch.

Below are the steps to test Fast IrDA using IrLAN on the board.

On one board:

```
modprobe irda
cp /lib/modules/2.6.18.1/kernel/net/irda/irlan/irlan.ko .
cp /lib/modules/2.6.18.1/kernel/drivers/net/irda/mxc_ir.ko .
insmod irlan.ko access=2
ifconfig irlan0 10.0.0.1 netmask 255.255.255.0 broadcast 10.0.0.255
insmod mxc_ir.ko
ifconfig irda0 up
```

```
echo 1 > /proc/sys/net/irda/discovery
ping 10.0.0.2
```

On the other board:

```
modprobe irda
cp /lib/modules/2.6.18.1/kernel/net/irda/irlan/irlan.ko .
cp /lib/modules/2.6.18.1/kernel/drivers/net/irda/mxc_ir.ko .
insmod irlan.ko access=2
ifconfig irlan0 10.0.0.2 netmask 255.255.255.0 broadcast 10.0.0.255
insmod mxc_ir.ko
ifconfig irda0 up
echo 1 > /proc/sys/net/irda/discovery
telnet 10.0.0.1
```

Chapter 18
Security Drivers

The security drivers provide several APIs that facilitate access to various security features in the processor. The Secure Controller (SCC) consists of 2 modules, a Secure RAM module and a Secure Monitor module. The SCC's Key Encryption Module (KEM) has a security feature of storing encrypted data in the on-chip RAM (Red data = plain text, Black data = encrypted), with a total size of 2 kBytes. This module is needed in cases where data must be stored securely in external memory in encrypted form. This module has a feature of clearing the secure RAM during intrusion. The system also includes:

- RNGA (Random Number Generator) which generates random numbers
- RTIC (Run-Time Integrity Checker) which hashes the data during run-time.

The security design covers the following modules:

- Boot Security.
- SCC (Secure RAM, Secure Monitor)
- Algorithm Integrity Checker
- Security Timer
- Key Encryption Module (KEM), Zeroization module
- RNGA (Random Number Generator Accelerator)
-
- RTIC (Run-Time Integrity Checker)

18.1 Hardware Security Modules

The platform has several different security blocks, and the details of the individual blocks are mentioned in the following sections.

18.1.1 Boot Security

During boot, the boot pins must be set to enable the processor to boot internally. The SCC module must be enabled by blowing specific fuses. By booting in this manner, the data in the Flash (kernel image) can be assured of integrity. Any violation in the data integrity raises an alarm.

18.1.2 SCC-Secure RAM

Figure 18-1 shows the SCC-Secure RAM and its modules. Individual modules are described in the following sections.

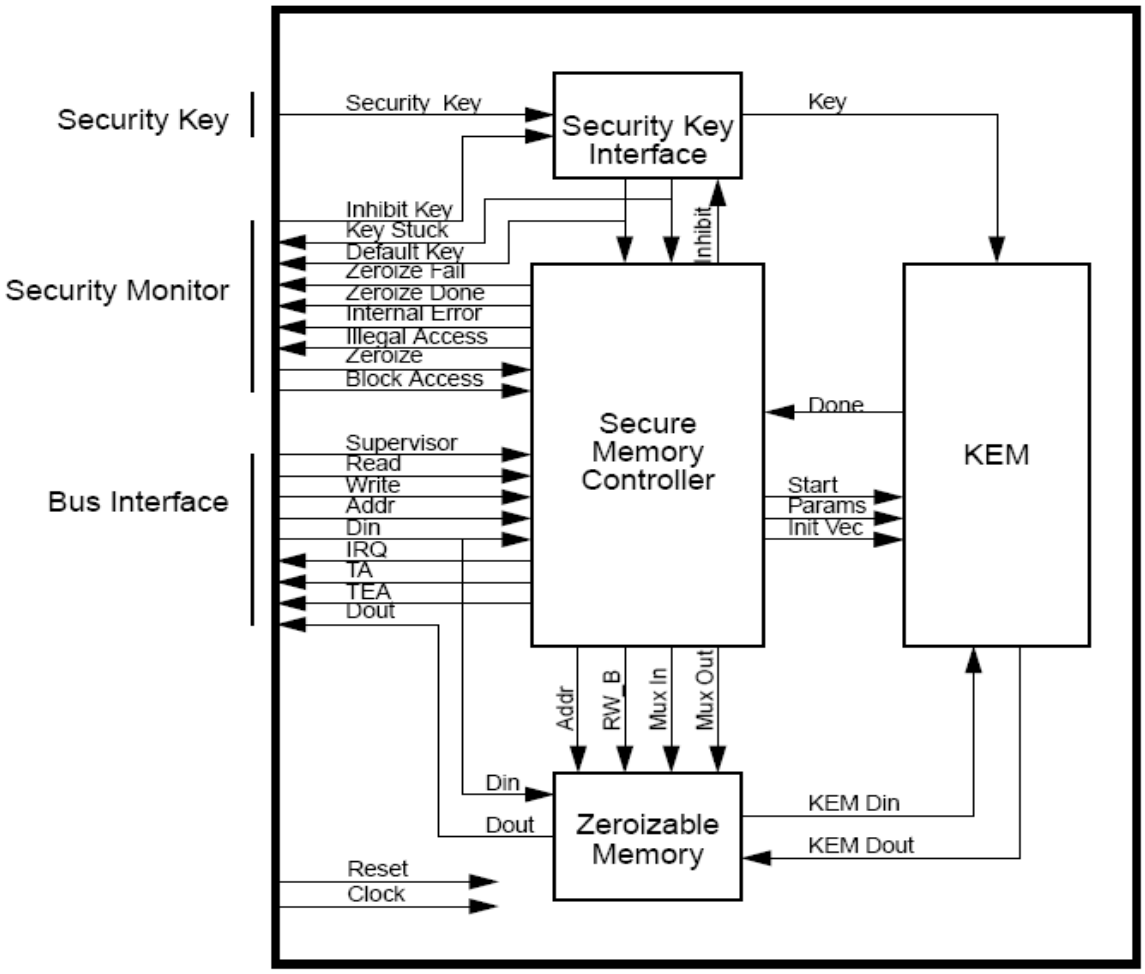

Figure 18-1. Secure RAM Block Diagram

18.1.3 SCC—Key Encryption Module (KEM)

The Key Encryption Module uses the 3DES algorithm and a 168-bit key for encryption of data. The key is programmed during manufacture and is accessible only to the encryption module. It is not accessible on any bus external to the secure memory module. The data in the external RAM is stored in an encrypted format. The data is encrypted using 3DES algorithm so that it can be decoded only using the SCC module.

18.1.4 SCC—Zeroizable Memory

The memory module can be multiplexed in and out of the RAM to allow the memory controller to switch paths according to the Secure RAM state and the host read and write accesses.When zeroing sections of memory, only the memory controller has access. When encrypting or decrypting, only the KEM module has access. When the Secure RAM is in the Idle state, then the host can access the memory. The Zeroize Done signal is also used to reset the encryption module and the memory controller. While the Zeroize Done signal is low, any attempted access by the host is ignored. When the Zeroize signal is asserted, or when the Zeroize Memory bit in the Interrupt Control register is set, not only is the Red and Black memory

initialized, but most of the registers are also reset. The Red Start, Black Start, Length, Control, Error Status, Init Vector 0, and Init Vector 1 registers are cleared. The encryption engine is also reset. The Zeroization takes place whenever there is a security violation like external bus intrusion. The Red and Black memory area is usually cleared during system boot-up.

18.1.5 SCC—Security Key Interface Module

The Security Key Interface module uses a 168-bit encryption key. The physical structures for the encryption key reside elsewhere. The Secret Key Interface contains a key mux to select between the encryption key and the default key and test the logic to determine the validity of the encryption key. In the Secure state the encryption key is used. In the Non-Secure state, the default key prevents unauthorized access to SCC-encrypted data and is useful for test purposes.

18.1.6 SCC—Secure Memory Controller

The Secure Memory controller implements an internal data handler that moves data in and out of the KEM, a memory clear function, and all of the supervisor-accessible Control and Status registers.

18.1.7 SCC—Security Monitor

The Security Monitor (SMN) is a critical component of security assurance for the platform. Specifically, it determines when and how Secure RAM resources are available to the system, and it also provides mechanisms for verifying software algorithm integrity. This block ensures that the system is running in such a manner as to provide protection for the sensitive data that is resident in the SCC. The Security Monitor consists of five main sub-blocks: The Secure State Controller, the Security Policy, the Algorithm Integrity Checker (AIC), the Security Timer and the Debug Detector.

Figure 18-2 shows a block diagram of the SMN.

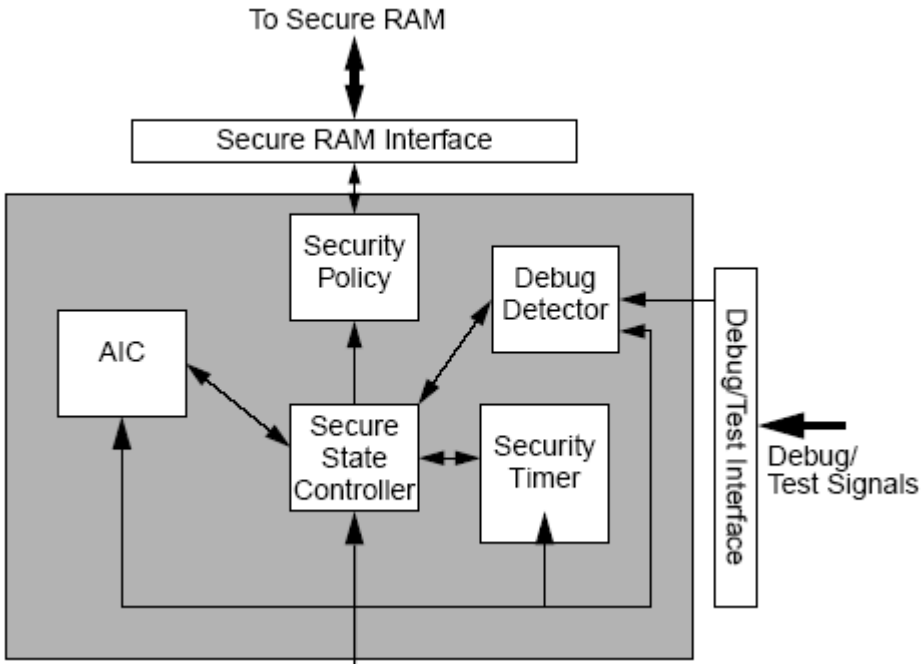

Figure 18-2. Security Monitor Block Diagram

18.1.8 SCC—Secure State Controller

The Secure State Controller, shown in Figure 18-3, is a state machine that controls the security states of the chip.

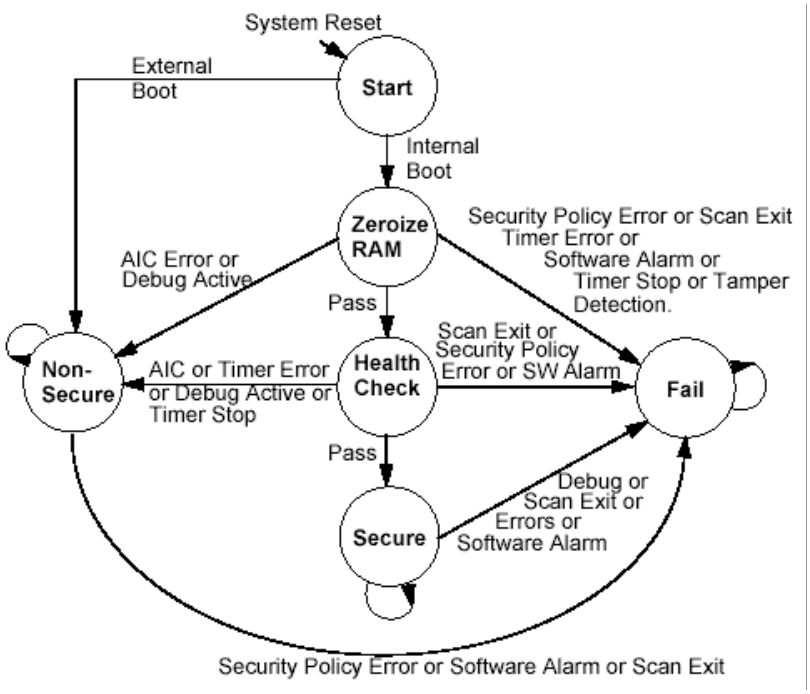

Figure 18-3. Secure State Controller State Diagram

18.1.9 SCC—Security Policy

The Security Policy block uses state information from the Secure State Controller along with inputs from the Secure RAM to determine what access to the Secure RAM is allowed based on the policy table, which is available in the corresponding platform's L3 specification document.

18.1.10 SCC—Algorithm Integrity Checker (AIC)

The Algorithm Integrity Checker (AIC) is used in conjunction with software to provide assurance that critical software (such as a software encryption algorithm) operates correctly. It is also an integral part of the power-up procedure since it must be used to achieve a secure state.

18.1.11 SCC—Secure Timer

The Secure Timer is a 32-bit programmable timer. It is used in conjunction with the Secure State Controller during power-up to ensure that the transition to the Secure state happens in the appropriate amount of time. After power-up, the timer can be used as a watchdog timer for any time-critical routines or algorithms. If the timer is allowed to expire, it generates an error.

18.1.12 SCC—Debug Detector

The Debug Detector monitors the various debug and test signals and informs the status to the Secure State Controller. The Secure State Controller gets an alert when debug modes such as JTAG and scan are active. The Debug Detector Status register can be read by the host processor to determine which debug signals are currently active.

18.1.13 RNGA (Random Number Generator Accelerator)

The RNGA module is used to generate 32-bit random numbers. This module is designed to comply with the FIPS-140 standards for randomness and non-determinism. The random bits are generated by clocking shift registers with clocks derived from ring oscillators. The configuration of the shift registers ensures statistically good data (i.e. data that looks random). The oscillators, with their unknown frequencies, provide the required entropy needed to create random data. There are different modes of operation of the RNGA:

1. Normal Mode
2. Secure Mode
3. Verification Mode
4. Oscillator Frequency Test Mode
5. Sleep Mode
6. Scan Mode

These modes can be achieved by setting appropriate bits in the RNGA set of registers. Secure Mode is functionally equivalent to normal mode. It is provided for applications requiring higher assurance. For details about how to enter the different modes, refer to the RNGA chapter in the IC documentation.

18.1.14 RTIC (Run-Time Integrity Checker)

RTIC is used to ensure the integrity of the peripheral memory contents and to assist with boot authentication. This module has the ability to check the memory contents during system boot and during run-time execution. If the memory contents at run-time fail to match the hash signature, an error in the security monitor is triggered. The RTIC communicates over two interfaces: the IP Skyblue (slave) and AHB-Lite (master). The IP-slave interface is used to read/write to the RTIC address space. The RTIC contains a DMA controller to perform reads of the peripheral memory block(s) on the AHB bus.

18.2 Software Security Modules

Besides the hardware security modules, there is optional, specialized software that helps to deliver security. This includes the RNGA (Random Number Generator) and the RTIC (Run-Time Integrity Checker) modules.

18.2.1 SCC Common Software Operations

The SCC driver is only available to other kernel modules. That is, there is no node file in `/dev`. Thus, it is not possible for a user-mode program to access the driver, and it is not possible for a user program to access the device directly.

With the exception of scc_monitor_security_failure(), all routines are synchronous, which means they will not return to their caller until the requested action completes, or fails to complete. Some of these functions could take some time to perform, depending upon the request.

Routines are provided to:

- Encrypt or decrypt secrets—`scc_crypt()`
- Trigger a security-violation alarm—`scc_set_sw_alarm()`
- Get configuration and version information—`scc_get_configuration()`
- Zero areas of memory—`scc_zeroize_memories()`
- Work on wrapped and stored secret values—`scc_alloc_slot()`, `scc_dealloc_slot()`, `scc_load_slot()`, `scc_decrypt_slot()`, `scc_encrypt_slot()`, `scc_get_slot_info()`
- Monitor the Security Failure alarm—`scc_monitor_security_failure()`
- Stop monitoring Security Failure alarm—`scc_stop_monitoring_security_failure()`
- Write registers of the SCC—`scc_write_register()`
- Read registers of the SCC—`scc_read_register()`

The driver does not allow storage of data in either the Red or Black memories. Any decrypted information is returned to the user. If the user wants to use the information at a later point, the encrypted form must again be passed to the driver, and it must be decrypted again.

The SCC encrypts and decrypts using Triple DES with an internally stored key. When the SCC is in Secure mode, it uses its secret, unique-per-chip key. When it is in Non-Secure mode, it uses a default key. This ensures that secrets stay secret if the SCC is not in Secure mode.

Not all functions that could be provided in a 'high level' manner have been implemented. Among the missing are interfaces to the ASC/AIC components and the timer functions. These and other features must be accessed through scc_read_register() and scc_write_register(), using the #define values provided.

18.2.2 Random Number Generator Accelerator (RNGA)

- The FSL SHW API is provided in both user mode and kernel mode.
- Blocking calls, and callback and non-callback non-blocking support are provided.
- Random number support allows the generation of an arbitrary number of bytes of random data, as well as the addition of additional entropy to the hardware random number generator.
- No other cryptographic functions (hashing, symmetric encryption, etc.) are supported by this driver.
- There is a debug-only interface to read/write RNG registers.

18.2.3 RTIC (Run-Time Integrity Checker)

The Run-Time Integrity Checker is intended to serve as a Hash accelerator. It has the ability to verify the memory contents during system boot (Hash-Once) and during run time (Run-Time) execution. The contents of both contiguous and non-contiguous memory blocks can be checked using the RTIC module.

The following API's can be used to access RTIC module:

- rtic_configure_mode—Configures the mode of operation of the RTIC; that is, Run-Time or Hash-Once. The parameter specifying which memory blocks need to be enabled must also be passed; that is, Memory A,B,C,D. RTIC does not support enabling multiple memory blocks that are not grouped together (that is, you cannot enable only memory blocks A and C without enabling memory B).
- rtic_start_hash—Starts the hashing process of either Run-Time or Hash-Once. If the Run-Time mode is selected then Run-Time memory registers need to be enabled before the start of hashing or vice-versa.
- rtic_configure_mem_blk—enables the configuration of the start address and block length of the memory content to be hashed. Start address indicates the starting location from where the data in the memory is to be hashed. The start address and block length should be aligned to a 4-byte boundary. The number of blocks that need to be hashed is loaded in the block count register. There are four memory blocks available. The user can configure any one of these four memory blocks by passing their appropriate address and block length to be hashed.
- rtic_hash_result—Reads the 160 bit hash result from the RTIC memory blocks A, B, C, D Hash Result Register.
- rtic_get_status—Reads the status register of the RTIC.

- rtic_get_control—Reads the control register of the RTIC.
- rtic_configure_interrupt—Enables or disables the interrupt for the RTIC module.
- rtic_get_faultaddress—Reads the fault address register of the RTIC.

If the RTIC is selected for non-interrupt based configuration (polling mode), then other operations are blocked during hashing until the hashing is done. If the RTIC is in HASH ONCE mode, it becomes idle after hashing is done. If RTIC is in RUN TIME CHECK mode, the RTIC Control register cannot be changed while the RTIC is busy. Thus, the Suspend and Resume functions are not implemented in RUN TIME CHECK mode.

18.3 Requirements

Requirements for SCC:

19.0.11 SCC provides an interface to check whether the SCC fuse is blown or not (SCC Disabled/Enabled).

19.0.12 It provides interface to configure the Red & Black memory area addresses and number of blocks to be encrypted/decrypted.

19.0.13 SCC provides an interface to load the data to be encrypted.

19.0.14 SCC provides an interface to load the data to be decrypted.

19.0.15 SCC provides an interface to start the Ciphering mechanism.

19.0.16 SCC provides an interface to report back the status of the KEM module.

19.0.17 SCC provides an interface to zero blocks in the Red/Black memory area.

19.0.18 SCC provides an interface to check for the boot type, that is, Internal or External.

19.0.19 SCC provides an interface to raise a software alarm.

19.0.20 SCC provides an interface to report back the status of the Zeroize module.

19.0.21 SCC provides an interface to configure the AIC's start and end algorithm sequence number.

19.0.22 SCC provides an interface to check the sequence of the algorithm.

19.0.23 SCC provides an interface to find the next sequence number given the current sequence number.

19.0.24 SCC provides an interface to configure the Security timer.

19.0.25 SCC provides an interface to report back the status of the Security Timer module.

Requirements for RNGA:

19.0.26 RNGA provides an interface to configure the RNGA module.

19.0.27 RNGA provides an interface to enter the initial seed number to the RNGA module.

19.0.28 RNGA provides an interface to read the random number generated from the RNGA module.

19.0.29 RNGA provides an interface to report back the status of the RNGA module.

19.0.30 RTIC provides an interface to configure the RTIC module.

19.0.31 RTIC provides an interface to data during run-time mode.

19.0.32 RTIC provides an interface to hash data during one-time mode.

19.0.33 RTIC provides an interface to report back the status of the RTIC module.

•

18.4 Source Code Structure

This section contains the various files that implement the Security modules.

Table 18-1 lists the source files associated with the Security driver. The C source files are found in the following directory:

```
linux/drivers/mxc/security
```

Header files are found in the following directory:

```
linux/include/asm/arch.
```

The RNG Driver also depends on the header files under sahara/include/

Table 18-1. SCC File List

File	Description
Makefile	Used to compile, link and generate the final binary image.
mxc_scc.c	Contains API's pertaining to SCC module's interface.
mxc_rtic.c	Contains API's pertaining to RTIC module's interface.
rng/rng_driver.c	contains the core driver.
rng/shw_driver.c	contains the shw api.
mxc_scc_driver.h	Header file related to SCC module interface.
mxc_scc_internals.h	Header file which contains definitions needed by the SCC driver. This is intended to be the file that contains most or all of the code or changes needed to port the driver.
rng/include/	contains the include files
iim.h	Header file related to IIM defines.
mxc_scc.h	Header file intended to be the file which contains all of code or changes needed to port the driver.
mxc_security_api.h	Header file that is used by the external module which needs to use the security API.

18.5 Configuration

This section provides the configurations required to execute the security system during boot-up.

18.5.1 Linux Kernel Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_MXC_SECURITY_SCC—Use the SCC module. In `menuconfig`, it is found under MXC Support drivers ->MXC Security Driver. By default, this option is Y for platform.
- CONFIG_MXC_SECURITY_RTIC—Use the RTIC module core API's. In `menuconfig`, it is found under MXC Support drivers->MXC Security Driver. By default, this option is Y for platform.
- CONFIG_RTIC_TEST_DEBUG—Debug the RTIC module. In `menuconfig`, it is found under MXC Support drivers->MXC Security Driver. By default, this option is N for platform.
- CONFIG_MXC_SECURITY_CORE—Use the security core module API's like RNGA, RTIC. In `menuconfig`, it is found under MXC Support drivers->MXC Security Driver. By default, this option is Y for platform.

18.5.2 Source Code Configuration Options

18.5.2.1 Board Configuration Option

To Configure the SCC:

1. Install Icepick and point it to license file.
2. Blow the following fuses to SCC key 0 - SCC Key 20. Please refer to the IC documentation for register details.

```
SCC Key0   = 0x77
SCC Key1   = 0xff
SCC Key2   = 0x3a
SCC Key3   = 0x76
SCC Key4   = 0x02
SCC Key5   = 0xb0
SCC Key6   = 0x0a
SCC Key7   = 0x0d
SCC Key8   = 0x90
SCC Key9   = 0x76
SCC Key10  = 0xf8
SCC Key11  = 0x07
SCC Key12  = 0x13
SCC Key13  = 0x9e
SCC Key14  = 0x36
SCC Key15  = 0xd3
SCC Key16  = 0xfa
SCC Key17  = 0x00
SCC key18  = 0x00
SCC Key19  = 0x9d
SCC Key20  = 0xfe
```

- Follow the instructions below to program the SCC key using Icepick.

1. Run Icepick
2. openSocket <IP Address of ICE>
3. initZas
4. source util_fuse_<platform>.tcl
5. init_iim
6. blow_fuse bank row bit

Step 6 command will write desired fuse. Here, the parameters passed to blow_fuse are bank, row and bit. For information about parameters to be passed refer to the appropriate platform's L3 specification.

example: The following example shows how to program the value 0x77 into SCC Key0.

```
blow_fuse 1 1 0
blow_fuse 1 1 1
blow_fuse 1 1 2
blow_fuse 1 1 4
blow_fuse 1 1 5
blow_fuse 1 1 6
```

7. sense_fuse bank row bit

   The command in Step 7 will read the desired fuse value.

- Write the following ASC Sequence in the debugger script (init_sdram.txt)

```
setmem /32 0x53FAD008 =0x00005CAA
setmem /32 0x53FAD00C =0x00002E55
setmem /32 0x53FAD010 =0x00002E55
```

- Configure the boot mode pins SW7-1 and SW7-2 to Internal Boot.

18.6 Interrupt Requirements

There are no interrupt requirements in this module, as it provides only an API interface to underlying hardware.

18.7 Usage Example

RTIC:

To hash the data from memory during run-time or one time.

To hash data in contiguous or non-contiguous Flash memory locations.

1. For Run-Time hashing of data in the memory do the following:

```
rtic_runtime_hash(start_address, block_length)
          {
               /* Pass the parameter for start_address as physical address */
                rtic_config_mem_blk(start_address, block_length, Mem Block=A/B/C/D);   /* Set
          the addr, length
                                                                    * and mem block */
                  rtic_configure_mode(mode = Run-Time, Mem Block=A/B/C/D);/* Configure the
          RTIC for mode, Run-Time or Hash-Once.*/
                  rtic_configure_interrupt(irq_enable);/* Configure the RTIC for interrupt
          enable/disable*/
                  rtic_start_hash(start hash);/* Start hashing of data*/
                  while (rtic_status() != RTIC_STAT_HASH_ERR);/* check the status for any
          error occurrence */
                 if(rtic_get_status == RTIC_STAT_HASH_DONE)
                /* HAshing for RUN-TIME is success */
                  rtic_hash_result(* hash_result);/* Read the 160 bit hash data from the
          register*/
          }
```

2. For One-Time hashing of data in the memory do the following:

```
rtic_onetime_hash(start_address, block_length)
          {
              /* Pass the parameter for start_address as physical address */
                  rtic_config_mem_blk(start_address, block_length, Mem Block = A/B/C/D);  /*
          Configure start addr,
                                                                          block length*/
                  rtic_config_mode(mode = one_time);/* Configure to one time hash mode*/
                  rtic_config_interrupt(irq_enable);/* Configure the RTIC for interrupt
          enable*/
                  rtic_start_hash(start_hash);/* Start the hashing of data*/
                  while (rtic_get_status()!=HASH_DONE);/* check the status for any error
          occurrence (if done for polling mode)*/
                  rtic_hash_result(* hash_result);/* Read the 160 bit hash data from the
          register*/
          }
```

18.8 Unit Test

SCC:

To run Unit Test for the SCC driver do the following:

- Blow fuses as mentioned in the board configuration option for the respective platforms.
- Download the test application (`scc_test.out`) and test kernel module (`scc_test_driver.ko`) onto the platform. Install the test module by typing the following command.

```
insmod scc_test_driver.ko
```

- Download and run the scripts titled `encrypt_decrypt_tests.sh`, `key_slot_tests.sh`, and `fail_mode_tests.sh` found in the `misc/test/mxc_scc` directory. These are pre-packaged, automated scripts to test various features of the SCC driver. Note that `fail_mode_tests.sh` will leave the SCC in a state in which it will do nothing other than `scc_test.out -Ls`, and the processor will have to be reset to do anything else. Type the following command before running the scripts:

```
export PATH=.:$PATH
```

- `scc_test.out` can be run without any arguments. It will retrieve the configuration from the SCC, print the values of some registers, run a small encryption, and then print the values of 'safe' registers.

```
scc_test.out -R1000           # print the SMN Status register

scc_test.out -R1000 -R10      # print SMN and Status registers

scc_test.out -W0:18           # write 18 (hex) into Red Start register

scc_test.out -S+l1024 -Le     # run ECB encryption on 1024 bytes of plaintext
```

```
scc_test.out -S+v -S+l17 -Le  # run validated&padded ECB encryption on 17 bytes

scc_test.out -Lrz             # Print registers, then zeroize memories

scc_test.out -Las            #  Set software alarm, then print 'safe' registers
```

Chapter 19
Smart Direct Memory Access (SDMA) API

19.1 Overview

SDMA API driver controls the SDMA hardware. It provides an API to other drivers for transferring data between MCU memory space, DSP memory space and peripherals. It supports the following features:

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

19.1.1 Hardware Operation

The SDMA controller is responsible for transferring data between the MCU memory space, peripherals and the DSP memory space.

- Multi-channel DMA supporting up to 32 time-division multiplexed DMA channels
- Powered by a 16-bit Instruction-Set microRISC engine
- Each channel executes specific script
- Very fast Context-Switching with 2-level priority based preemptive multi-tasking
- 4 Kbytes ROM containing startup scripts (i.e. boot code) and other common utilities that can be referenced by RAM-located scripts
- 8 Kbytes RAM area is divided into a processor context area and a code space area used to store channel scripts that are downloaded from the system memory.

19.1.2 Software Operation

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

Complete support for SDMA is provided in three layers (see Figure 3-2). The first layer is the I.API, the second layer is the Linux DMA API and the third layer is the TTY driver. The first two layers are part of the MSL and both are custom. I.API is the lowest layer and it interfaces the Linux DMA API with the SDMA controller. The Linux DMA API interfaces other drivers (e.g. MMC/SD, Sound) with the SDMA controller through the I.API.

Table 19-1 provides a list of drivers that use SDMA and the number of SDMA physical channels used by each driver. A driver can specify the SDMA channel number that it wishes to use (static channel allocation) or can have the SDMA driver provide a free SDMA channel for the driver to use (dynamic channel allocation). For dynamic channel allocation, the list of SDMA channels is scanned from channel 32 to channel 1. On finding a free channel, that channel is allocated for the requested DMA transfers.

Table 19-1. SDMA Channel Usage

Driver Name	No. of SDMA Channels	SDMA Channel Used
SDMA TTY	8	Static Channel allocation -- uses SDMA channels 1, 2, 3, 4, 5, 6, 7, 8
Unified IPC	8	Static Channel allocation -- uses SDMA channels 1, 2, 3, 4, 5, 6, 7, 8
Sound	2 per device	Dynamic channel allocation
UART	2 per device	Dynamic channel allocation
MMC	1 per device	Dynamic channel allocation
Fast IR (FIRI)	2 per device	Dynamic channel allocation
DVFS	1	Dynamic channel allocation

19.2 Source Code Structure

Table 19-2 lists the source files contained in the directory `include/asm-arm/arch-mxc`.

Table 19-2. SDMA API Header Files

File	Description
sdma.h	Header file for SDMA API

Table 19-3 lists the source files contained in the directory `arch/arm/plat-mxc/sdma`

Table 19-3. SDMA API Source files

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

Table 19-4 lists the header files contained in the directory `arch/arm/mach-mx*`

Table 19-4. SDMA Script files

File	Description
sdma_script_code.h	SDMA RAM scripts for SDMA ROM Pass 1
sdma_script_code_pass2.h	SDMA RAM scripts for SDMA ROM Pass 2

19.3 Configuration

19.3.1 Linux Menu Configuration Options

CONFIG_MXC_SDMA_API - This is the configuration option for the SDMA API driver. In the `menuconfig` this option is found under `System type and features->Freescale MXC implementations`. By default, this option is Y for all architectures.

19.4 Programming Interface

The module implements custom API and partially standard DMA API. Custom API is needed for supporting non-standard DMA features like loading scripts, interrupts handling and DVFS control. Standard API is supported partially. It can be used along with custom API functions only.

19.5 Example Usage

Refer to one of the drivers from Table 19-1 that use the SDMA API driver for a usage example.

19.6 Unit Test

Running the tests on a peripheral that uses SDMA API should test the SDMA driver. Refer to Table 19-1 for a list of drivers that use SDMA.

Chapter 20
Direct Memory Access Controller (DMAC) API

20.1 Overview

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

20.1.1 Hardware Operation

- Sixteen channels support linear memory, 2D Memory, 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 trigger 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.
- DMA request time-out errors are generated for channels that are triggered by DMA requests to interrupt the CPU when a DMA burst does not start on that channel after a programmed time count.
- Interrupts provided to the interrupt controller (and subsequently to the core) on bulk data transfer complete or transfer error.
- Each peripheral supporting DMA transfer generates a DMA_REQ signal to the DMA controller, assuming that each FIFO has a unique system address and generates a dedicated `dma_req` signal to the DMA controller. For example, a USB device with 8 end-points has 8 DMA request signals to the DMA if they all support DMA transfer.
- The DMA controller provides an acknowledge signal to the peripheral after a DMA burst is complete. This signal is sometimes used by the peripheral to clear status bits.
- Repeat data transfer function supports automatic USB host-USB device bulk/iso data stream transfer.
- Dedicated external DMA request signal.

20.1.2 Software Operation

The module provides an API for other drivers to control DMA channels. The DMA software operations mainly contain:

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

20.2 Requirements

- The module shall implement functions parallel to standard DMA API.
- The module shall conform to the Linux coding standard as documented in the appendix.

20.3 Source Code Structure

Table 20-1 lists the header files contained in the directory `include/asm-arm/arch`.

Table 20-1. DMA API Header Files

File	Description
dma.h	Header file for DMA API

Table 20-2 lists the source files contained in the directory arch/arm/

Table 20-2. DMA API Files

File	Description
mach-xxx/dma.c	Parameters of DMA channels
plat-mxc/dma_mx2.c	DMA API functions

20.4 Programming Interface

The module implements standard DMA API. Standard API is supported partially. It can be used along with custom API functions only.

Chapter 21
Real Time Clock (RTC) Driver

Each MXC processor has an integrated Real Time Clock (RTC) module. The RTC is used to keep the time and date while the system is turned off. Besides this, it can also:

- Provide periodic interrupt at certain frequency (PIE).
- Wake up the system by providing the alarm feature (AIE).

21.1 Hardware Operation

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

21.2 Software Operation

The RTC module's software implementation is through a RTC driver. Besides the initialization function, it provides `ioctl` functions to set up the RTC timer, interrupt, etc. The periodic interrupt is supported at fixed frequencies of 2Hz, 4Hz, 8Hz, 16Hz, 32Hz, 64Hz, 128Hz, 256Hz and 512Hz given the clock input of 32.768kHz (Other clock input frequencies are not supported by the driver.) The 1Hz periodic interrupt is also called "update interrupt" (UIE).

Note that MXC RTC driver implementation follows what is stated in the `rtc.txt` file under Linux kernel `Documentation` directory that "Programming and/or enabling interrupt frequencies greater than 64Hz is only allowed by root."

21.3 Requirements

This RTC implementation meets the following requirements:

- The RTC module implements all the functions required by Linux to provide the real time clock, alarm interrupt and periodic interrupt.
- The RTC module conforms to the Linux coding standard as documented in the *Coding Conventions* chapter.

21.4 Source Code Structure

The RTC module is implemented in the following files:

```
drivers/char/mxc_rtc.c
drivers/char/mxc_rtc.h
```

Table 21-1 lists the source files for the RTC.

Table 21-1. RTC File List

File	Description
mxc_rtc.c	RTC function implementations

21.5 Programming Interface

All the Linux RTC functions are implemented in the `time.c` file.

NOTE

The following file specifies all the ioctls for RTC:

```
include/linux/rtc.h
```

The following file is not used for MXC platforms:

```
drivers/char/rtc.c
```

The following RTC ioctls are supported on MXC platforms.

```
column (1) = IOCTLs listed in include/linux/rtc.h
column (2) = Supported by MXC platform RTC driver. "Y" means supported.
                                                                    (1)    (2)
RTC_UIE_ON       Y     Y
RTC_UIE_OFF      Y     Y
RTC_RD_TIME      Y     Y
RTC_SET_TIME     Y     Y
RTC_ALM_READ     Y     Y
RTC_ALM_SET      Y     Y
RTC_WKALM_RD     Y     Y
RTC_WKALM_SET    Y     Y
RTC_AIE_ON       Y     Y
RTC_AIE_OFF      Y     Y
RTC_WIE_ON       Y     -
RTC_WIE_OFF      Y     -
RTC_IRQP_READ    Y     Y
RTC_IRQP_SET     Y     Y
RTC_PIE_ON       Y     Y
RTC_PIE_OFF      Y     Y
RTC_EPOCH_READ   Y     Y
RTC_EPOCH_SET    Y     -
RTC_PLL_GET      Y     -
RTC_PLL_SET      Y     -
```

21.6 Unit Test

There is a user space test program for RTC that can be found in this release in the `misc.tar.gz` file. The source file is `rtctest.c` which is under `source/test/mxc_rtc` directory after `untar` the `misc` tarball. This file is taken from `rtc.txt` under kernel `Documentation` directory with slight modifications in order to run on MXC platforms.

There should be a file called "rtctest.out" after successful build. To run the test, download it on to the target and type ./rtctest.out. You should see the output like the following:

```
          RTC Driver Test Example.

Counting 5 update (1/sec) interrupts from reading /dev/misc/rtc: 1 2 3 4 5
Again, from using select(2) on /dev/misc/rtc: 1 2 3 4 5

Current RTC date/time is 11-9-2001, 09:39:19.
Alarm time now set to 09:39:24.
Waiting 5 seconds for alarm... okay. Alarm rang.

Periodic IRQ rate was 2Hz.
Counting 20 interrupts at:
2Hz:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
4Hz:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
8Hz:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
16Hz: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
32Hz: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
64Hz: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

                                                  *** Test complete ***

Typing "cat /proc/interrupts" will show 131 more events on IRQ 25.
```

21.7 Unit Test

Run test /unit_test/_mxc_rtc/rtctest.out. You should see the output similar to the following:

```
RTC Driver Test Example.

Counting 5 update (1/sec) interrupts from reading /dev/misc/rtc: 1 2 3 4 5
Again, from using select(2) on /dev/misc/rtc: 1 2 3 4 5

Current RTC date/time is 11-9-2001, 09:39:19.
Alarm time now set to 09:39:24.
Waiting 5 seconds for alarm... okay. Alarm rang.

Periodic IRQ rate was 2Hz.
Counting 20 interrupts at:
2Hz:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
4Hz:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
8Hz:  1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
16Hz: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
32Hz: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20
64Hz: 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19 20

                                                  *** Test complete ***

Typing "cat /proc/interrupts" will show 131 more events on IRQ 25.
```

Chapter 22
Watchdog (WDOG) Driver

The Watchdog Timer module protects against system failures by providing an escape from unexpected hang or infinite loop situations or programming errors. Some platforms may have two WDOG modules with one of them having interrupt capability.

22.1 Hardware Operation

Once the WDOG timer is activated, it must be serviced by software on a periodic basis. If servicing does not take place in time, WDOG times out. Upon a time-out, WDOG either asserts the `wdog_b` signal or a `wdog_rst_b` system reset signal, depending on software configuration. The watchdog module can not be deactivated once it is activated.

22.2 Software Operation

The Linux OS has a standard WDOG interface that allows a WDOG driver for a specific platform to be supported.

For the platforms that have two WDOG hardware modules, another implementation is done in the Machine-specific Layer as part of the `time.c` file per the requirement from the customers.

The following sections describe both implementations.

22.2.1 Generic WDOG driver

This is implemented under the `drivers/char/watchdog/mxc_wdt.c` file. It essentially provides those functions for various IOCTL and read/write calls from the user level program to control the WDOG.

22.2.1.1 Requirements

This WDOG implementation meets the following requirements:

- The WDOG module generates the reset signal if it is enabled but not serviced within a predefined timeout value.
- The WDOG module does not generate the reset signal if it is serviced within a predefined timeout value.
- The WDOG module provides IOCTL/read/write required by the standard WDOG subsystem.

22.2.1.2 Source Code Structure

The WDOG source code is:

```
drivers/char/watchdog/mxc_wdt.c and mxc_wdt.h
```

Table 22-1 lists the source files for WDOG.

Table 22-1. WDOG File List

File	Description
mxc_wdt.c	WDOG function implementations
mxc_wdt.h	header file for WDOG implementation

22.2.1.3 Programming Interface

22.2.1.4 Unit Test

There is a user space test program for WDOG that can be found in this release in the `misc.tar.gz` file. The source file is `wdt_driver_test.c` which is under `test/wdog` directory after untar the `misc` tarball. This file is taken from `watchdog.txt` under kernel `Documentation/watchdog` directory with slight modifications in order to run on MXC platforms.

There should be a file called `wdt_driver_test.out` after successful build. To run the test, download it on to the target and type `./wdt_driver_test.out` and one can see:

```
Usage: wdt_driver_test <timeout> <sleep> <test>
    timeout: value in seconds to cause wdt timeout/reset
    sleep: value in seconds to service the wdt
    test: 0 - Service wdt with ioctl(), 1 - with write()
```

To test the WDOG timeout feature with ioctl, try:

```
./wdt_driver_test.out 1 2 0 &
```

This should generate reset after a short period of time (2 seconds). If use:

```
./wdt_driver_test.out 2 1 0 &
```

The system should keep running without being reset. This test requires the kernel to be executed with the "`jtag=on`" on some platforms. For details, refer Section 22.3, "Device - Specific Information.

Chapter 23
MMC/SD/SDIO Host Driver

The MMC/SD/SDIO Host driver implements a standard Linux driver interface to the MMC/Secure Digital Host Controller (SDHC). The host driver is part of the Linux kernel's MMC framework.

The MMC driver has following features:

- MMC version 4.1 spec is supported.
- SD version 1.10 spec is supported.
- SDIO version 1.10 spec is supported.
- Hardware contains 32x16 bit in built data buffer.
- 100 Mbps Maximum hardware data rate in 4-bit mode.
- 1-bit or 4-bit operation.
- Supports card insertion and removal events.
- Supports the standard MMC commands.
- Interrupt-driven and DMA-driven data transfer.
- Power management

The driver provides the same features across all supported hardware platforms.

23.1 Hardware Operation

The MMC communication is based on an advanced 7-pin serial bus designed to operate in a low voltage range.

The MMC/SDHC module supports MMC along with SD memory and I/O functions. The Multimedia Card/Secure Digital Host module (SDHC) controls the MMC, SD memory and I/O cards by sending commands to cards and performing data accesses to/from the cards.The SD Memory Card system defines two alternative communication protocols: SD and SPI. The SDHC will only support the SD bus protocol.

The SDHC command number and SDHC command argument register allows a command to be issued to the card.

The SDHC Command and Data Control Register allows the user to specify the format of the data and the response and to control the read wait cycle.

The SDHC Block Length Register defines the number of bytes in a block (block size). Since the Stream mode of MMC is not supported, the block length must be set for every transfer.

There is an 8-bit x16-bit FIFO to store the response from the card in the SDHC. The SDHC Response FIFO Access register is used to access this FIFO. The SDHC uses two 64-byte data buffers. These buffers are used as temporary storage for data being transferred between the host system and the card, and vice versa. The SDHC Data Buffer Access Register bits hold 32-bit data upon a read or write transfer. For reception, these are the steps:

1. The SDHC controller generates an SDMA request when the FIFO is full.

2. Upon receiving this request, SDMA starts transferring data from the SDHC FIFO to system memory by reading the Data Buffer Access Register.

To transmit data, follow these steps:

3. The SDHC controller generates an SDMA request whenever the transmit FIFO is empty.
4. Upon receiving this request, the SDMA starts moving data from the system memory to the SDHC FIFO by writing to the Data Buffer Access Register for a number of pre-defined bytes.

The read-only SDHC Status Register provides SDHC operations status, application FIFO status, error conditions and interrupt status.

When certain events occur in the module, the SDHC has the ability to generate an interrupt as well as setting corresponding Status Register bits. The SDHC Interrupt Control Register allows the user to control whether these interrupts should occur.

23.2 Software Operation

The Linux OS contains an MMC bus driver which implements the MMC bus protocols. The MMC block driver handles the file system read/write calls and uses the low level MMC host controller interface driver to send the commands to SDHC. Figure 23-1 shows how the MMC-related drivers are layered.

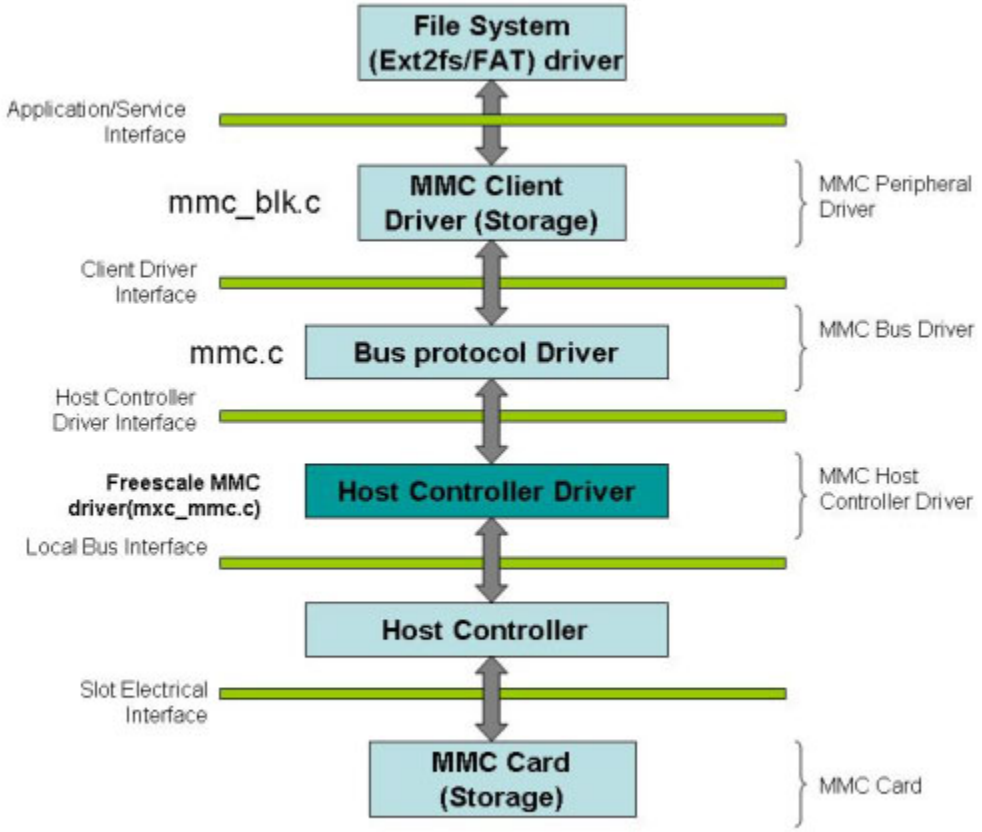

Figure 23-1. Layering of MMC drivers

The MXC MMC driver is responsible for implementing standard entry points for init, exit, request and set_ios. The driver implements the following functions:

1. The init function `mxcmci_init()`—Registers the device_driver structure.
2. The probe function `mxcmci_probe()`—Performs initialization and registration of the MMC device specific structure with MMC bus protocol driver. The driver probes for memory and IRQ resources. Configures the IOMUX to enable SDHC I/O pins and resets the hardware. Requests for IRQ and allocates DMA channel along with transfer completion routine `mxcmci_dma_irq()`.
3. `mxcmci_set_ios()`—Sets bus width, voltage level, and clock rate according to core driver requirements.
4. The `mxcmci_request()`—Handles both read and write operations. Sets up the number of blocks and block length. It configures an DMA channel, allocates safe DMA buffer and starts the DMA channel. It configures the SDHC command number register and SDHC command argument register to issue a command to the card. This function starts the SDMA and starts the clock.
5. MMC driver ISR `mxcmci_gpio_irq()`—Called when the MMC card is detected or removed.

6. MMC driver ISR `mxcmci_irq()` — Interrupt from SDHC called when command is done or errors like CRC or buffer underrun or overflow occurs.
7. DMA completion routine `mxcmci_dma_irq()` —Called after completion of a DMA transfer. It informs the MMC core driver of a request completion by calling `mmc_request_done()` API.

23.3 Requirements

24.0.41 The MMC driver provides support for multiple SDHC modules.

24.0.42 The MMC driver provides all the entry points to interface with the Linux MMC core driver.

24.0.43 The MMC driver supports MMC, SD, and SDIO cards.

24.0.44 The MMC driver recognizes data transfer errors like command time outs and CRC errors.

24.0.45 The MMC driver supports power management.

24.0.46 The MMC driver conforms to the Linux coding standards.

23.4 Source Code Structure

The iMX1 files imxmmc.c and imxmmc.h are used for MXC MMC driver development.

Table 23-1 lists the source files contained in the source directory:

```
drivers/mmc
```

Table 23-1. MMC Driver File List

File	Description
mxc_mmc.h	Header file defining registers.
mxc_mmc.c	MXC MMC driver.

23.5 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_MMC—Build support for the MMC bus protocol. In `menuconfig`, this option is found under Multimedia devices--> MMC/SD Card support. By default, this option is Y for all architectures.
- CONFIG_MMC_BLOCK—Build support for MMC block device driver, which can be used to mount the file system. In `menuconfig` ,this option is found under Multimedia devices--> MMC/SD Card support. By default, this option is Y for all architectures.
- CONFIG_MMC_MXC—MXC MMC driver used for the MXC SDHC ports. In `menuconfig`, this option is found under Multimedia devices-->MMC/SD Card support. By default, this option is Y for all architectures.

23.6 Programming Interface

This driver implements all the functions that are required by the MMC bus protocol to interface with the MXC SDHC ports.

23.7 Unit Test

The following tests are performed.

- Test for MMC card insertion event detection.
- Test for MMC card removal event detection.

BLOCK READ/WRITE TEST:

For this testing you can use `mmc_raw.sh` script found under directory `LINUX2.6/misc/scripts/`

'dd' is the command for reading/writing blocks. Note this command can corrupt the partitions and filesystem on MMC card.

Examples:

To clear the first 5 MB of the card:

```
$dd if=/dev/zero of=/dev/mmcblk0 bs=1024 count=5
```

To write a file content to the card enter the following text, substituting the name of the file to be written for `file_name`.

```
$dd if=file_name of=/dev/mmcblk0
```

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

```
$dd if=/dev/mmcblk0 of=output_file bs=1024 count=1
```

FILE SYSTEM TEST:

For the filesystem tests select appropriate Filesystem types and Partition types when configuring the kernel.

Insert an MMC or SD card.

For an SD card, the following type of information will be displayed:

mmcblk0: mmc0:cffc SD512 495488KiB
mmcblk0: p1

For an MMC card, the following type of information will be displayed:

mmcblk0: mmc:001 000000 1003520KiB
mmcblk0:p1

The following command can be used to find out the capacities of the card.

If the card is pre-formatted, this command shows the size of the card, partitions and filesystem type:

```
$fdisk -l /dev/mmcblk0
```

If the card is not formatted:

— Create the partitions on the card using the following command:

```
$fdisk /dev/mmcblk0
```

— After the partition the created files resemble the following:

```
/dev/mmcblk0p[1-4]
```

Format the card using mkfs.minix or mkfs.ext2, depending on the filesystem:

```
$mkfs.ext2 /dev/mmcblk0p1
$mkfs.minix /dev/mmcblk0p1 no_blocks
```

Mount the file system

```
$mkdir /mnt/mmc_part1
$mount -t ext2 /dev/mmcblk0p1 /mnt/mmc_part1
```

After mounting, file/directory operations can be performed in /mnt/mmc_part1

Unmount the filesystem before ejecting the card (or at least use “sync” command).

```
$umount /mnt/mmc_part1
```

Chapter 24
Inter-IC (I^2C) Driver

The MXC I^2C driver for Linux has two parts; an I^2C bus driver and an I^2C chip driver. The I^2C bus driver is a low level interface that is used to talk to the I^2C bus, while the I^2C chip driver acts as an interface between other device drivers and the I^2C bus driver.

I^2C is a two-wire, bidirectional serial bus that provides a simple, efficient method of data exchange, minimizing the interconnection between devices.

24.1 I^2C Bus Driver Overview

The I^2C bus driver is invoked only by the MXC 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 I^2C bus driver to transfer data over the I^2C bus. The chip driver uses a standard kernel space API that is provided in the Linux kernel to access the core I^2C module. The standard I^2C kernel functions are documented in the files found under `Documentation/i2c` in the kernel source tree. This bus driver supports the following features:

- Compatibility with the I^2C bus standard
- Supports bit rates up to 400 kbps
- Start and stop signal generation/detection
- Acknowledge bit generation/detection
- Interrupt-driven, byte-by-byte data transfer
- Supports standard I^2C master mode

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

24.2 I^2C Client Driver Overview

The I^2C Client Driver implements all the Linux I^2C data structures that are required to communicate with the I^2C bus driver. It exposes a custom kernel space API to the other device drivers to transfer data to their device that is connected to the I^2C bus. Internally these API functions use the standard I^2C kernel space API to call the I^2C core module. The I^2C core module looks up the MXC I^2C bus driver and calls the appropriate function in the I^2C bus driver to do the data transfer. This driver provides the following functions to other device drivers:

- A read function to read the device registers
- A write function to write to the device registers

The iMagic Camera driver would use the APIs provided by this driver to interact with the iMagic camera.

24.3 Hardware Operation

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

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

For more details see the chapter on I^2C in the hardware specification document.

24.4 Software Operation

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

24.4.1 I^2C Bus Driver Software Operation

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

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

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

24.4.2 I^2C Client Driver Software Operation

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

The MXC I^2C chip driver is registered with the I^2C core when the chip driver is loaded and unregistered when the driver is unloaded. The chip driver provides a custom kernel level API that other device drivers can use to read and write the device registers over the I^2C bus. The function provided to read the registers is called `mxc_i2c_read`, and the function to write to device registers is called `mxc_i2c_write`.

24.5 Requirements

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

- The driver supports the I^{2}C communication protocol.
- The driver supports the I^{2}C master mode of operation.
- The driver does not support the I^{2}C slave mode of operation.
- The driver provides two custom kernel level API functions that other device drivers can use to read and write the device registers that are connected to the I^{2}C bus.

24.6 Source Code Structure

Table 24-1 lists the I^{2}C bus driver source files contained in the following directory:

```
drivers/i2c/busses
```

Table 24-1. I^{2}C Bus driver Files

File	Description
mxc_i2c.c	I^{2}C bus driver source file
mxc_i2c_reg.h	Header file with I^{2}C register offsets

Table 24-2 lists the I^{2}C chip driver source files contained in the following directory:

```
drivers/i2c/chips
```

Table 24-2. I^{2}C Client driver Files

File	Description
mxc_i2c_client.c	I^{2}C chip driver source file

Table 24-3 lists the I^{2}C API header file contained in the following directory:

```
include/asm-arm/arch
```

Table 24-3. I^{2}C API header Files

File	Description
mxc_i2c.h	I^{2}C header file that other device drivers should include to access the read and write API methods

24.7 Configuration

24.7.1 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_I2C_MXC—Configuration option for the MXC I^{2}C bus driver. In `menuconfig`, this option is found under Characters->I2C support->I2C Hardware bus support. This option is dependent on the “CONFIG_I2C” option. By default, this option is Y for all architectures.
- CONFIG_MXC_I2C_CLIENT—Configuration option to choose the MXC I^{2}C chip driver. This option is dependent on the “CONFIG_I2C” option. In `menuconfig`, this option is found under

Characters->I2C support->I2C Hardware sensors chip support. By default, this option is Y for all architectures.

24.7.2 Source Code Configuration Options

24.7.2.1 Chip Configuration Options

The following chip-specific configuration options for the driver are provided in `mxc_i2c.h`:

- Number of I^{2}C modules (`I2C_NR`)—This defines the number of I^{2}C modules in the platform.
- I^{2}C Frequency Divider (`I2Cx_FRQ_DIV`)—This specifies the frequency divider to configure the clock for bit-rate selection.

The x in I^{2}Cx denotes the individual I^{2}C number. The default configuration for these are shown in Table 24-5 in the Device-Specific Information section. If the options do not vary by architecture, then a table may not be required.

24.8 Programming Interface

The MXC I^{2}C driver provides a custom kernel-space API that is used by other device drivers to read and write the registers of the device connected to the MXC I^{2}C bus.

24.9 Interrupt Requirements

The I^{2}C module generates many kinds of interrupts. The highest interrupt rate is associated with the transfer complete interrupt

Table 24-4. I^{2}C Interrupt Requirements

Parameter	Equation	Typical	Worst-Case
Rate	Transfer Bit Rate/8	25,000/sec	50,000/sec
Latency	8/Transfer Bit Rate	40us	20us

The typical value of the transfer bit-rate is 200 kbps. The worst-case is based on a baud rate of 400 kbps (max supported by the I^{2}C interface).

24.10 Usage Example

1. To read the registers of a device connected to the I^{2}C bus:

```
char *buf;
char reg[1];
unsigned int slave_addr = 0x48;

reg[0] = 0x01;
mxc_i2c_read(slave_addr, reg, 1, buf, 1);
```

2. To write to the registers of the device connected to the I^{2}C bus:

```
char buf[2];
char reg[1];
```

```
unsigned int slave_addr = 0x48;

buf[0] = 0x12;
buf[1] = 0x15;
reg[0] = 0x01;
mxc_i2c_write(slave_addr, reg, 1, buf, 2);
```

24.11 Device-Specific Information

The x in I^2Cx denotes the individual I^2C number.

Table 24-5. Default Configuration

Option	i.MX31
I2C_NR	1
I2C1_FRQ_DIV	0x17
I2C2_FRQ_DIV	N/A
I2C3_FRQ_DIV	N/A

Chapter 25
Digital Audio Multiplexer (AUDMUX) Driver

The Digital Audio Multiplexer (AUDMUX) Driver provides multiple, simultaneous interfaces between internal and external ports and peripherals. With AUDMUX, resources do not need to be hard-wired and can be effectively shared in different configurations. The AUDMUX interconnections allow multiple, simultaneous audio/voice/data flows between the ports in point-to-point or point-to-multipoint configurations.

AUDMUX includes two types of interfaces. Internal ports connect to the processor serial interfaces and external ports connect to off-chip audio devices and serial interfaces of other processors. A desired connectivity is achieved by configuring the appropriate internal and external ports.

25.1 Hardware Operation

The Digital Audio Multiplexer (AUDMUX) Driver module configures and deals with the hardware registers for the AUDMUX module.

- At most three internal ports
- Four external ports
- Full 6-wire SSI interfaces for asynchronous receive and transmit
- Configurable 4-wire (synchronous) or 6-wire (asynchronous) peripheral interfaces
- Independent Tx/Rx Frame sync and clock direction selection for host or peripheral
- Each host interface can be connected to any other host or peripheral interface in a point-to-point or point-to-multipoint (network mode)
- Transmit and Receive Data switching to support external network mode
- CE Bus network mode to provide synchronous switching on RxD

For more information, see the chapter on Audio Multiplexer in the documentation for the multimedia applications processor.

25.2 Software Operation

The AUDMUX driver is a hardware abstraction located between its client (the audio driver) and the multimedia applications processor registers.

The purpose of this low level API is only to set and read registers.

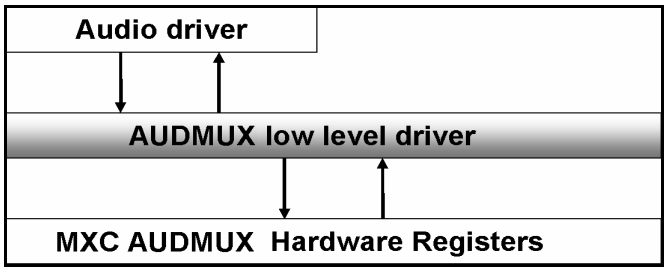

Figure 25-1. AUDMUX Driver Interactions

25.3 Requirements

The AUDMUX module's implementation meets the following requirements:

- The AUDMUX module implements each of the functions required by such a module to interface to Linux and configure all hardware registers related to this module.
- The AUDMUX module conforms to the WMSG Linux coding standards.

25.4 Source Code Structure

Table 25-1 lists the source files contained in the devices directory:

```
linux/drivers/dam
```

Table 25-1. AUDMUX Source Files

File	Description
registers.h	MXC registers definition header file
dam_types.h	Header file providing AUDMUX-specific types
dam.h	Header file providing external API
dam.c	AUDMUX version 2 registers access implementation
dam_v1.c	AUDMUX version 1 registers access implementation

25.4.1 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_MXC_DAM—Configuration option for the Digital Audio Multiplexer (AUDMUX) Driver. In `menuconfig`, this option is found under Characters->MXC Digital Audio Multiplexer support.

25.5 Programming Interface (Exported API)

The AUDMUX exported API allows the user to process standard AUDMUX operations.

Table 25-2. AUDMUX Exported Functions

Function	Description
dam_select_mode()	This function selects the operation mode of the port.
dam_select_RxClk_direction()	This function controls Receive clock signal direction for the port.
dam_select_RxClk_source()	This function controls Receive clock signal source for the port.
dam_select_RxD_source()	This function selects the source port for the RxD data.
dam_select_RxFS_direction()	This function controls Receive Frame Sync signal direction for the port.

Table 25-2. AUDMUX Exported Functions

Function	Description
dam_select_RxFS_source()	This function controls Receive Frame Sync signal source for the port.
dam_select_TxClk_direction()	This function controls Transmit clock signal direction for the port.
dam_select_TxClk_source()	This function controls Transmit clock signal source for the port.
dam_select_TxFS_direction()	This function controls Transmit Frame Sync signal direction for the port.
dam_select_TxFS_source()	This function controls Transmit Frame Sync signal source for the port.
dam_set_internal_network_mode_mask ()	This function sets a bit mask that selects the port from which of the RxD signals are to be ANDed together for internal network mode. Bit 6 represents RxD from Port7 and bit0 represents RxD from Port1. 1 excludes RxDn from ANDing. 0 includes RxDn for ANDing.
dam_set_synchronous()	This function controls whether or not the port is in synchronous mode. When the synchronous mode is selected, the receive and the transmit sections use common clock and frame sync signals. When the synchronous mode is not selected, separate clock and frame sync signals are used for the transmit and the receive sections. The default value is the synchronous mode selected.
dam_switch_Tx_Rx()	This function swaps the transmit and receive signals from (Da-TxD, Db-RxD) to (Da-RxD, Db-TxD). This default signal configuration is Da-TxD, Db-RxD.

The exact description of this API is available in the generated doxygen `api_output` directory.

25.6 Interrupt Requirements

No interrupts are generated by the Digital Audio Multiplexer.

25.7 Unit Test

ALSA test applications either in the native mode or in the OSS emulation mode should presently suffice to verify the AUDMUX driver APIs. Separate test application to test or to confirm AUDMUX driver APIs is not yet ready.

Chapter 26
Synchronous Serial Interface (SSI) Driver

The Synchronous Serial Interface (SSI) driver manages a full-duplex, serial port that allows the multimedia applications processor to communicate with a variety of serial devices. These serial devices can be standard CODECs, Digital Signal Processors (DSPs), microprocessors, peripherals, and popular industry audio codecs that implement the inter-IC sound bus standard (I2S) and Intel AC97 standard.

The SSI is typically used to transfer samples in a periodic manner. The SSI consists of independent transmitter and receiver sections with independent clock generation and frame synchronization. It supports the configuration of all SSI block registers.

26.1 Hardware Operation

SSI includes the following features:

- Independent (asynchronous) or shared (synchronous) transmit and receive sections with separate or shared internal/external clocks and frame syncs, operating in Master or Slave mode.
- Normal mode operation using frame sync.
- Network mode operation allowing multiple devices to share the port with as many as thirty-two time slots.
- Gated Clock mode operation requiring no frame sync.
- Two sets of Transmit and Receive FIFOs. Each of the four FIFOs is 8x24 bits. The two sets of Tx/Rx FIFOs can be used in Network mode to provide two independent channels for transmission and reception.
- Programmable data interface modes such like I2S, LSB, MSB aligned.
- Programmable word length (8, 10, 12, 16, 18, 20, 22 or 24 bits).
- Program options for frame sync and clock generation.
- Programmable I2S modes (Master, Slave or Normal). Over-sampling clock, `ccm_ssi_clk` available as output from SRCK in I2S Master mode.
- AC97 support.
- Completely separate clock and frame sync selections for the receive and transmit sections. In AC97 standard, the clock is taken from an external source and frame sync is generated internally.
- External `ccm_ssi_clk` input for use in I2S Master mode. Programmable over-sampling clock (SYS_CLK/`ccm_ssi_clk`) of the sampling frequency available as output in master mode at SRCK, when operated in sync mode.
- Programmable internal clock divider.
- Time Slot Mask Registers for reduced CPU overhead (for Tx and Rx both).
- SSI power-down feature.
- Programmable wait states for CPU accesses.
- IP Interface for register accesses, compliant to SRS 3.0.2 standard.

For more information, see the chapter on SSI in the multimedia applications processor documentation.

26.2 Software Operation

The SSI driver is a hardware abstraction located between its client (Audio driver) and the multimedia applications processor registers.

The purpose of this low level API is only to set and read registers. Figure 26-1 shows a block diagram of the software interaction.

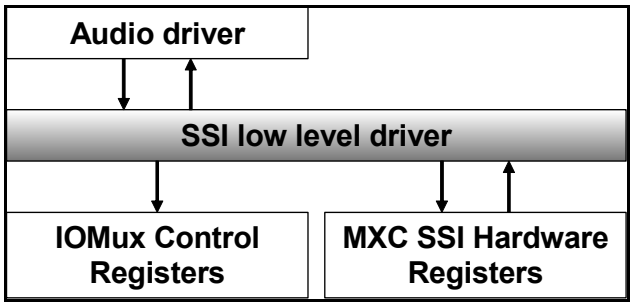

Figure 26-1. SSI Driver Interactions

26.3 Requirements

The SSI module implements each of the functions required by an SSI module to interface to Linux and configure all hardware registers related to this module.

26.4 Source Code Structure

Table 26-1 lists the source files contained in the devices directory:

```
linux/drivers/ssi
```

Table 26-1. SSI Source File List

File	Description
registers.h	MXC registers definition header file
ssi_types.h	Header file providing SSI specific types
ssi.h	Header file providing external API
ssi.c	SSI registers access implementation

26.4.1 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- CONFIG_MXC_SSI—Configuration option for the multimedia applications processor SSI driver used for the MXC SSI ports. In `menuconfig`, this option is found under Characters->MXC SSI support.

26.5 Programming Interface (Exported API)

The SSI Exported API allows the user to process standard SSI operations. The exact description of this API is available in the generated doxygen `api_output` directory.

Table 26-2. SSI Exported Functions

Function	Description
ssi_ac97_frame_rate_divider()	This function controls the AC97 frame rate divider.
ssi_ac97_get_command_address_register()	This function gets the AC97 command address register.
ssi_ac97_get_command_data_register()	This function gets the AC97 command data register.
ssi_ac97_get_tag_register()	This function gets the AC97 tag register.
ssi_ac97_mode_enable()	This function controls the AC97 mode.
ssi_ac97_tag_in_fifo()	This function controls the AC97 tag in FIFO behavior.
ssi_ac97_read_command()	This function controls the AC97 read command.
ssi_ac97_set_command_address_register()	This function sets the AC97 command address register.
ssi_ac97_set_command_data_register()	This function sets the AC97 command data register.
ssi_ac97_set_tag_register()	This function sets the AC97 tag register.
ssi_ac97_variable_mode ()	This function controls the AC97 variable mode.
ssi_ac97_write_command()	This function controls the AC97 write command.
ssi_clock_idle_state()	This function controls the idle state of the transmit clock port during SSI internal gated mode.
ssi_clock_off()	This function turns off/on the ccm_ssi_clk to reduce power consumption.
ssi_enable()	This function enables/disables the SSI module.
ssi_get_data()	This function gets the data word in the Receive FIFO of the SSI module.
ssi_get_status()	This function returns the status of the SSI module (SISR register) as a combination of status.

Table 26-2. SSI Exported Functions (Continued)

Function	Description
ssi_i2s_mode()	This function selects the I2S mode of the SSI module.
ssi_interrupt_disable()	This function disables the interrupts of the SSI module.
ssi_interrupt_enable()	This function enables the interrupts of the SSI module.
ssi_network_mode()	This function enables/disables the network mode of the SSI module.
ssi_receive_enable()	This function enables/disables the receive section of the SSI module.
ssi_rx_bit0()	This function configures the SSI module to receive data word at bit position 0 or 23 in the Receive shift register.
ssi_rx_clock_direction()	This function controls the source of the clock signal used to clock the Receive shift register.
ssi_rx_clock_divide_by_two()	This function configures the divide-by-two divider of the SSI module for the receive section.
ssi_rx_clock_polarity()	This function controls which bit clock edge is used to clock in data.
ssi_rx_clock_prescaler()	This function configures a fixed divide-by-eight clock pre-scaler divider of the SSI module in series with the variable pre-scaler for the receive section.
ssi_rx_early_frame_sync()	This function controls the early frame sync configuration.
ssi_rx_fifo_counter()	This function gets the number of data words in the Receive FIFO.
ssi_rx_fifo_enable()	This function enables the Receive FIFO.
ssi_rx_fifo_full_watermark()	This function controls the threshold at which the RFFx flag will be set.
ssi_rx_flush_fifo()	This function flushes the Receive FIFOs.
ssi_rx_frame_direction()	This function controls the direction of the Frame Sync signal for the receive section.
ssi_rx_frame_rate()	This function configures the Receive frame rate divider for the receive section.
ssi_rx_frame_sync_active()	This function controls the Frame Sync active polarity for the receive section.
ssi_rx_frame_sync_length()	This function controls the Frame Sync length (one word or one bit long) for the receive section.
ssi_rx_mask_time_slot()	This function configures the time slot(s) to mask for the receive section.

Table 26-2. SSI Exported Functions (Continued)

Function	Description
ssi_rx_prescaler_modulus()	This function configures the Prescale divider for the receive section.
ssi_rx_shift_direction()	This function controls whether the MSB or LSB will be received first in a sample.
ssi_rx_word_length()	This function configures the Receive word length.
ssi_set_data()	This function sets the data word in the Transmit FIFO of the SSI module.
ssi_set_wait_states()	This function controls the number of wait states between the core and SSI.
ssi_synchronous_mode()	This function enables/disables the synchronous mode of the SSI module.
ssi_system_clock()	This function allows the SSI module to output the SYS_CLK at the SRCK port.
ssi_transmit_enable()	This function enables/disables the transmit section of the SSI module.
ssi_two_channel_mode()	This function allows the SSI module to operate in the two channel mode.
ssi_tx_bit0()	This function configures the SSI module to transmit data word from bit position 0 or 23 in the Transmit shift register.
ssi_tx_clock_direction()	This function controls the direction of the clock signal used to clock the Transmit shift register.
ssi_tx_clock_divide_by_two()	This function configures the divide-by-two divider of the SSI module for the transmit section.
ssi_tx_clock_polarity()	This function controls which bit clock edge is used to clock out data.
ssi_tx_clock_prescaler()	This function configures a fixed divide-by-eight clock prescaler divider of the SSI module in series with the variable prescaler for the transmit section.
ssi_tx_early_frame_sync()	This function controls the early frame sync configuration for the transmit section.
ssi_tx_fifo_counter()	This function gets the number of data words in the Transmit FIFO.
ssi_tx_fifo_empty_watermark()	This function controls the threshold at which the TFEx flag will be set.
ssi_tx_fifo_enable()	This function enables the Transmit FIFO.
ssi_tx_flush_fifo()	This function flushes the Transmit FIFOs.
ssi_tx_frame_direction()	This function controls the direction of the Frame Sync signal for the transmit section.

Table 26-2. SSI Exported Functions (Continued)

Function	Description
`ssi_tx_frame_rate()`	This function configures the Transmit frame rate divider.
`ssi_tx_frame_sync_active()`	This function controls the Frame Sync active polarity for the transmit section.
`ssi_tx_frame_sync_length()`	This function controls the Frame Sync length (one word or one bit long) for the transmit section.
`ssi_tx_mask_time_slot()`	This function configures the time slot(s) to mask for the transmit section.
`ssi_tx_prescaler_modulus()`	This function configures the Prescale divider for the transmit section.
`ssi_tx_shift_direction()`	This function controls whether the MSB or LSB will be transmitted first in a sample.
`ssi_tx_word_length()`	This function configures the Transmit word length.

26.6 Interrupt Requirements

The SSI module generates interrupts but this driver is only a hardware abstraction. The interrupt requirements depends on the client which will use the API.

26.7 Unit Test

ALSA test applications either in the native mode or in the OSS emulation mode should presently suffice to verify the SSI driver APIs. Separate test application to test or to confirm SSI driver APIs is not yet ready.

Chapter 27
Configurable Serial Peripheral Interface (CSPI) Driver

The Configurable Serial Peripheral Interface (CSPI) driver implements a standard Linux driver interface to MXC CSPI Controllers. It is based on SPI Framework by David Brownell. It supports the following features:

- Interrupt-driven transmit/receive of bytes
- Supports multiple master controller interface
- Supports the multiple slaves select
- Supports multi-client requests

27.1 Hardware Operation

CSPI is used for fast data communication with fewer software interrupts than conventional serial communications. Each CSPI is equipped with data FIFO and is a master/slave configurable serial peripheral interface module, allowing the processor to interface with external SPI master or slave devices.

The primary features of the CSPIs include:

- Master/slave configurable
- Two chip selects allowing maximum of 4 different slaves each for master mode operation
- Up to 32-bit programmable data transfer
- 8 by 32-bit FIFO for both Tx and Rx data
- Polarity and phase of the Chip Select (SS) and SPI Clock (SCLK) are configurable

27.2 Software Operation

27.2.1 SPI sub-system in Linux

The CSPI driver layer is located between the client layer (PMIC is one of the clients) and the MXC hardware access layer.

Figure 27-1 shows the block diagram for SPI subsystem in Linux.

SPI requests always go into I/O queues. Requests for a given SPI device are always executed in FIFO order, and complete asynchronously through completion callbacks. There are also some simple synchronous wrappers for those calls, including ones for common transaction types like writing a command and then reading its response.

Each SPI client must have a protocol driver associated with them. And those must all be sharing the same controller driver. Only controller driver can interact with the underlying SPI hardware module.

Figure 27-2 shows how the different SPI drivers are layer in the SPI subsystem.

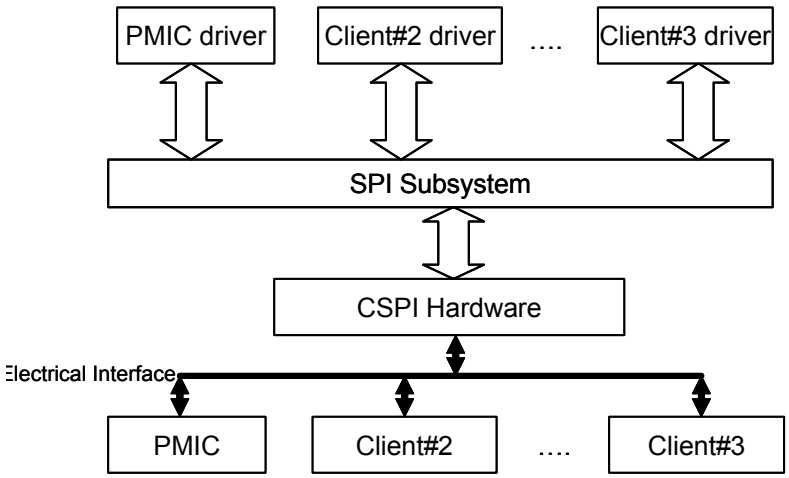

Figure 27-1. SPI Sub-system

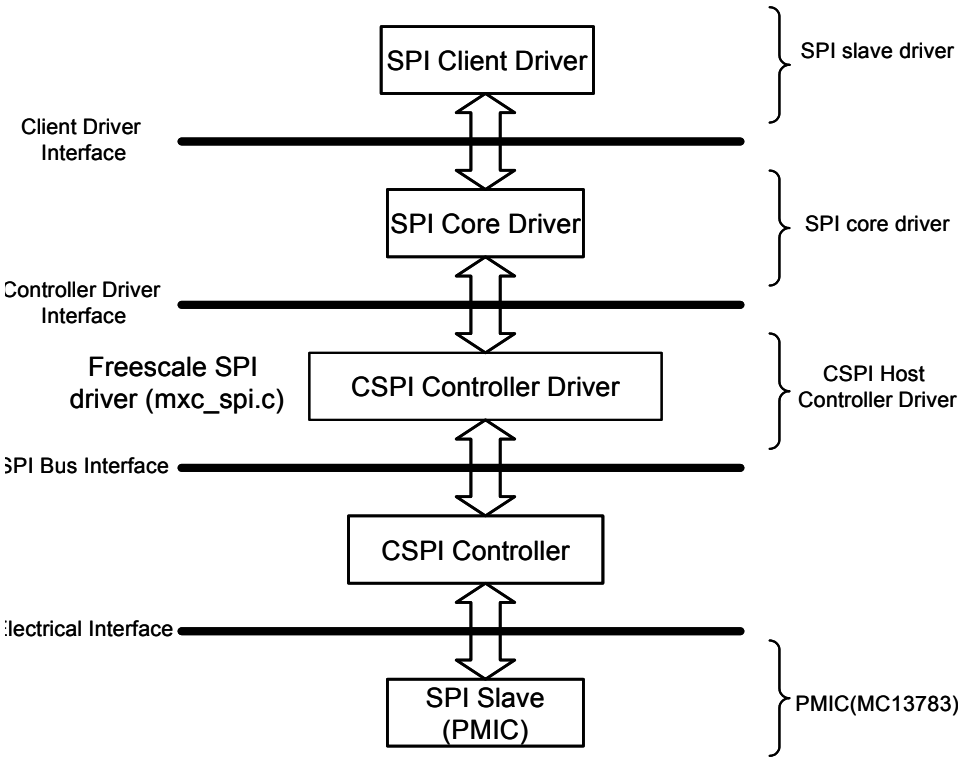

Figure 27-2. Layering of SPI drivers in SPI subsystem

27.2.2 Limitations

1. It does not have SPI Slave logic implementation yet.
2. It does not support single client connected to multiple masters.

3. It presently does not implement user space interface with the help of device node entry but supports "sysfs" interface.

27.2.3 Standard Operations

The CSPI driver is responsible for implementing standard entry points for init, exit, chip select and transfer. The driver implements the following functions:

1. The init function `mxc_spi_init( )` – Registers the `device_driver` structure.
2. The probe function `mxc_spi_probe( )` - Performs initialization and registration of the SPI device specific structure with SPI core driver. The driver probes for memory and IRQ resources. Configures the IOMUX to enable CSPI I/O pins, requests for IRQ and resets the hardware.
3. The chip select function `mxc_spi_chipselect( )` - Configures the hardware CSPI for the current SPI device. Sets the word size, transfer mode, data rate for this device.
4. SPI transfer function `mxc_spi_transfer( )` – Handles data transfers operations.
5. SPI setup function `mxc_spi_setup( )` – Initialize the current SPI device.
6. SPI driver ISR `mxc_spi_isr( )` – Called when the data transfer operation is completed and an interrupt is generated.

27.2.4 CSPI synchronous operation

Figure 27-3 shows how CSPI provides synchronous read/write operations.

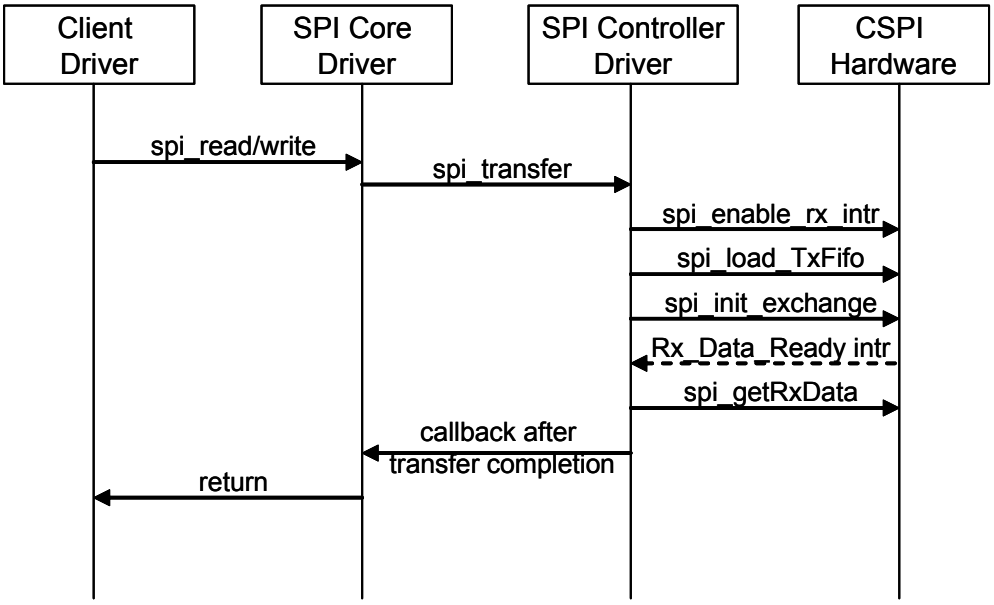

Figure 27-3. CSPI synchronous operation

27.2.5 PMIC access

Figure 27-4 shows the how PMIC can be accessed through the SPI subsystem.

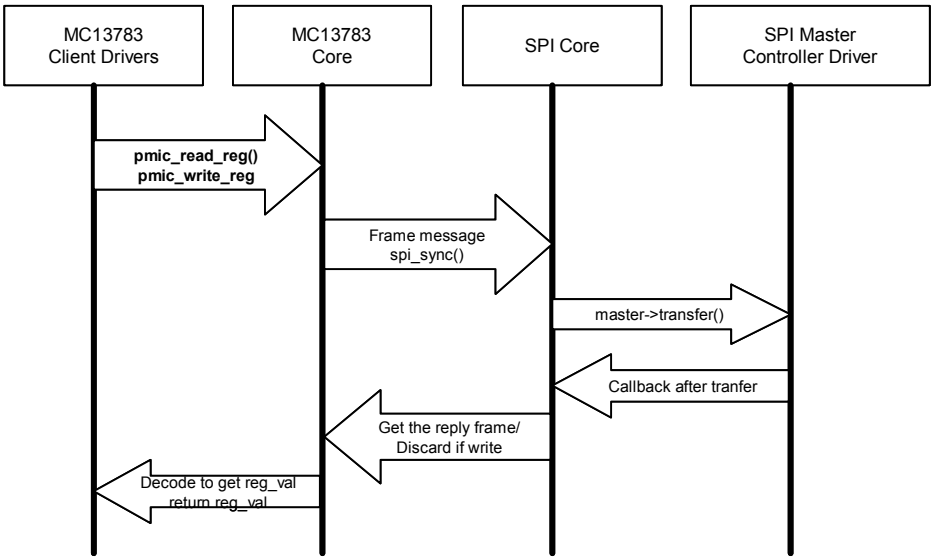

Figure 27-4. PMIC access through SPI

27.3 Requirements

The CSPI module implements the following requirements:

1. .It implements each of the functions required by a CSPI module to interface to Linux.
2. It provides support for multiple SPI master controllers.
3. It provides support to handle multi-client synchronous requests.

27.4 Source Code Structure

Table 27-1 lists the source files contained in the devices directory:

```
linux/drivers/spi
```

Table 27-1. CSPI Source File List

File	Description
mxc_spi.h	Header file defining registers.
mxc_spi.c	Freescale SPI Master Controller driver

27.5 Configuration

The following Linux kernel configurations are provided for this module:

- CONFIG_SPI: Build support for the SPI core. In menuconfig, this option is found under Device Drivers->SPI Support->SPI Support.

- CONFIG_SPI_BITBANG: This is library code, and is automatically selected by drivers that need it. SPI_MXC selects it. In menuconfig, this option is found under Device Drivers->SPI Support-> Bitbanging SPI master
- CONFIG_SPI_MXC: This implements the SPI master mode for MXC CSPI. In menuconfig, this option is found under Device Drivers->SPI Support->MXC CSPI controller as SPI Master.
- CONFIG_SPI_MXC_SELECTn: This is to select the CSPI hardware modules into the build(where n = 1, 2, or 3). In menuconfig, this option is found under Device Drivers->SPI Support->CSPIn.
- CONFIG_SPI_MXC_TEST_LOOPBACK: This is to select the enable testing of CSPIs in loop back mode. In menuconfig, this option is found under Device Drivers->SPI Support->LOOPBACK Testing of CSPIs. By default this is disable as it is intended to use only for testing purposes.

27.6 Programming Interface

This driver implements all the functions that are required by the SPI core to interface with the CSPI hardware.

27.7 Interrupt Requirements

The SPI interface generates interrupts. CSPI interrupt requirements are shown in Table 27-2.

Table 27-2. CSPI Interrupt Requirements

Parameter	Equation	Typical	Worst-Case
BaudRate / Transfer Length	(BaudRate / (TransferLength)) * (1 / Rxtl)	31250	1500000

The typical values are based on a baud rate of 1 Mbps with a receiver trigger level (Rxtl) of 1 and a 32-bit transfer length. The worst-case is based on a baud rate of 12 Mbps (max supported by the SPI interface) with a 8-bits transfer length.

27.8 Unit Test

The source code for the unit test applications are available at

```
LINUX2.6/misc/test/mxc_spi_test/
```

The built unit test application for SPI tests is

```
/unit_tests/mxc_spi_test/mxc_spi_test1.out
```

The built unit test modules for SPI test is

```
/unit_tests/modules/mxc_spi_testmod.ko
```

This test send up to 32 bytes to a specific SPI device. SPI writes data received data from the user into Tx FIFO and waits for the data in the Rx FIFO. Once the data is ready in the Rx FIFO, it is read and sent to user. Test is considered successful if data received matches with the data sent.

NOTE: As this test is intended to test the SPI device, it is always configured in loop back mode reasons is that it is dangerous to send random data to SPI slave devices. This requires the kernel image to be rebuilt

with CONFIG_SPI_MXC_TEST_LOOPBACK (Refer Section 27.5, "Configuration)enabled in the menuconfig. Also the test module mxc_spi_testmod.ko must be inserted before testing.

To run the SPI test:

```
Options: ./mxc_spi_test1.out <spi_no> <nb_bytes> <value>
<spi_no> - CSPI Module number in [0, 1, 2]
<nb_bytes> - No. of bytes: [1-32]
<value> - Actual value to be sent

Example:
./mxc_spi_test/mxc_spi_test1.out 0 9 FreeScale
Execute data transfer test: 0 9 FreeScale
Data sent : FreeScale
Data received : FreeScale
Test PASSED.
```

27.9 Device - Specific Information

Table 27-3 gives the number of CSPI controllers in different MXC platforms.

Table 27-3. CSPI Controllers in MXC Platforms

Platforms (SOC)	No. of CSPI Controllers
iMX31	3

Chapter 28 MX31 Low-level Power Management Driver

28.1 Overview

The purpose of this document is to describe the design of the low-level PM driver for the i.MX31 platform. This driver implements Dynamic Frequency Scaling (DFS) techniques and low-power modes. Dynamic Voltage and Frequency Scaling (DVFS) is described in a different chapter.

DFS is used to change the frequency when the Dynamic Power Management (DPM) level decides to change the operating point in order to meet the power requirements. This is done when the system is in RUN mode to conserve power. Low-power modes such as WAIT, DOZE, STOP and DSM are implemented in order to save power. In all these cases, power consumption is achieved by reducing the frequency and increasing the severity of clock gating.

28.1.1 Hardware Operation

The DFS operation and low-power modes on the MCU side are controlled by software using the Clock Controller Module (CCM).

Features of CCM:

- PLL Control
- Dynamic Frequency Change (DFS) - using dividers to change core frequency on the fly and PLL scaling to lock PLL
- Clock gating for various modules during low-power modes
- Low-power modes

28.1.2 Software Operation

For DFS operation, software is responsible for setting the desired frequency of ARM, AHB (MAX clock) and IP using either PLL scaling or using integer scaling (dividers). Core frequency depends on MAX clock and the PLL clock. In case of changing frequency using dividers, software should set the desired divider values in the divider register to enable frequency change. In case of PLL scaling, software should set the desired frequency value using PDF, MFD, MFN registers in the CCM. For WAIT, DOZE, STOP and DSM low-power modes, software should disable interrupts before executing a wait-for-interrupt (WFI) instruction and re-enable interrupts afterwards.

28.2 Requirements

The Low-level PM driver API requires DPM to make the appropriate calls and pass the required arguments. The MCU clock domain is partitioned into four synchronous clocks and two sub-domains. The main clock of this domain is called mcu_main_clk, and it is the output of the mcu clock switch unit.

- mcu_clk (ipg_clk_arm) is the clock of the ARM platform. The target frequency of this clock is 532 MHz. This clock is generated from MCU BRM with a division factor as defined by the BRMM bits in PDR0.
- max_clk sub-domain (ipg_clk_ahb) is the clock domain of the internal ARM platform peripherals like the cross bar switch and chip modules. Clocks in this domain are generated from the max postdivider with a division factor as defined by the MAX_PDF bits in PDR0 register. These clocks should be an integer multiple (value of between 1 and 8) of the mcu_main_clk. Maximum target frequency of these clocks is 133MHz.
- hsp_clk is the clock for the IPU. This clock is generated from the hsp postdivider with a division factor as defined by the HSP_PDF bits in PDR0 register. These clocks should be an integer multiple (value of between 1 and 8) of the mcu_main_clk. Maximum target frequency of this clock is 133MHz for 1.2V supply.
- ipg_clk sub-domain is the clock domain of certain parts of the IP peripherals. These clocks are generated from the ipg postdivider with a division factor defined by the IPG_PDF bits in the PDR0 register. These clocks should be an integer multiple (either 1 or 2) of the max_clk. Maximum target frequency of these clocks is 62.5MHz.
- nfc_clk (ipg_clk_nfc_20m) is the clock for Nand Flash controller. This clock is generated from the nfc postdivider with a division factor as defined by the NFC_PDF bits in the PDR0 register.
- ckil_mcu_sync_ipg is the clock for the peripheral modules. They require a 32KHz clock
- ipg_clk_gacc_mbx_clk is the clock for the MBX module. It is 1/2 of the ipg_ahb_clk, which is 66 MHz.

28.3 Hardware Issues

DSM mode does not work and may be fixed in the future releases or silicon revs.

28.4 Source Code Structure

Table 28-1 lists the source files for i.MX31 contained in the directory linux/arch/arm/mach-mx3

Table 28-1.

File	Description
mxc_pm.c	Source file with all the implementation
crm_regs.h	Header File with all register and bit definitions for CCM module

Table 28-2 lists the header file associated with low-level PM driver located in include/asm-arm/arch-mxc

Table 28-2.

File	Description
mxc_pm.h	PM header file that contains the API declaration

28.5 Programming Interface

The following set of APIs is currently provided for frequency scaling and low-power modes.

1. mxc_pm_intscale
   - This is used to perform all the steps required to enable scaling on the fly using PCDR0 divider
   - DPM passes the required Core, AHB and IPG frequency.
2. mxc_pm_pllscale
   - This is used to perform all the steps required to enable PLL scaling
   - DPM passes the required Core, AHB and IPG frequency.
3. mxc_pm_lowpower
   - Implements all the steps required to put the system under STOP, DOZE, WAIT or DSM mode

28.6 Unit Test

Low-level PM driver is tested on the hardware. Frequency scaling using both PCDR0 values and PLL scaling have been tested. A test module called mxc_pm_test.c was written to test the kernel space API provided by the low-level PM driver. This test module provides IOCTL's, namely,

1. MXCTEST_PM_INTSCALE - This option is used to scale frequency using scaling using PCDR0
2. MXCTEST_PM_PLLSCALE - This option is used to scale frequency using PLL Scaling
3. MXCTEST_PM_LOWPOWER - This option is used to test WAIT, DOZE, STOP or DSM low-power modes. To perform this test, JTAG has to be disconnected from the hardware and the kernel should be executed using the command-line option, "jtag = off", which is the default option.
   - Note that Ethernet interrupts will make LPM testing with NFS root not work. Ethernet, Keypad and External UART interrupts will wake up the core. Wake up from the low-power modes are tested using external UART interrupt and keypad interrupt.
4. MXCTEST_PM_CKOH - This option is used to select different clock outputs like Core, AHB or IP so that their corresponding frequencies can be seen on the oscilloscope. The default clock output is set by Redboot.

The unit test application and the test module are called mxc_pm_test.out and mxc_pm_test.ko, respectively, and can be found in the ROOTFS.

Table 28-3.

File	Location
mxc_pm_test.ko	/unit_tests/modules
mxc_pm_test.out	/unit_tests/mxc_pm_test

insmod mxc_pm_test.ko

./mxc_pm_test.out

The actual frequency change or a voltage change request can only be measured and detected using an oscilloscope.

The State Retention Mode can be tested only using DPM command, "`echo -n mem > /sys/power/state`". Unit-test cannot be used to test this mode.

Chapter 29
Dynamic Voltage Frequency Scaling (DVFS)/Dynamic Process and Temperature Compensation Driver

The Linux Dynamic Voltage Frequency Scaling (DVFS) and the Dynamic Process and Temperature Compensation (DPTC) device driver monitors the current operating point, using four reference circuits that test the IC processing under the current ambient temperature. Dynamic Process Temperature Compensation (DPTC) is a power management technique that reduces power consumption by adjusting supply voltages according to the specific process case, chip fabrication, and ambient temperature.

DVFS (designed as part of the CCM module) allows simple S/W dynamic voltage frequency scaling. Frequency of MCU clock domain and voltage of the chip can be changed on the fly with all modules, including MCU, continue running. Voltage of the chip can be changed by setting of DVS0 and DVS1 pins connected to the MC13783 Power and Audio Management IC (PMIC). Frequency of MCU clock domain can be changed by switching to alternate PLL clock (MCU or SR PLL's) with previous locking to required frequency or just changing post dividers division factors.

The software module is comprised of a Linux driver that allows privileged users to control and monitor the DVFS/DPTC operation. The DVFS/DPTC Linux driver is designed as a character driver with a dynamically selected major number and the minor number 1.

29.1 Hardware Operation

29.1.1 DVFS

The DFVS module is a power management module designed as part of the CCM module (i.MX31 and i.MX31L multimedia applications processors documentation). The purpose of the DFVS module is to detect the appropriate operation frequency for the IC, considering the frequency of idle mode in the ARM core and considering other signals, using weights for such signals set by the user. It generates an ARM interrupt or SDMA event when the frequency must be changed. It records a log buffer for power patterns analysis and can generate an ARM interrupt or SDMA event each predefined time quantum for frequency change according to the log buffer.

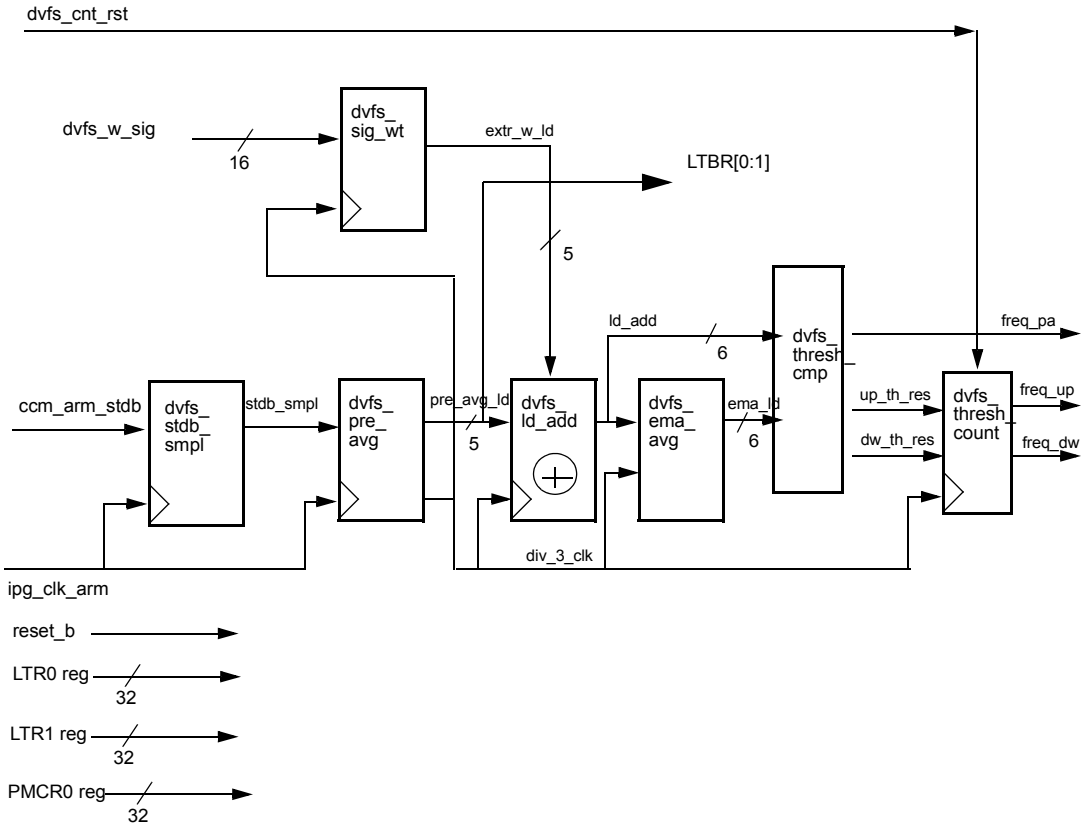

Figure 29-1. DVFS Load Tracking Module Block Diagram

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

29.1.2 Dynamic Process and Temperature Compensation (DPTC)

The Dynamic Process and Temperature Compensation (DPTC) module is a power management module designed as part of the CCM module (see hardware documentation). The purpose of the DPTC module is to detect the minimum operation voltage for the IC, regarding process corner case and temperature for a given frequency. It gets predefined values for process speed performance measurement and generates an interrupt if a supply voltage value update is required.

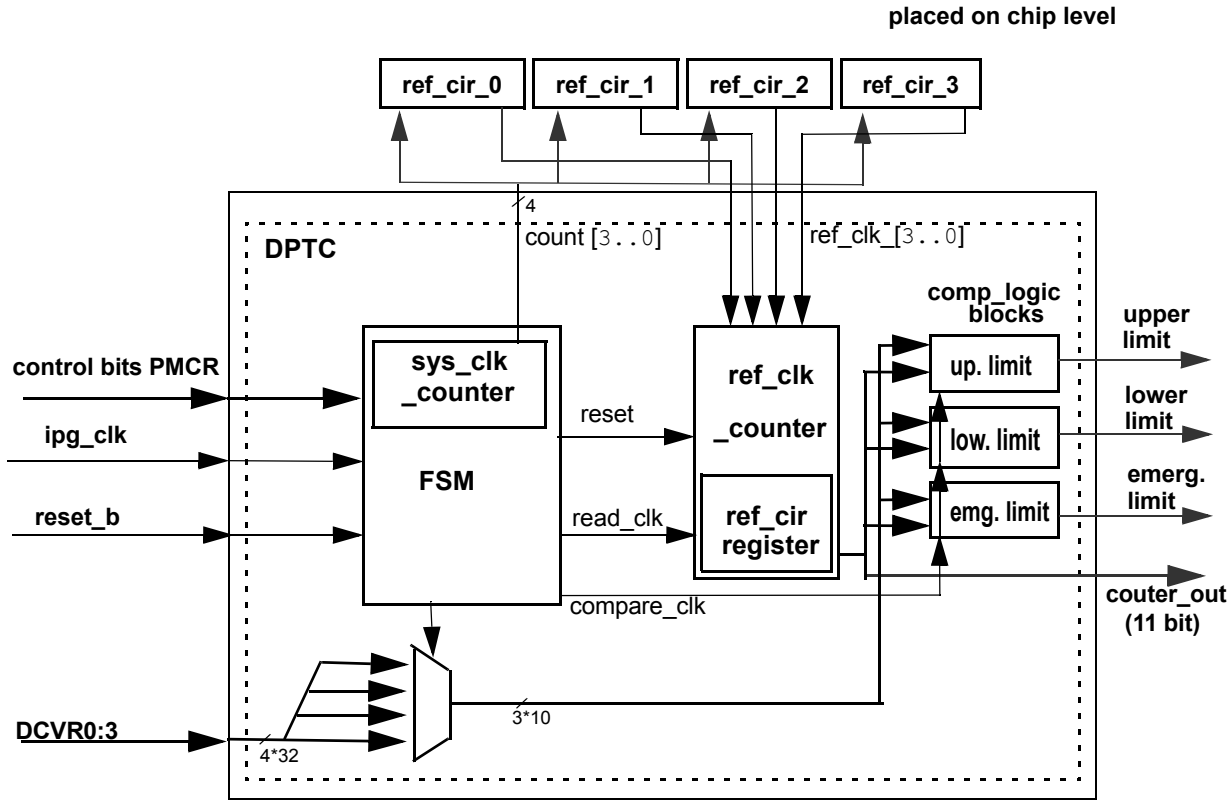

Figure 29-2. DPTC Hardware Module Design

The DPTC module is comprised of 4 reference circuits (ref_cir_0 - ref_cir_3), control module (FSM module), counter and a comparison block.

The FSM module is in charge of operating the DPTC module. On the DPTC module, enabling FSM selects one of the reference circuits (each circuit tests a different process parameter). The selected reference circuit then produces a clock signal (`ref_clk`), which is counted by the `ref_clk_counter`. After the measurement is completed the `ref_clk_counter` value is compared with 3-threshold values upper limit, lower limit and emergency limit. If one of the thresholds is exceeded an interrupt is triggered.

On receiving an interrupt the DPTC driver checks which of the thresholds was exceeded and changes the IC voltage and DPTC thresholds accordingly.

29.1.3 Software Operation

The DVFS/DPTC device driver is designed to monitor and control the DVFS/DPTC hardware module, and perform the transitions between IC working points.

The DVFS/DPTC driver is controlled by a user space daemon that can read/update the voltage/frequency transition table and configure the driver via IOCTL commands.

Although DVFS/DPTC are independent hardware modules, they use the same power states transition table. In order to avoid a race condition, the state transitions are operated by single SDMA script for both DVFS/DPTC.

Therefore, mutual power management driver controls both DVFS/DPTC. Figure 29-3 shows the high level software design for the Linux Dynamic Voltage Frequency Scaling (DVFS) and the Dynamic Process and Temperature Compensation (DPTC) device driver.

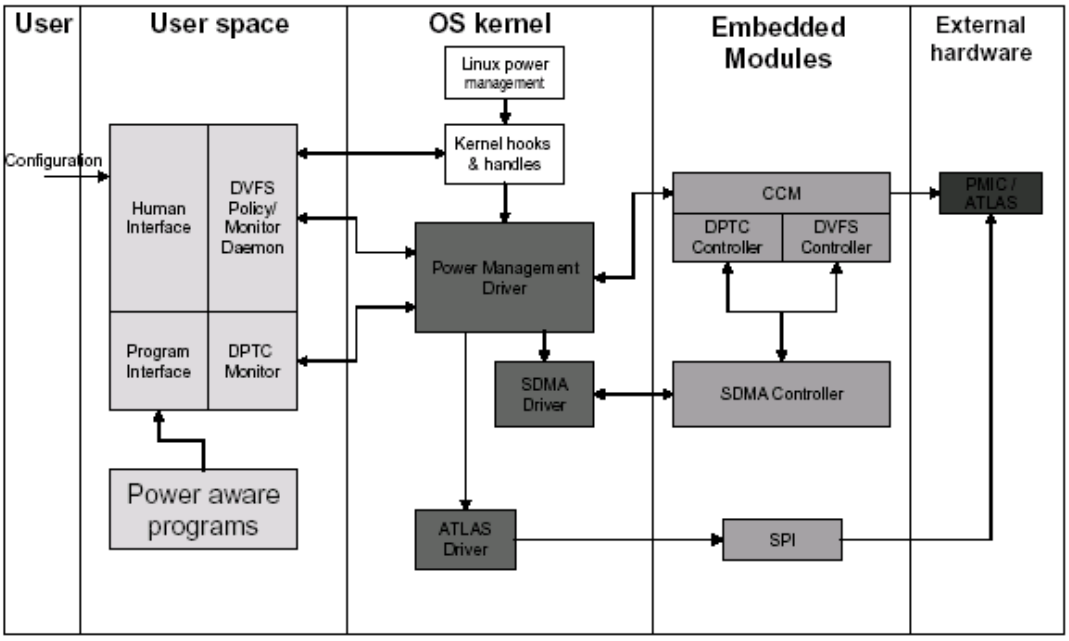

Figure 29-3. DFVS and DPTC Driver Hi-Level Software Design

Figure 29-4 shows the FSM Control loop.

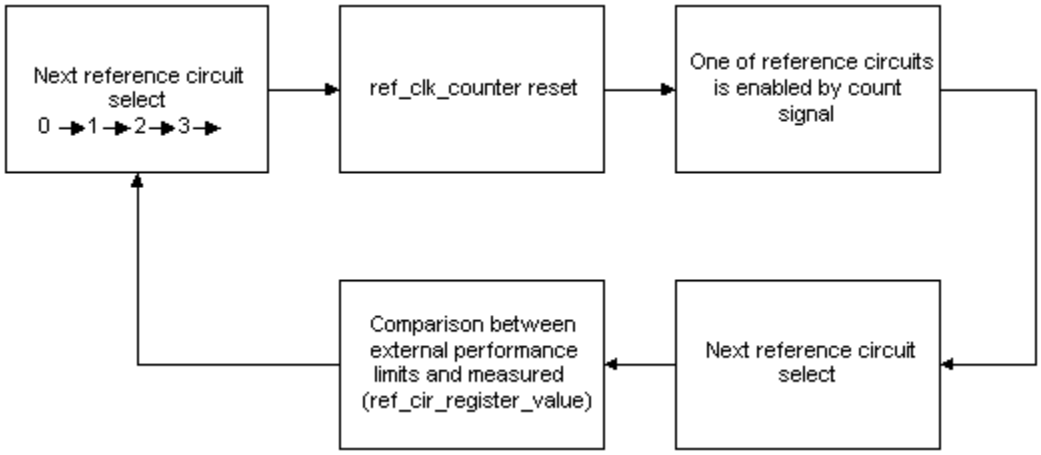

Figure 29-4. FSM Control Loop

29.1.4 DVFS Driver Operation

DVFS hardware can work in three different modes:

- user controlled
- hardware controlled

- hardware supported load analysis.

The driver enables switching between the modes and operates according to the selected mode.

1. User controlled mode
   — DVFS Controller load tracking module is disabled.
   — User sets the current performance level using `ioctl`.
2. Hardware controlled mode

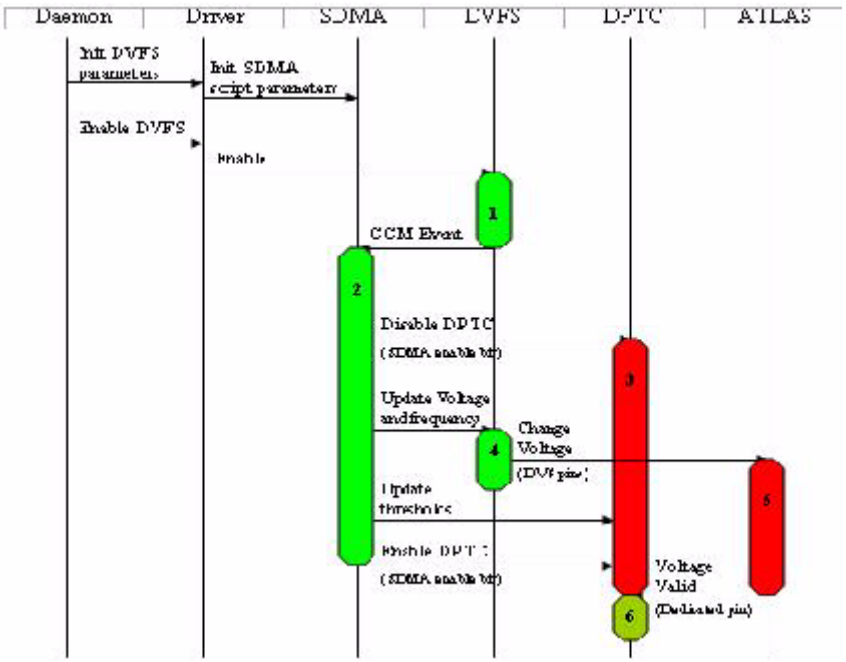

Figure 29-5. DVFS Operation in Hardware Controlled Mode

   — DVFS Operation in Hardware Controlled Mode
   — DVFS controller is enabled and performs load monitoring.
   — SDMA CCM script starts running after the DVFS controller signals a DVFS change.
   — DPTC is disabled.
   — DVFS (CCM) hardware performs frequency and voltage change. The DVFS selects current DVFS voltage by changing the 2 DVS pins connected to the MC13783 Power and Audio Management IC.
   — MC13783 voltages transition to the new values. During the transition the MC13783 signals that the voltages are not valid using a dedicated signal.
   — MC13783 voltages transition to a valid state and DPTC is re-enabled if the `DPTC_cpu_enable` bit is set.
3. Hardware supported load analysis mode
   — not yet defined.

29.1.5 DPTC Driver Operation

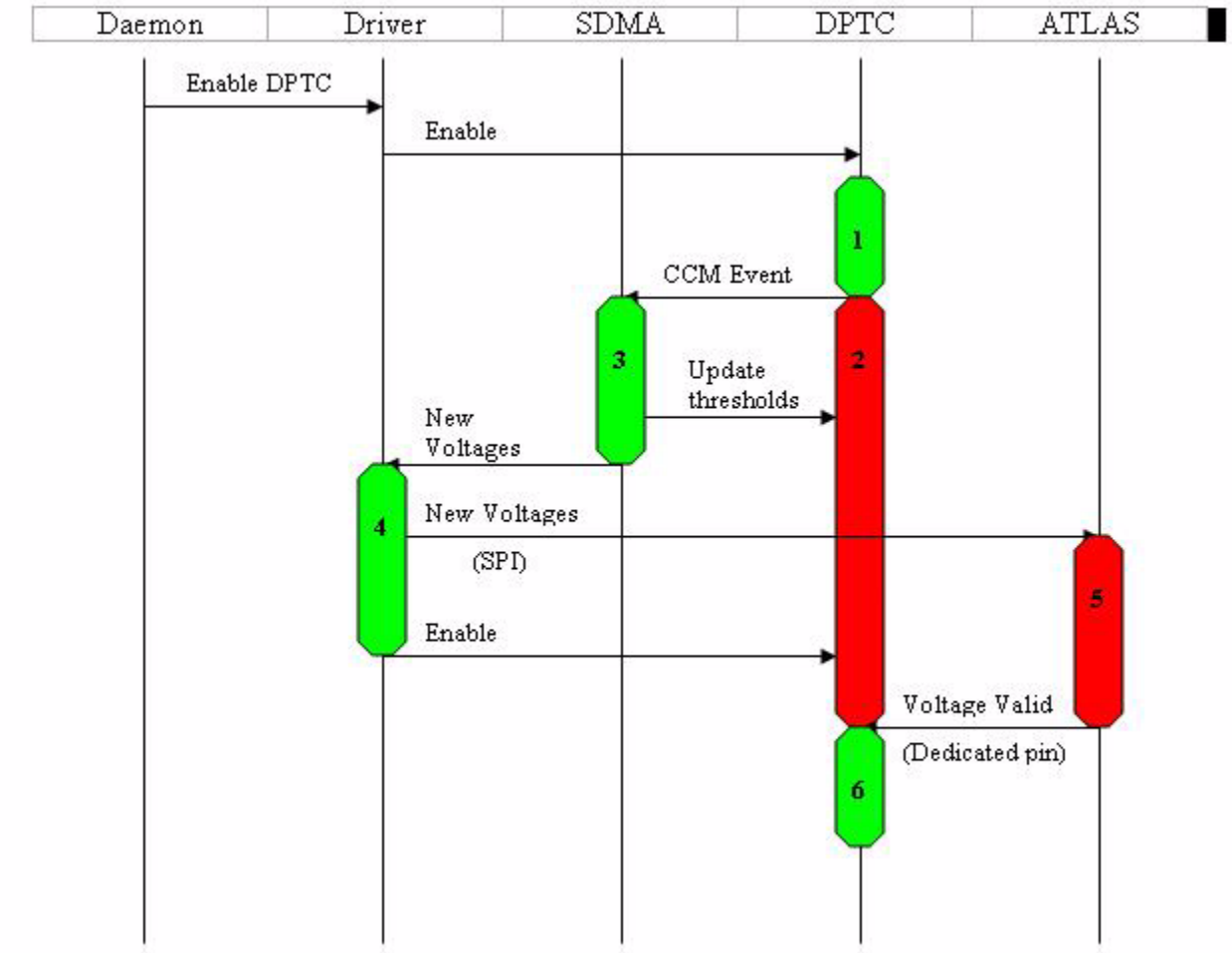

Figure 29-6. DPTC Driver Operation

- DPTC controller is enabled and starts working point calculation.
- DPTC controller signals a working point change and disables itself by clearing the `DPTC_cpu_enable` bit.
- SDMA CCM script starts. The script checks the requested change, calculates a new working point, reads new DPTC thresholds from the DPTC translation table and writes them to the DPTC controller. The script also reads the new voltage values from the table and sends them to the CPU as SDMA data.
- The APM receives new voltage values and sends them to the MC13783 Power and Audio Management IC using the SPI bus via the MC13783 driver. After MC13783 values are updated the driver re-enables the DPTC controller (DPTC remains disabled until MC13783 voltages are valid).
- MC13783 voltages transition to the new values. During the transition the MC13783 signals that the voltages are not valid using a dedicated signal.
- MC13783 voltages transition to a valid state and DPTC is re-enabled.

29.1.6 Simultaneous DVFS/DPTC Driver Operation

DPTC can work in each one of described DVFS operating modes simultaneously. Figure 29-7 describes the driver operation of DPTC with DVFS hardware controlled mode.

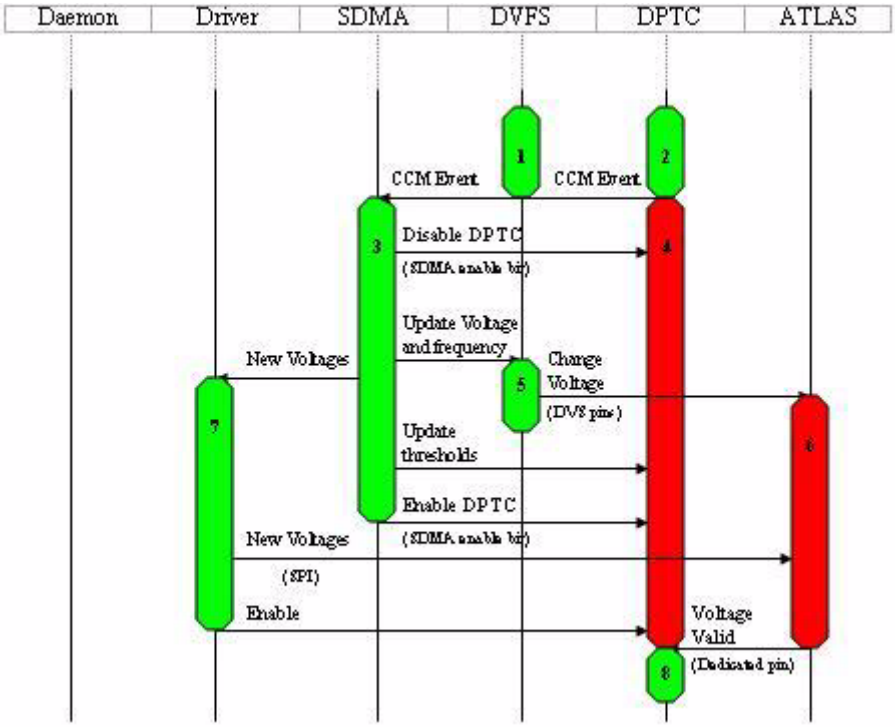

Figure 29-7. Simultaneous DVFS in HW Controlled Mode with DPTC

- DVFS controller is enabled and performs load monitoring.
- DPTC controller is enabled and performs working point calculation.
- SDMA CCM script starts running after the DVFS/DPTC controllers signaled a CCM event.
- DPTC is disabled.
- DVFS (CCM) hardware performs frequency and voltage change. The DVFS selects current DVFS voltage by changing the 2 DVS pins connected to the MC13783 Power and Audio Management IC.
- MC13783 voltages transition to the new values. During the transition the MC13783 signals that the voltages are not valid using a dedicated signal.
- The APM receives new voltage values and sends them to the MC13783 using the SPI bus via the MC13783 driver. After MC13783 values are updated the driver re-enables the DPTC controller (DPTC remains disabled until MC13783 voltages are valid).
- MC13783 voltages transition to a valid state and DPTC is re-enabled.

29.1.7 DVFS/DPTC Driver Interaction with MC13783

The DVFS/DPTC driver uses the MC13783 Power and Audio Management IC IC to change the voltage of the chip.

On DPTC working point change request, the driver sets values for four different voltages on the MC13783 Power and Audio Management IC—SW1A SW setting, SW1A DVF setting, SW1B DVS setting and SW1B STANDBY setting. The change is done using MC13783 Power and Audio Management IC Power API functions, through SPI. After the voltage change, the DPTC is enabled when it gets the power-ready interrupt from the MC13783. This signal comes from the PWRRDY pin of the MC13783. It is connected to the GPIO1_5 pin of the i.MX31 and i.MX31L multimedia applications processors.

On DVFS frequency change request, the driver selects one of the four voltages according to the new frequency. The change is done by writing to the DVSUP[0-1] bits of the CCM PMCR0 register. These bits are connected to the DVFS0 and DVFS1 output pins of the i.MX31 and i.MX31L multimedia applications processors. These pins are connected to the DVSSW1A and DVSSW1B pins of the MC13783.

29.1.8 SDMA Operation

SDMA handles the CCM DMA event. This event occurs in any one of the following conditions:

1. DPTC voltage change request
2. DVFS frequency change request
3. DVFS log buffer full (TBD)
4. All conditions listed above

On each event, SDMA checks the DPTC state and the DVFS state.

The DVFS/DPTC driver installs a script according to the current DVFS operating mode. In manual mode DVFS is disabled. No script is running on SDMA.

On DPTC voltage change request, SDMA sets new DPTC thresholds according to the new working point and current frequency, and sends an interrupt to the ARM core. The ARM core will update the voltages using the MC13783 Power and Audio Management IC driver. SDMA should handle DPTC requests, although it doesn’t update the voltage, because only SDMA knows the current state of DVFS.

On DVFS frequency change request, SDMA updates the frequency, sets new DPTC thresholds according to the new working point and current frequency. SDMA selects one of the four voltages by writing to the DVSUP bits.

On DVFS log buffer full event - TBD.

The SDMA script is divided into two sections: DPTC and DVFS. Figure 29-8 describes DVFS section.

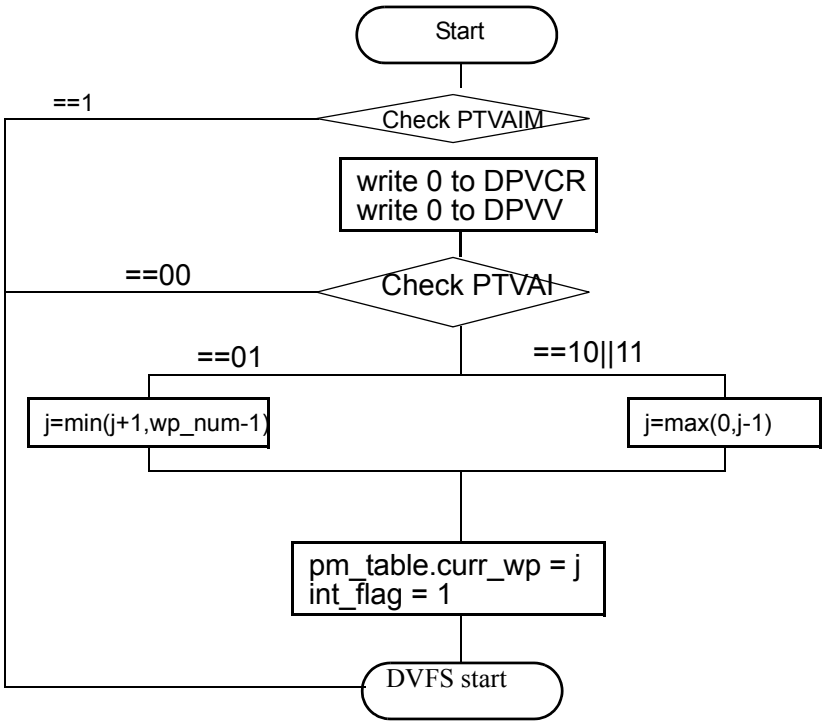

Figure 29-8. SDMA Script (DVFS Hardware Controlled Mode)

29.2 Requirements

The Dynamic Voltage Frequency Scaling (DVFS) and the Dynamic Process and Temperature Compensation (DPTC) module implement the following requirements.

- [R-DVFS-DPTC-1] The DVFS/DPTC driver will allow a privileged user to control DVFS/DPTC operation
  - — Update power states transition table.
  - — Read currently loaded transition table.
- [R-DVFS-DPTC-2] The DVFS/DPTC driver will set up the SDMA script according to the operating mode
- [R-DVFS-1] The DVFS driver will allow a privileged user to control DVFS operation
  - — Switch between 3 DVFS operating modes
- [R-DVFS-2] The DVFS driver will allow setting power state in user controlled mode
- [R-DVFS-3] The DVFS driver will allow the following operations in hardware controlled mode:
  - — Update general purpose register value (for DPM)
  - — Read statistics file
- [R-DVFS-4] The DVFS driver will allow the following operations in hardware supported load analysis mode:
  - — Read log buffer

  - Write new pattern
  - Read statistics file
- [R-DVFS-5] On DVFS interrupt from SDMA in hardware supported load analysis mode, the driver will read a a log buffer and/or set a new pattern
- [R-DPTC-1] The DPTC driver will allow a privileged user to control DPTC operation:
  - `Enable/Disable module.`
- [R-DPTC-2] On DPTC interrupt from SDMA the driver will update the current IC voltage according to the DPTC controller measurements.
- The sample module conforms to the WMSG Linux coding standards.

29.3 Source Code Structure

Table 29-1 lists the source files contained in the devices directory `linux/devices.`

Table 29-1. Source Code Files

File	Description
pm_api.h	Header file containing structure definitions and defines used in the DVFS/DPTC driver API.
dptc_dvfs_struct.h	Header file containing structure definitions for DVFS/DPTC driver.
dptc.h	Header file containing DPTC structure definitions and constants.
dptc.c	Linux DPTC driver module.
dvfs.h	Header file containing DVFS structure definitions and constants.
dvfs.c	Linux DVFS functions.
dvfs_dptc.h	Header file Linux DVFS/DPTC driver module
dvfs_dptc.c	Linux DVFS/DPTC driver module.

29.4 Linux Menu Configuration Options

The following Linux kernel configurations are provided for this module:

- MXC_DVFS—This is the configuration option for the MXC DVFS driver. In the `menuconfig` this option is found under Advanced Power Management. By default, this option is Y for i.MX31 and i.MX31L multimedia applications processors architecture.
- MXC_DVFS_SDMA—This is the configuration option for the DVFS SDMA support. It is dependent on MXC_DVFS. In the `menuconfig` this option is found under Advanced Power Management. By default, this option is N for i.MX31 and i.MX31L multimedia applications processors architecture.
- MXC_DPTC—This is the configuration option for the MXC DPTC driver. In the `menuconfig` this option is found under Advanced Power Management. By default, this option is Y for the ARM core and i.MX31 and i.MX31L multimedia applications processors architectures.

29.4.1 Source Code Configuration Options

- DPTC_REF_CIRCUITS_STATUS—defines default status of four reference circuits. Each reference circuit is represented by 1 bit. The 1st LSB is for reference circuit 0, the 2nd for reference circuit 1, the 3rd for reference circuit 2, the 4th for reference circuit 3. If the bit is set, the reference circuit is enabled.

29.4.1.1 Board Configuration Options

There are no board configuration options for the Linux Dynamic Voltage Frequency Scaling (DVFS) and the Dynamic Process and Temperature Compensation (DPTC) device driver.

29.5 Programming Interface

This driver creates `/dev/dvfs_dptc*` and `/proc/dptc` device files and allows standard Linux driver API functions `(open, close, ioctl)` together with a standard proc file system read entry to allow reading the DPTC log buffer.

NOTE

If only MXC_DPTC is enabled during compilation, the driver creates the `/dev/dptc` device file. If only MXC_DVFS is enabled during compilation, the driver creates the `/dev/dvfs` device file.

- `open`–Allows programs to access to the driver. Access is permitted only to programs that have root privileges.
- `close`–Program that opened the driver relinquishes its driver access.
- `ioctl`–Allows programs to send `ioctl` commands to the DPTC driver. `Ioctl` commands:
- `PM_IOCSTABLE`–Changes the current DPTC driver lookup table. The lookup table format is described below.
- `PM_IOCGTABLE`–Returns the current DPTC driver lookup table.
- `PM_IOCGFREQ`–Returns current ARM frequency in Hz.
- `DPTC_IOCTENABLE`–Enables the DPTC module.
- `DPTC_IOCTDISABLE`–Disables the DPTC module.
- `DPTC_IOCSENABLERC`–Enables reference circuits. The argument passed to this `ioctl` function is an integer. 4 LSBs define which reference circuits will be enabled.
- `DPTC_IOCSDISABLERC`–Disables reference circuits. The argument passed to this `ioctl` function is an integer. 4 LSBs define which reference circuits will be disabled.
- `DPTC_IOCGSTATE`–Returns the current DPTC module state (Enabled/Disabled).
- `DPTC_IOCSWP`–Sets new working point (for debugging only). The argument passed to `ioctl` is an integer - number of working point. The `ioctl` will return an error if DPTC HW is enabled.
- `DVFS_IOCTENABLE`–Enables the DVFS module.
- `DVFS_IOCTDISABLE`–Disables the DVFS module.
- `DVFS_IOCGSTATE`–Returns the current DVFS module state (Enabled/Disabled)

- DVFS_IOCSSWGP—Sets SW general purpose bits value. The argument passed to the ioctl is an integer. 4 LSB bits specify the status of 4 SW general purpose bits.
- DVFS_IOCSWFI—Enables/Disables wait-for-interrupt monitoring feature. If the argument equals 1 - the WFI monitoring is enabled, 0 - disabled.
- DVFS_IOCSFREQ—Sets new frequency. The argument passed to ioctl is an integer - number of DVFS state corresponding to required frequency. The ioctl will return an error if DVFS HW is enabled.

The DPTC driver creates a proc file system entry that allows programs to read the DPTC driver log buffer. The log buffer is read using a regular file read command.

29.6 Lookup Table Format

The argument passed to PM_IOCSTABLE and returned by PM_IOCGTABLE ioctls functions is a string of several lines concatenated with a '\n' (new line) character. Blank lines are allowed. '#' character at the start of line is a start of comments symbol. An example of the table follows.

```
########################## START of EXAMPLE table ##########################
WORKING POINT 17

# MC13783 switcher SW values for each working point.
# The first line is for WP of highest voltage.
# The first column is for highest frequency.
#      SW1A        SW1A DVS     SW1B DVS    SW1B STANDBY
WP   0x1c            0xd                      0xc                        0xc
WP   0x1b            0xd                      0xc                       0xc
WP   0x1a            0xd                      0xc                       0xc
WP   0x19            0xd                      0xc                       0xc
WP   0x18            0xd                      0xc                       0xc
WP   0x17            0xd                      0xc                       0xc
WP   0x16            0xd                      0xc                       0xc
WP   0x15            0xd                      0xc                       0xc
WP   0x14            0xc                      0xc                       0xc
WP   0x13            0xc                      0xc                       0xc
WP   0x12            0xc                      0xc                       0xc
WP   0x11            0xc                      0xc                       0xc
WP   0x10            0xc                      0xc                       0xc
WP   0xf               0xc                      0xc                       0xc
WP   0xe             0xc                      0xc                       0xc
WP   0xd             0xc                      0xc                       0xc
WP   0xc             0xc                      0xc                       0xc

#   pll_sw_up      pll_sw_down      pdr_up         pdr_down        pll_up
pll_down        vscnt
# 532MHz
FREQ 0                   1                      0xff871e58       0xff871e50
0x0033280c        0x00331c23        7
# 399MHz
FREQ 1                   1                      0xff871e58       0xff871e59
0x0033280c        0x0033280c        7
# 266MHz
FREQ 1                   1                      0xff871e58       0xff871e5b
0x0033280c        0x0033280c        7
```

```
# 133MHz
FREQ 1                    0                       0xff871e58       0xff871e5b
0x0033280c        0x0033280c        7

# 532MHz
DCVR 0xffc00000 0x801e9794 0xffc00000 0xffc00000
DCVR 0xffc00000 0x805e9794 0xffc00000 0xffc00000
DCVR 0xffc00000 0x805e9794 0xffc00000 0xffc00000
DCVR 0xffc00000 0x80dea794 0xffc00000 0xffc00000
DCVR 0xffc00000 0x811ea794 0xffc00000 0xffc00000
DCVR 0xffc00000 0x811eb794 0xffc00000 0xffc00000
DCVR 0xffc00000 0x815eb798 0xffc00000 0xffc00000
DCVR 0xffc00000 0x81deb798 0xffc00000 0xffc00000
DCVR 0xffc00000 0x821ec798 0xffc00000 0xffc00000
DCVR 0xffc00000 0x825ec798 0xffc00000 0xffc00000
DCVR 0xffc00000 0x829ed79c 0xffc00000 0xffc00000
DCVR 0xffc00000 0x831ee79c 0xffc00000 0xffc00000
DCVR 0xffc00000 0x83dee79c 0xffc00000 0xffc00000
DCVR 0xffc00000 0x845ef7a0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x84df07a0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x845f17a4 0xffc00000 0xffc00000
DCVR 0xffc00000 0x83df27a4 0xffc00000 0xffc00000

# 399MHz
DCVR 0xffc00000 0x8016f5b0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8056f5b0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8056f5b0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x80d6f5b0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x811705b0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x811705b0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x815705b0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x81d715b0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x821715b4 0xffc00000 0xffc00000
DCVR 0xffc00000 0x825715b4 0xffc00000 0xffc00000
DCVR 0xffc00000 0x829725b4 0xffc00000 0xffc00000
DCVR 0xffc00000 0x831725b4 0xffc00000 0xffc00000
DCVR 0xffc00000 0x83d735b8 0xffc00000 0xffc00000
DCVR 0xffc00000 0x845735b8 0xffc00000 0xffc00000
DCVR 0xffc00000 0x84d745b8 0xffc00000 0xffc00000
DCVR 0xffc00000 0x845755bc 0xffc00000 0xffc00000
DCVR 0xffc00000 0x83d765bc 0xffc00000 0xffc00000

# 266MHz
DCVR 0xffc00000 0x800f53c8 0xffc00000 0xffc00000
DCVR 0xffc00000 0x804f53c8 0xffc00000 0xffc00000
DCVR 0xffc00000 0x804f53c8 0xffc00000 0xffc00000
DCVR 0xffc00000 0x80cf53cc 0xffc00000 0xffc00000
DCVR 0xffc00000 0x810f53cc 0xffc00000 0xffc00000
DCVR 0xffc00000 0x810f53cc 0xffc00000 0xffc00000
DCVR 0xffc00000 0x814f53cc 0xffc00000 0xffc00000
DCVR 0xffc00000 0x81cf63cc 0xffc00000 0xffc00000
DCVR 0xffc00000 0x820f63cc 0xffc00000 0xffc00000
DCVR 0xffc00000 0x824f63cc 0xffc00000 0xffc00000
DCVR 0xffc00000 0x828f63cc 0xffc00000 0xffc00000
DCVR 0xffc00000 0x830f73cc 0xffc00000 0xffc00000
DCVR 0xffc00000 0x83cf73d0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x844f83d0 0xffc00000 0xffc00000
```

```
DCVR 0xffc00000 0x84cf83d0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x844f93d0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x83cf93d4 0xffc00000 0xffc00000

# 133MHz
DCVR 0xffc00000 0x8007a1e8 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8047a1e8 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8047a1e8 0xffc00000 0xffc00000
DCVR 0xffc00000 0x80c7a1e8 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8107b1ec 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8107b1ec 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8147b1ec 0xffc00000 0xffc00000
DCVR 0xffc00000 0x81c7b1ec 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8207b1ec 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8247b1ec 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8287b1ec 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8307b1ec 0xffc00000 0xffc00000
DCVR 0xffc00000 0x83c7c1f0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8447c1f0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x84c7c1f0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x8447c1f0 0xffc00000 0xffc00000
DCVR 0xffc00000 0x83c7d1f4 0xffc00000 0xffc00000

#############     END of EXAMPLE table ##########################
```

29.7 Usage Example

TBD

29.8 Unit Test

The driver can be tested using `dptc_hi` demo located at `LINUX2.6/misc/source/demo/dvfs_dptc_hi_demo/dvfs_dptc_hi.c`. (Select Demo Programs -> Available Demo Programs -> DVFS & DPTC Human Interface Demo option in `menuconfig`).

The demo provides several options:

DPTC module status:

Enabled - 0

DVFS module status:

Enabled - 0

DPTC driver commands

1. Enable DPTC—Enables DPTC. When DPTC is enabled, “DPTC module status: Enabled - 1” appears above the menu.
2. Disable DPTC—Disables DPTC. When DPTC is disabled, “DPTC module status: Enabled - 0” appears above the menu.
3. Update DPTC driver translation table—Updates the driver translation table. When it is chosen, provide a file name for the new table.

4. Read DPTC driver translation table—Dumps the table into a file. When it is chosen, provide an output file name.
5. Set DPTC reference circuits—Sets the reference circuit status. When it is chosen a number in hexadecimal format should be provided. Only 4 LSBs are used, each bit corresponds to 1 of the 4 reference circuits. The number says which reference circuits should be enabled and which reference circuits should be disabled.
6. Show DPTC log buffer—Shows the log buffer. The log buffer is a table with 2 columns: time point and DPTC working point. It can be used to verify that DPTC changes the voltages when it is enabled. The table updates automatically, until the key is pressed to exit from the menu.
7. Set DPTC working point—Sets a DPTC working point. It can be used for debugging only, and is activated only when DPTC is disabled. It updates the voltage according to new working point number.

DVFS driver commands

8. Enable DVFS—Enables DVFS. When DVFS is enabled, “DVFS module status: Enabled - 1” should appear above the menu.
9. Disable DVFS—Disables DVFS. When DVFS is disabled, “DVFS module status: Enabled - 0” should appear above the menu.

a) Set SW general purpose bits—Writes to SW general purpose bits. A number in hexadecimal format should be provided. Only 4 LSBs will be used. These 4 bits will be written to register

b)Set WFI monitor—Enables/disables WFI monitoring. If WFI monitoring is disabled, then DVFS will take into account only general purpose bits.

c)Show frequency—Displays current frequency.

---

q) Quit—Quits the program.

For testing DPTC, use options 1 and 2 to enable/disable DPTC. Use option 6 to see that the voltage is changing. Use option 5 to select reference circuits. Use option 3 and 4 to update and dump the driver translation table.

For testing DVFS, use options 8 and 9 to enable/disable DVFS. Use option c to see that the frequency is changing.

Chapter 30
Dynamic Process and Temperature Compensation (DPTC) Driver

The Dynamic Process Temperature Compensation (DPTC) Driver manages the DPTC power management technique. This technique reduces power consumption by adjusting the supply voltages according to the specific process case, chip fabrication and ambient temperature.

The DPTC hardware module (designed as part of the CCM module) monitors the current operating point using four reference circuits that test the chip process under the current ambient temperature.

The software module is a Linux driver that allows privileged users to control and monitor the DPTC operation. The DPTC Linux driver is designed as a character driver with a dynamically selected major number and the minor number 1.

30.1 Hardware Operation

The DPTC module is a power management module designed as part of the CCM module. The purpose of the DPTC module is to detect the minimum operation voltage for the IC, regarding process corner case and temperature for a given frequency. It gets predefined values for process speed performance measurement and generates an interrupt if a supply voltage value update is required.

Figure 30-1 shows a block diagram of the DPTC hardware operation.

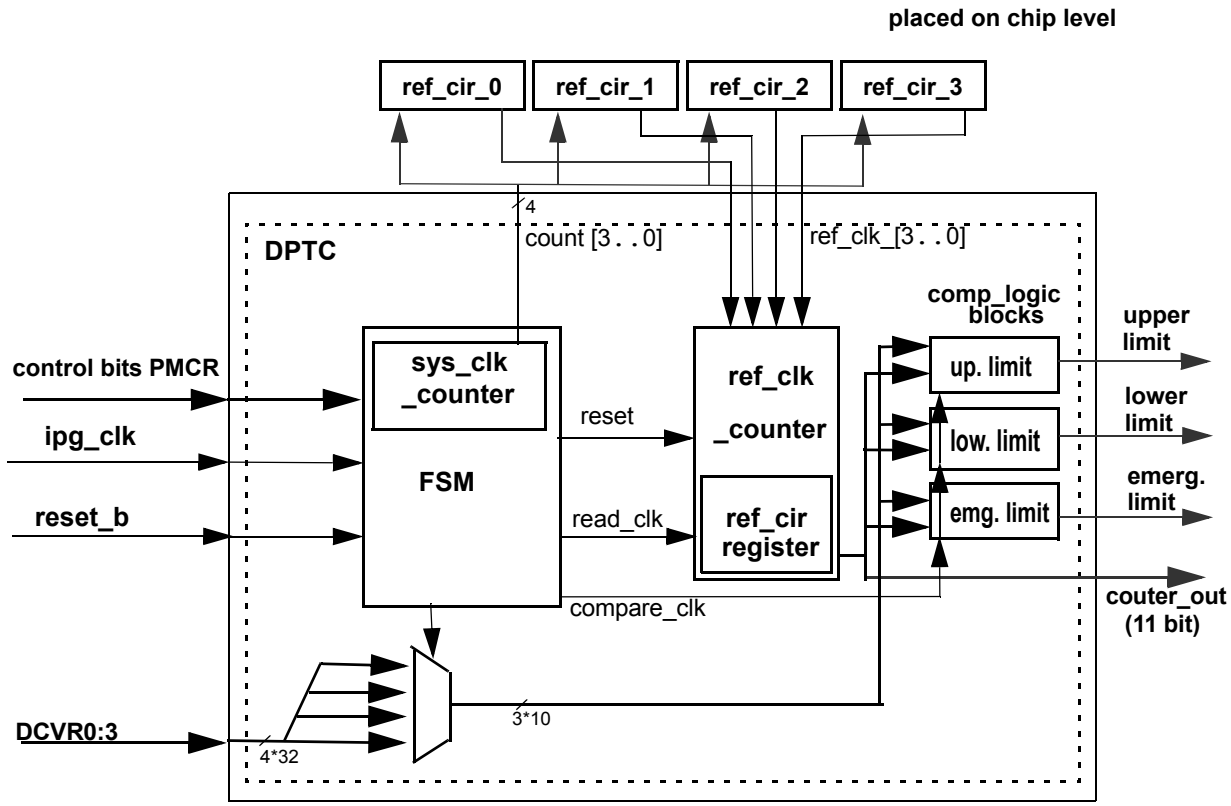

Figure 30-1. DPTC Hardware Module Design

The DPTC module contains four reference circuits (ref_cir_0 - ref_cir_3), control module (FSM module), counter and a comparison block.

The FSM module manages the operation of the DPTC module. On DPTC module enable, FSM selects one reference circuit (each circuit tests a different process parameter), the selected reference circuit then produces a clock signal (`ref_clk`), which is counted by the `ref_clk_counter`. After the measurement is completed, the `ref_clk_counter` value is compared with three threshold values: upper limit, lower limit and emergency limit. If one of the thresholds is exceeded, an interrupt is triggered.

On receiving an interrupt, the DPTC driver checks which of the thresholds was exceeded and changes the IC voltage and DPTC thresholds accordingly.

Figure 30-2 shows the FSM control loop.

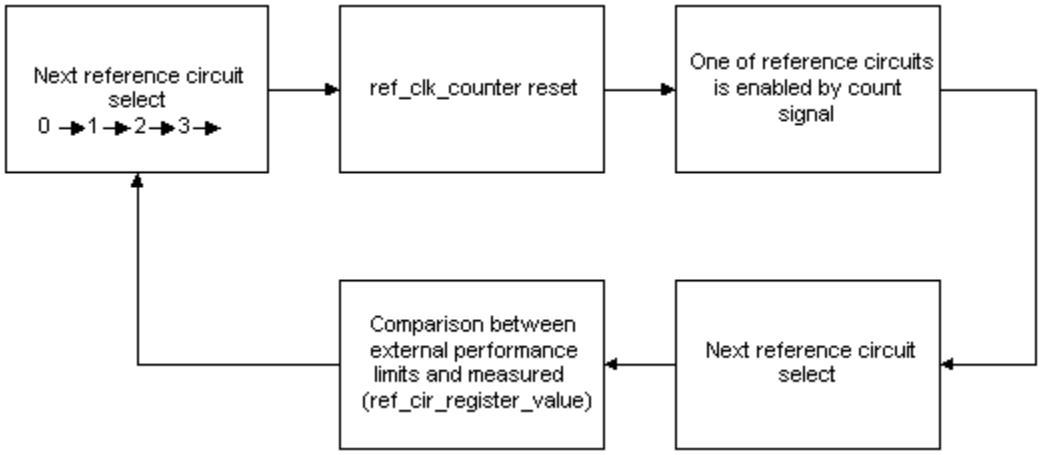

Figure 30-2. FSM Control Loop

30.2 Software Operation

The DPTC device driver is designed to monitor and control the DPTC hardware module, and it performs the transitions between IC working points.

The DPTC driver is controlled by a user space daemon that can read/update the DPTC table and configure the driver via IOCTL commands. Figure 30-3 shows the DPTC driver high level software design.

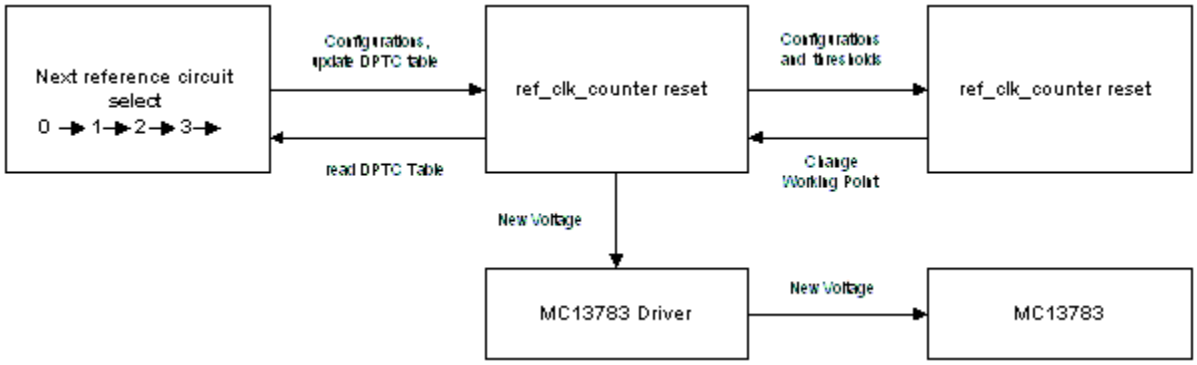

Figure 30-3. DPTC Driver Software Design

Driver operation:

1. DPTC user space software enables the DPTC driver using an IOCTL command.
2. The driver configures and enables the DPTC controller.
3. The DPTC controller measures the current IC working point and signals an interrupt if there is a need to move to another working point.
4. On receiving an interrupt, the DPTC driver calculates the new working point.
5. Using the DPTC lookup table, the driver calculates the new IC voltage and updates the current IC voltage via the MC13783 driver.
6. The driver writes new thresholds to the DPTC controller.
7. The DPTC controller starts a new measurement.

30.2.1 DVFS&DPTC - MC13783 interaction

DVFS&DPTC driver uses MC13783 to change the voltage of the chip.

On DPTC working point change request, the driver sets values for four different voltages on the MC13783 - SW1A SW setting, SW1A DVF setting, SW1B DVS setting and SW1B STANDBY setting. The change is done using MC13783 Power API functions, through SPI. After the voltage change, the DPTC is enabled when it gets a power-ready interrupt from MC13783. This signal comes from the PWRRDY pin of the MC13783, and it is connected to GPIO1_5 pin of the multimedia application processor.

On DVFS frequency change request, the driver selects one of the four voltages according to the new frequency. The change is done by writing to the DVSUP[0-1] bits of the CCM PMCR0 register. These bits are connected to the DVFS0 and DVFS1 output pins of the multimedia application processor, and these pins are connected to the DVSSW1A and DVSSW1B pins of the MC13783.

30.2.2 DVFS&DPTC - MC13783 interaction

The DVFS&DPTC driver uses the MC13783 to change the voltage of the chip.

On DPTC working point change request, the driver sets values for four different voltages on the MC13783 - SW1A SW setting, SW1A DVF setting, SW1B DVS setting and SW1B STANDBY setting. The change is done using MC13783 Power API functions, through SPI. After the voltage change, the DPTC is enabled when it gets power-ready interrupt from the MC13783. This signal comes from the PWRRDY pin of the MC13783, and it is connected to the GPIO1_5 pin of the multimedia application processor.

On DVFS frequency change request, the driver selects one of the four voltages according to the new frequency. The change is done by writing to the DVSUP[0-1] bits of the CCM PMCR0 register. These bits are connected to the DVFS0 and DVFS1 output pins of the multimedia application processor, and these pins are connected to the DVSSW1A and DVSSW1B pins of the MC13783.

30.3 Requirements

The DPTC driver implements the following requirements:

- [R-DPTC-1] The DPTC driver allows a privileged user to control the DPTC operation and contains the following features:
  - Enable/Disable module.
  - Update the DPTC table.
  - Read currently loaded table.
  - Enable/Disable reference circuits
- [R-DPTC-2] On DPTC interrupt, the driver updates the current IC voltage according to the DPTC controller measurements.
- The sample module conforms to the WMSG Linux coding standards.

30.4 Source Code Structure

Table 30-1 lists the source files contained in the devices directory:

```
linux/devices
```

Table 30-1. Source Code Files

File	Description
pm_api.h	header file containing API structures, functions and definitions
dptc.h	header file containing structure definitions and constants.
dptc.c	Linux DPTC driver functions.
dvfs_dptc.h	Header file Linux DPTC driver module.
dvfs_dptc.c	Linux DPTC driver module.

30.5 Configuration

The following Linux kernel configurations are provided for this module:

- MXC_DPTC—Configuration option for the MXC DPTC driver. In `menuconfig`, this option is found under MXC Support Drivers/Advanced Power Management menu. By default, this option is Y for the multimedia application processor architectures. It is dependent on MC13783_POWER configuration option.

30.5.1 Source Code Configuration Options

- DPTC_REF_CIRCUITS_STATUS—Defines default status of four reference circuits. Each reference circuit is represented by 1 bit. The 1st LSB is for reference circuit 0, the 2nd for reference circuit 1, the 3rd for reference circuit 2, the 4th for reference circuit 3. If the bit is set, the reference circuit is enabled.

30.5.1.1 Board Configuration Options

None.

30.6 API Functions

This driver creates /dev/dptc and /proc/dptc device files and allows standard Linux driver API functions (open, close, ioctl) together with a standard proc file system read entry to allow reading the DPTC log buffer.

- open—Allows programs to access to the driver. Access is permitted only to programs that have root privileges.
- close—Program that opened the driver relinquishes its driver access.
- ioctl—Allows programs to send ioctl commands to the DPTC driver. Ioctl commands:
- PM_IOCSTABLE—changes the current DPTC driver lookup table. The lookup table format is described below.
- PM_IOCGTABLE—returns the current DPTC driver lookup table.
- PM_IOCGFREQ —Returns current ARM frequency in Hz
- DPTC_IOCTENABLE—Enables the DPTC module.
- DPTC_IOCTDISABLE—Disables the DPTC module.
- DPTC_IOCSENABLERC—Enables reference circuits. The argument passed to this ioctl function is an integer. 4 LSBs define which reference circuits will be enabled.
- DPTC_IOCSDISABLERC—Disables reference circuits. The argument passed to this ioctl function is an integer. 4 LSBs define which reference circuits will be disabled.
- DPTC_IOCGSTATE—Returns the current DPTC module state (Enabled/Disabled).
- DPTC_IOCSWP—Sets new working point (for debugging only). The argument passed to ioctl is an integer - the number of the working point.

The DPTC driver creates a proc file system entry that allows programs to read the DPTC driver log buffer. The log buffer is read using a regular file read command.

30.6.1 Lookup Table Format

The argument passed to PM_IOCSTABLE and returned by PM_IOCGTABLE ioctls function is a string of several lines concatenated with '\n' (new line) character. Blank lines are allowed. A '#' character at the start of a line is a start of comments symbol. Below is the example of the table.

```
########################## START of EXAMPLE table ##########################
# Number of working points

WORKING POINT 17

# MC13783 switcher (SW1A) SW values for each working point.
# The first line is for WP of highest voltage.

WP 0x1f
WP 0x1e
WP 0x1d
WP 0x1c
WP 0x1b
WP 0x1a
```

```
WP 0x19
WP 0x18
WP 0x17
WP 0x16
WP 0x15
WP 0x14
WP 0x13
WP 0x12
WP 0x11
WP 0x10
WP 0xf

# 4 DCVR values (thresholds) for each reference circuit.
# The first line is for WP of highest voltage.

DCVR 0xffc00000 0x9c23f8d4 0xffc00000 0xebf62d44
DCVR 0xffc00000 0x9c6418d8 0xffc00000 0xed367d50
DCVR 0xffc00000 0x9ce428dc 0xffc00000 0xee76bd58
DCVR 0xffc00000 0x9d6438dc 0xffc00000 0xf036cd58
DCVR 0xffc00000 0x9e2438dc 0xffc00000 0xf236cd58
DCVR 0xffc00000 0x9ee458e0 0xffc00000 0xf336fd60
DCVR 0xffc00000 0x9f6478e4 0xffc00000 0xf3f72d64
DCVR 0xffc00000 0xa06488e4 0xffc00000 0xf4f71d64
DCVR 0xffc00000 0xa16488e8 0xffc00000 0xf5f70d64
DCVR 0xffc00000 0xa1e4a8ec 0xffc00000 0xf5f72d64
DCVR 0xffc00000 0xa2a4b8ec 0xffc00000 0xf6374d68
DCVR 0xffc00000 0xa364c8f0 0xffc00000 0xf7376d6c
DCVR 0xffc00000 0xa464d8f0 0xffc00000 0xf8777d70
DCVR 0xffc00000 0xa56508f8 0xffc00000 0xfa37cd78
DCVR 0xffc00000 0xa66528fc 0xffc00000 0xfbf80d80
DCVR 0xffc00000 0xa8255900 0xffc00000 0xfeb84d8c
DCVR 0xffc00000 0xa9e58908 0xffc00000 0xfff89d94

#############     END of EXAMPLE table ##########################
Example Usage

#include <asm/arch/pm_api.h>

#include <fcntl.h>

extern char *lookup_table;

int main(){
        int dptc_fh;
        char table[4096];

        // Open device file
        dptc_fh = open("/dev/pm",O_RDWR);

        // Set lookup table
        ioctl(dptc_fh,PM_IOCSTABLE,lookup_table);

        // Disable all reference circuits
        ioctl(dptc_fh,DPTC_IOCSDISABLERC,0xf);
        // Enable reference circuits 0 and 3
        ioctl(dptc_fh,DPTC_IOCSENABLERC,0x9);
```

```
            // Enable DPTC
            ioctl(dptc_fh,DPTC_IOCTENABLE);

            printf("Current DPTC status is: %d\n",
                   ioctl(dptc_fh,DPTC_IOCGSTATE));

            // Disable DPTC
            ioctl(dptc_fh,DPTC_IOCTDISABLE);

            // Read current lookup table
            ioctl(dptc_fh,PM_IOCGTABLE,table);
            printf("Current table :\n",
                   table);
}
```

30.7 Unit Test

The driver can be tested using `dptc_hi` demo located at `LINUX2.6/misc/source/demo/dptc_hi_demo/dptc_hi.c`. (Select Demo Programs -> Available Demo Programs -> DPTC Human Interface Demo option in `menuconfig`).

The demo provides eight options, each based on a DPTC driver command:

1. 1) Enable DPTC—Enables DPTC. When DPTC is enabled, “DPTC module status: Enabled - 1” should appear above the menu.
2. 2) Disable DPTC—Disables DPTC. When DPTC is disabled, “DPTC module status: Enabled - 0” should appear above the menu
3. 3) Update DPTC driver translation table—Updates the driver translation table. When it is chosen, the file name with new table should be provided
4. 4) Read DPTC driver translation table—Dumps the table to a file. When it is chosen, the output file name should be provided.
5. 5) Set DPTC reference circuits—Sets the reference circuit status. When it is chosen, a number in hexadecimal format should be provided. Only 4 LSBs are used, each bit corresponds to one of the four reference circuits. The number says which reference circuits should be enabled and which reference circuits should be disabled.
6. 6) Show DPTC log buffer—Shows the log buffer. The log buffer shows a table with two columns: time point and DPTC working point. It can be used to see that DPTC changes the voltages when it is enabled. The table updates automatically, until some key is pressed to exit from the menu.
7. 7) Set DPTC working point—Sets the DPTC working point. It can be used for debugging only. It is working only when DPTC is disabled. It updates the voltage according to new working point number.
8. 8) Quit—Quits the program.

To test the driver, use Options 1 and 2 to enable/disable the DPTC. Use Option 6 to verify that the voltage is changing. Use Option 5 to select reference circuits. Use Options 3 and 4 to update and dump the driver translation table.

30.8 Unit Test

The driver can be tested using `dptc_hi` demo located at `LINUX2.6/misc/source/demo/dptc_hi_demo/dptc_hi.c`. (Select Demo Programs -> Available Demo Programs -> DPTC Human Interface Demo option in `menuconfig`).

This demo provides several options:

DPTC driver commands tested in the demo

- Enable DPTC—Enables DPTC. When DPTC is enabled, “DPTC module status: Enabled - 1” appears above the menu
- Disable DPTC—Disables DPTC. When DPTC is disabled, “DPTC module status: Enabled - 0” appears above the menu.
- Update DPTC driver translation table—Updates the driver translation table. When this command is chosen, a file name for the new table must be provided.
- Read DPTC driver translation table—Dumps the table into a file. When this command is chosen, the output file name must be provided.
- Set DPTC reference circuits—Sets the reference circuit status. When this command is chosen, a number in hexadecimal format must be provided. Only 4 LSBs are used, and each bit corresponds to one of the four reference circuits. The number determines which reference circuits should be enabled and which reference circuits should be disabled.
- Show DPTC log buffer—Shows the log buffer. The log buffer shows a table with two columns: time point and DPTC working point. It can be used to determine how DPTC changes the voltages when it is enabled. The table updates automatically, until some key is pressed to exit from the menu.
- Set DPTC working point—Sets the DPTC working point. It can be used for debugging only. It works only when DPTC is disabled. It updates the voltage according to the new working point number.
- Quit—quits the program.

For testing whether the driver is working at all, use the enable and disable DPTC commands. Use the Show DPTC Log Buffer command to see how the voltage is changing. Use the Set DPTC Reference Circuit command to select reference circuits. Use the Update DPTC driver translation table command and the Read DPTC driver translation table command to update and dump the driver translation table.

Chapter 31
NAND Flash Boot

NAND Flash memory can be configured as a boot device. This requires an Initial Program Loader (IPL) to be written to the first block of the NAND Flash memory, and the IPL performs the initial boot operations. Because IPL has a size limit, the IPL also loads a Secondary Program Loader (SPL) which initializes the system and loads the OS. This section discusses the steps involved to design the IPL and SPL for NAND Flash memory to perform as a boot device.

31.1 Hardware Operations

The system has several means of booting, including a boot from the NAND Flash. In order to boot from NAND Flash, several pages must first be copied from the NAND Flash to RAM, in particular to the NAND Flash Controller (NFC) internal RAMbuffer. The system is then booted from this RAMbuffer. The BOOTLOADER, which is a part of the NAND Flash control block, is responsible for automatically loading 2 kBytes from the NAND Flash device to the internal 2 kByte RAMbuffer. The AHB Host then reads the code from the internal NAND Flash Controller RAMbuffer in order to initialize the system.

The NAND Flash is not XIP (execute in place) memory, so the boot program cannot be executed directly from NAND Flash memory. The boot code must be copied to NFC's RAMbuffer. The BOOTLOADER starts the data transfer whenever one of the Boot inputs is asserted (NF8BOOT_B or NF16BOOT_B is low.) At System Power-On reset (`ipp_resetb` rising), 2 kBytes of boot code is copied from NAND Flash to RAM.

In NAND Flash, pages are either 512 bytes or 2 kBytes in size. The space required for IPL and SPL is 16 words for 8-bit NAND Flash memory, and 8 words for 16-bit NAND Flash memory.

Each block has 32 pages. The size of the 0th block is 512bytes × 32pages or 2kbytes × 32pages (unused area is not considered). Since only the 0th block is guaranteed by the manufacturer to be a good block, the IPL and the SPL must reside in the 0th block.

31.2 Software Operations

The following sections describe software operations including operations of the IPL and SPL.

31.2.1 Initial Program Loader (IPL)

At system power on reset (POR), the BOOT[4:0] pins are configured for 10000 – 10010, the ARM11 vectors to the NFC Base. If the NFC is configured for booting from Flash, the NFC Bootloader copies 2 kBytes from the NAND Flash to the NFC RAMbuffer, which comes to the NFC Base location in about 160us (estimated). Since the ARM11 is vectored to the RAMbuffer of the NAND, the IPL in the buffer gets executed.

The IPL is divided into two parts, IPL1 and IPL2. IPL1 is the initial portion that assumes control on CPU reset. IPL1 initializes the NFC and the system RAM (SDRAM). Next, IPL1 copies IPL2 from the NFC RAM buffer into SDRAM and passes control to IPL2. IPL2 is required to run from SDRAM and operates

on NAND Flash, so IPL2 is not executed from the NFC RAM Buffer. IPL2 loads the SPL from NAND Flash to SDRAM.

31.2.2 Secondary Program Loader (SPL)

The size of IPL cannot exceed 2K bytes. Because of this, IPL cannot provide boot options and handle bad blocks of NAND Flash at the same time. IPL loads SPL into SDRAM and vectors ARM11 or ARM9 to SPL. The SPL can detect the bad blocks and is responsible for loading the kernel image from the NAND Flash. It first reads the 8 bytes ahead of the kernel image which contains the size and CRC info of the kernel image, and then loads the kernel in RAM.

Bad Block Management: SPL detects whether a block is bad by reading the first page of the block (page 0). For 8-bit memory, this includes 512 bytes per page. In this case, the 517th byte has Bad block Information (BI) In case of 16-bit memory with 512 bytes per page, the 523rd byte will have BI. If the block is bad this byte contains a value other than 0xFF. More information is found in the hardware documentation.

The SPL skips bad blocks and copies good blocks to RAM in sequence.

31.2.3 Flash File System Build

The filesystem layout on the NAND Flash is shown in the diagram below.

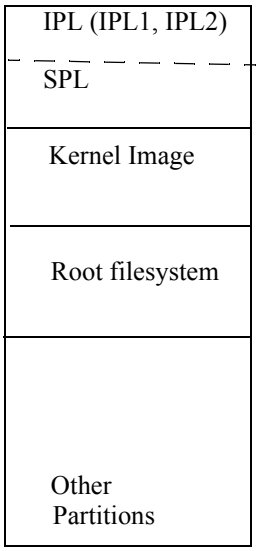

Figure 31-1. NAND Flash File System

31.2.3.1 Creating NAND Filesystem for Evaluation Boards (EVB)

For a NAND boot to an Evaluation board (EVB), follow these steps:

1. Boot into Redboot (For more information, see Redboot documentation shipped with this product)
2. From Redboot, boot the Linux kernel, which contains the NAND MTD driver. The other requirement is that the NOR Flash partition contains the Root filesystem required by the kernel)

3. Using the NAND MTD utilities, do the following:
   — write the `iplspl.bin` file to the first NAND partition
   — write the `Image_crc` kernel image to the second NAND partition, as follows:

```
$ flash_eraseall /dev/mtd/5
$ nandwrite -p /dev/mtd/5 iplspl.bin
$ flash_eraseall /dev/mtd/6
$ nandwrite -p /dev/mtd/6 Image_crc
```

   — Verify the written images as follows:

```
$ mtd_debug read /dev/mtd/5 0 <size_of_iplspl_bin> mtd5
$ cmp -l iplspl.bin mtd5 | more
$ mtd_debug read /dev/mtd/6 0 <size_of_Image_crc> mtd6
$ cmp -l Image_crc mtd6 | more
```

   The comparisons given above should not report any difference in the two files.
4. To use the root filesystem, there are two options.
   — If the kernel is compiled to use the root filesystem from the NOR Flash, then write the root filesystem to the appropriate partition on the NOR Flash using RedBoot.
   — If the kernel is compiled to use the root filesystem from NAND partition 3, then write the root filesystem to the NAND Flash using the NAND MTD utilities (as done in step 3).

Now the NAND Flash is ready to be booted. Resetting the board after disconnecting the debugger should boot the board from NAND Flash.

31.3 Requirements

The NAND boot has the following requirements for IPL, SPL and Filesystem Build:

31.3.1 IPL Requirement

- Initialize the SDRAM.
- Initialize the NFC and read the pages from first block of NAND Flash.
- Copy the SPL to the RAM and pass control to SPL.

31.3.2 SPL Requirement

- Initialize the AIPS and Watchdog.
- Configure Watchdog timer for 2 sec. time-out period and shall service it before 2 seconds of time-out.
- Set up NFC.
- Initialize UART to display text.
- Display options for loading kernel.
   — Load kernel from NAND Flash to RAM and boot from RAM
   — Enter the command line option for Linux kernel
   — Change the address for storing the command line option. Default location is 0x80000100.

- Handle the bad blocks and verify addition CRC checksum while loading the kernel image.

31.4 Source Code Structure

The IPL code starts with an assembly program which initializes SDRAM and then jumps to the external main() function. The main() function calls different routines to initialize the NFC, copies the SPL code to the SDRAM, and passes control to it.

The SPL code starts with an assembly program which initializes the stack and calls the main() function of SPL. The main() function calls routines to initialize AIPS, UART, Watchdog Timer and NFC and downloads the kernel image to a location in the RAM and passes control to it.

The filesystem build is a utility that creates the filesystem image. It writes the IPL, the SPL and the kernel image of different sizes in the image flashmem0.dat.

Table 31-1. Directory list of NANDboot

Directory	Description
bootloader/nandboot	Nandboot specific files
bootloader/bin	Output binary file for nandboot
bootloadet/drivers/uart	Contains MXC UART specific routine
bootloader/drivers/nfc	Contains Nand Flash routines
bootloader/drivers/crcgen	Contains for generating CRC
bootloader/include	Hardware specific routines for initialization
bootloader/platform	Contains code for menu options and setup.

Table 31-2. IPL Files List

File	Description
nandboot/ipl_startup.S	SDRAM Initialization code in assembly
nandboot/ipl_startup2.c	Code which copies the remaining part of the IPL code into memory and jumps to it
nandboot/ipl_link.lds.S	Linker support asm file
nandboot/mxc_nb_ipl.c	IPL source file containing main and NFC routine
bootloadet/bin/ipl1.bin	Output binary file of initial IPL which would execute from NFC RAM buffer.
bootloader/bin/ipl2.bin	Output binary file of secondary IPL which would execute from the SDRAM to further load SPL.
bootloader/bin/iplspl.bin	IPL/SPL image to be used with EVB

Table 31-3. SPL Files List

File	Description
nandboot/spl_startup.S	Initialization code in assembly
nandboot/spl_link.lds.S	Linker support file asm file
platform/boot.c	SPL source file containing main() function
drivers/nfc/mxc_nfc.c	Contains NFC read/write routines
drivers/uart/mxc_uart.c	Contains UART related routines
drivers/uart/mxc_uart.h	Contains UART related defines
drvers/nfc/mxc_nfc.h	Contains NFC related defines
platform/mxc_init.c	Contains watchdog enable,reset and initialization code
include/mxc.h	Contains configurable defines
drivers/uart/spl/mxc_extuart.c	Contains External UART related routines
bin/spl.bin	Output binary file of SPL

Table 31-4. NAND File System Build Utility Files List

File	Description
nfsbuild/mxc_nandfsbuild.c	Contains code to build flashmem file system image

31.5 Configuration

This section describes the required configurations to make a NAND Flash memory bootable.

31.5.1 CPU Board Configuration

For the EVB to boot from NAND Flash, the fuses on the CPU card must be blown to boot from NAND Flash. This requires the IcePick utility, and RVD ICE must be connected to the board. See the IcePick package for further details on blowing the fuse.

For NAND Boot, the fuses SEC_NAND and DIR_BT_DEV must be burned. Currently, these fuses are located at bank 0, address 0x16 (word) of the fuse bank. The example IcePick commands to burn these fuses are given below.

```
% initZas
% source util_fuse_<platform>.tcl
    ; Here <platform> should be replaced with appropriate CPU, like mxc91131,
mxc91231 etc.
% init_iim
% blow_fuse 0 16 1
    ; For blowing DIR_BT_DEV fuse
% blow_fuse 0 16 7
    ; For blowing SEC_NAND fuse

%blow_fuse 0 5 7
    ; For blowing GPIO_BT_SEL fuse. This fuse need to be burnt only for MXC91131
```

CAUTION

Once the fuses are blown the operation is not reversible. See the appropriate fuse definition worksheet for the correct fuse bank, offset and bit number.

31.5.2 Source Code Configuration

31.5.2.1 Chip Configuration Option

The following chip-specific configuration options are provided for a NAND Flash boot.

1. UART Port number (`UART_PORT`)—This defines the UART port to be used for message display.
2. Watchdog Timer—This timer can be enabled or disabled by defining the `WDOG_EN` value in the source code.
3. UART_OUTPUT—This defines either Internal or External UART to be used for output.
4. UART_BAUD_RATE—This configures different baud rates for the UART
5. CHANGE_COMMAND_LINE_OPTION—Enable /Disable command line menu option in Nandboot.
6. CHANGE_COMMAND_LINE_ADDRESS—Enable/Disable change of command line address option in Nandboot.

31.5.2.2 Board Configuration Option

Modify the PLATFORM variable in the `nand_boot/Makefile` with the appropriate value to compile the `nand_boot` for a specific board. The valid values are found in the Makefile.

Example:

```
PLATFORM = value(platform name)
```

Or the “makebuild.sh” a script file which is under the bootloader directory should be invoked as shown below.

```
makebuild.sh “platform_name” boot_type(miniboot or nandboot)
```

31.6 Programming Interface

IPL and SPL are part of the NAND boot module. They need to be placed in the NAND Flash at the appropriate place and the board should be configured to boot from NAND Flash to start execution of the IPL and SPL.

31.7 Unit Test

As this is boot code, once the UART is up, successful test conditions can be verified by displaying messages on the console. Enable UNIT_TEST flag in source files to enable unit testing support.

- Testing transmitter and receiver functions of UART by entering data at the terminal and reprinting it.

- Testing the menu options selected by displaying the option on the terminal through UART.
- Testing the command line options by displaying the option on the terminal through UART.
- Testing the bad kernel by checking its CRC and displaying the message on UART if successful or failure. Displaying the hexadecimal source address of Linux kernel where it is located and destination address from where it should execute.