SCM1612
Wi-Fi 6 and BLE 5 Low-Power SoC

Device Driver Development Guide

Revision 1.3
Date 2025-3-3

Contact Information
Senscomm Semiconductor (www.senscomm.com)
Room 303, International Building, West 2 Suzhou Avenue,
SIP, Suzhou, China
For sales or technical support, please send email to
info@senscomm.com

Disclaimer and Notice

This document is provided on an “as-is” basis only. Senscomm reserves the right to make corrections, improvements and other changes to it or any specification contained herein without further notice.
All liability, including liability for infringement of any proprietary rights, relating to use of information in this document is disclaimed. No licenses express or implied, by estoppel or otherwise, to any intellectual property rights are granted herein.
All third party’s information in this document is provided as is with NO warranties to its authenticity and accuracy.
All trade names, trademarks and registered trademarks mentioned in this document are property of their respective owners and are hereby acknowledged.

Version History

Version	Date	Description
0.1	2023-11-23	Initial draft
1.0	2023-12-22	Added UART, ADC
1.1	2024-10-21	Added I2S
1.2	2024-11-27	Updated SPI
1.3	2025-3-3	Added Audio

Table of Contents

1 Introduction

This document serves as a comprehensive guide for implementing applications utilizing the SoC peripherals through the SCM1612 device driver. It is designed to assist developers in leveraging the capabilities of the SCM1612 Wi-Fi 6 and BLE 5 Low-Power SoC, focusing on the effective use of Device Driver APIs and their integration into performance-critical applications.

1.1 Device Driver APIs

The SCM1612 SDK provides two distinct levels of APIs:

- Lower Layer APIs
  - Located in: hal/drivers/xxxx/xxxx.c
  - Headers in: include/hal/xxxx.h
  - Purpose: These APIs offer hardware abstraction, primarily involving direct access to hardware registers.
- Application Layer APIs (SCM APIs)
  - Located in: api/scm_xxxx.c
  - Headers in: api/include/scm_xxxx.h
  - Functionality: SCM APIs facilitate easy access to functionalities through generic functions, encapsulating calls to one or more Lower Layer APIs, operating system functions, and basic error handling.

1.2 Performance Critical Applications

While SCM APIs simplify development by adding a layer of abstraction over Lower Layer APIs, this might introduce some overhead. For resource-intensive or time-sensitive applications, direct use of Lower Layer APIs is recommended. For example, in scenarios where a custom board requires simultaneous communication over SPI and I2C with an external IC, creating a bespoke device driver utilizing the Lower Layer APIs for SPI and I2C is advisable. Applications can then interface with this custom driver instead of the higher-level SCM APIs.

1.3 Driver Notification Callback

In the SCM1612 SDK, certain peripherals may take time to complete operations initiated by the user. To address this, the SDK offers two categories of APIs:

- Synchronous APIs (scm_xxxx) : These APIs include a timeout parameter and are designed for operations where the application can wait for completion within a specified time frame.
- Asynchronous APIs (scm_xxxx_async): These APIs do not include a timeout parameter and are suitable for operations where waiting for completion is not feasible. The completion of these operations is communicated through notification callbacks.

For asynchronous operations, completion notifications are delivered via callback functions. These callbacks are triggered within interrupt contexts and should be designed to execute quickly to avoid delays in interrupt handling. It's important to note that usage of other APIs within these callbacks might be restricted due to the context in which they are executed. Generally, for most use cases, synchronous APIs are recommended for their simplicity and ease of integration into applications.

2 PIN Control

Some of the peripherals require SoC pins such SPI, I2C, GPIO, and PWM.

2.1 Pin Mux Options

The available pin mux options are listed in the following file

- hal/drivers/pinctrl/pinctrl-scm2010.c

Each SoC pin can be configured for one of several predefined functions.

For instance, UART0 and UART1 have the following pin mux options:

```
#ifdef CONFIG_USE_UART0
struct pinmux uart0_pin_mux[] = {
    MUX("cts", 19, 3),
    MUX("rts", 20, 3),
    MUX("rxd",  0, 4),
    MUX("rxd", 21, 0),
    MUX("txd",  1, 4),
    MUX("txd", 22, 0),
    MUX("txd",  8, 4),
};
#endif

#ifdef CONFIG_USE_UART1
struct pinmux uart1_pin_mux[] = {
    MUX("cts", 19, 4),
    MUX("rts", 20, 4),
    MUX("rxd",  0, 0),
    MUX("rxd",  6, 4),
    MUX("rxd", 17, 4),
    MUX("txd",  1, 0),
    MUX("txd",  7, 4),
    MUX("txd",  8, 3),
    MUX("txd", 18, 4),
};
#endif
```

2.2 Pin Mux Selection

Pin selection is board-specific, as each board design uses pins differently. For example, the SCM2010 evaluation board selects pins for UART as follows, found in:

- hal/board/scm2010-evb-qfn40/board.c

```
static struct pinctrl_pin_map pin_map[] = {
#ifdef CONFIG_USE_UART0
    /* UART0 */
    pinmap(21, "atcuart.0", "rxd", 0),
    pinmap(22, "atcuart.0", "txd", 0),
#endif

#ifdef CONFIG_USE_UART1
    /* UART1 */
    pinmap( 0, "atcuart.1", "rxd", 0),
    pinmap( 1, "atcuart.1", "txd", 0),
#endif
```

Peripheral drivers, such as the UART driver, typically request the pin driver to configure the pin mux automatically.

3 UART

3.1 Overview

The SCM1612 SoC supports Universal Asynchronous Receiver Transmitter (UART) to facilitate communication with external devices. The SoC features three UART instances, with one of them configurable as a console to aid in firmware development.

3.2 API Functions

- Initialization and Configuration:
  - `scm_uart_init()`: Initializes the UART module.
  - `scm_uart_deinit()`: Deinitializes the UART module.
  - `scm_uart_reset()`: Resets the UART module.
- Transmittor and Receiver:
  - Data Transmission: `scm_uart_tx()`, `scm_uart_tx_async()`
  - Data Reception: `scm_uart_rx()`, `scm_uart_rx_async()`

3.3 Development Guide

[Configuration Structure]

```
struct scm_uart_cfg {
    enum scm_uart_buadrate buadrate;
    enum scm_uart_data_bits data_bits;
    enum scm_uart_parity parity;
    enum scm_uart_stop_bits stop_bits;
    uint8_t dma_en;
};
```

- baudrate: Selects the baud rate (ranging from 50 to 115200).
- data_bits: Chooses the length of data bits (5, 6, 7, or 8 bits)
- parity: Sets parity (options: no parity, odd parity, even parity).
- stop bits: Determines the length of stop bits (1 bit or 2 bits).
- dma_en: Enables or disables DMA transfer.

[Events Handling]

- SCM_UART_EVENT_TX_CMPL: Triggered upon the completion of UART transmission using asynchronous API.
- SCM_UART_EVENT_RX_CMPL: Triggered upon the completion of UART reception using asynchronous API.

Usage Steps for Transmission (Tx) and Reception (Rx):

Step 1: **Initialize the UART** with the required configuration.

Step 2: **Transmit or Receive data** to/from the connected devices.

Step 3: **Deinitialize the UART** when it is no longer in use.

```
void sample_spi(void)
{
    static struct scm_uart_cfg cfg = {
        .buadrate = SCM_UART_BDR_115200,
        .data_bits = SCM_UART_DATA_BITS_8,
        .pairty = SCM_UART_NO_PARITY,
        .stop_bits = SCM_UART_STOP_BIT_1,
        .dma_en = 1,
    };
    int ret;

    scm_uart_init(SCM_UART_IDX_0, &cfg);

    scm_uart_tx(SCM_UART_IDX_0, tx_buf, tx_len, 1000);
    scm_uart_rx(SCM_UART_IDX_0, rx_buf, rx_len, 1000);

    scm_uart_deinit(SCM_UART_IDX_0);
}
```

3.4 Precautions

4 SPI (Serial Peripheral Interface)

4.1 Overview

The Serial Peripheral Interface (SPI) of SCM1612 supports both master and slave modes, enabling communication with external devices. It can be configured for standard SPI or Quad SPI (QSPI) for interfacing with Quad SPI compatible devices.

SPI mater supports command, address and data phase and the data can be transferred without command and address phase.
To receive or send data through the SPI slave, the peer master must send 8bits command and 8bits dummy data via single IO before sending or receiving actual data.

4.2 API Functions

- Initialization and Configuration:
  - `scm_spi_init()`: Initializes the SPI module.
  - `scm_spi_deinit()`: Deinitializes the SPI module.
  - `scm_spi_configure()`: Sets up SPI parameters and options.
  - `scm_spi_reset()`: Resets the SPI module.
- Master Mode Operations:
  - Data Transmission: `scm_spi_master_tx()`, `scm_spi_master_tx_async()`
  - Data Reception: `scm_spi_master_rx()`,`scm_spi_master_rx_async()`
  - Combined Transmission and Reception:` scm_spi_master_tx_rx()`,`scm_spi_master_tx_rx_async()`
  - Command-based Operations:` scm_spi_master_tx_with_cmd()`,`scm_spi_master_rx_with_cmd()`,etc .
- Slave Mode Operations:
  - Buffer Management: `scm_spi_slave_set_tx_buf()`, `scm_spi_slave_set_rx_buf()`, `scm_spi_slave_set_tx_rx_buf()`
  - User State Management: `scm_spi_slave_set_user_state()`

4.3 Development Guide

[Configuration]

```
struct scm_spi_cfg {
    enum scm_spi_role role;
    enum scm_spi_mode mode;
    enum scm_spi_data_io_format data_io_format;
    enum scm_spi_bit_order bit_order;
    enum scm_spi_dummy_cycle slave_extra_dummy_cycle;
    uint32_t master_cs_bitmap;
    enum scm_spi_clk_src clk_src;
    uint8_t clk_div_2mul;
    uint8_t dma_en;
}
```

- role: select master or slave
- mode: CPOL and CPHA mode
  - SCM_SPI_MODE_0: active high, odd edge sampling
  - SCM_SPI_MODE_1: active high, even edge sampling
  - SCM_SPI_MODE_2: active low, odd edge sampling
  - SCM_SPI_MODE_3: active low, even edge sampling
- data_io_format: select io format
  - SPI_DATA_IO_FORMAT_SIGNLE
  - SPI_DATA_IO_FORMAT_DUAL
  - SPI_DATA_IO_FORMAT_QUAD

  note : scm_spi_master_tx_rx and scm_spi_master_tx_rx_sync cannot be used in DUAL or QUAD IO format.
- bit_order: select bit order
  - SPI_BIT_MSB_FIRST
  - SPI_BIT_LSB_FIRST
- slave_extra_dummy_cycle
  - SCM_SPI_DUMMY_CYCLE_NONE
  - SCM_SPI_DUMMY_CYCLE_SINGLE_IO_8
  - SCM_SPI_DUMMY_CYCLE_SINGLE_IO_16
  - SCM_SPI_DUMMY_CYCLE_SINGLE_IO_24
  - SCM_SPI_DUMMY_CYCLE_SINGLE_IO_32
  - SCM_SPI_DUMMY_CYCLE_DUAL_IO_4
  - SCM_SPI_DUMMY_CYCLE_DUAL_IO_8
  - SCM_SPI_DUMMY_CYCLE_DUAL_IO_12
  - SCM_SPI_DUMMY_CYCLE_DUAL_IO_16
  - SCM_SPI_DUMMY_CYCLE_QUAD_IO_2
  - SCM_SPI_DUMMY_CYCLE_QUAD_IO_4
  - SCM_SPI_DUMMY_CYCLE_QUAD_IO_6

  - SCM_SPI_DUMMY_CYCLE_QUAD_IO_8
- master_cs_bitmap: Bitmap of GPIO numbers to be used for chip-selects
- clk_src: select SPI IO clock source
  - SPI_CLK_SRC_XTAL: XTAL 40Mhz
  - SPI_CLK_SRC_PLL: PLL clock. SPI0 and SPI1 use 240Mhz PLL clock, SPI2 uses 80Mhz PLL clock
- clk_div_2mul: source clock divide value. Divide the setting value by multiplying it by 2.
  ex) clk_src = SPI_CLK_SRC_XTAL and clk_div_2mul = 20,
  SPI IO clock io is 1Mhz.
- dma_en: enable or disable DMA transfer

[Events]

- Master event
  - SCM_SPI_EVENT_MASTER_TRANS_CMPL : SPI transfer complete event. This event is triggered for asynchronous API
- Slave event
  - SCM_SPI_EVENT_SLAVE_TX_REQUEST : Occurs when read command is received.
  - SCM_SPI_EVENT_SLAVE_RX_REQUEST : Occurs when read command is received.
  - SCM_SPI_EVENT_SLAVE_USER_CMD: Occurs when an undefined command is received.
  - SCM_SPI_EVENT_SLAVE_TRAN_CMPL: SPI transfer complete event.

Usage Steps for Master Mode:
Step 1: Initialize the SPI

Step 2: Configure the SPI as a master role

Step 3: Transmit or Receive data to and from the slave devices

Step 4: Deinitialize the SPI when it is no longer used

```
void sample_spi(void)
{
    struct scm_spi_cfg cfg;
```

```
    int ret;

    memset(&cfg, 0, sizeof(cfg));
    cfg.role = SCM_SPI_ROLE_MASTER;
    cfg.mode = SCM_SPI_MODE_0;
    cfg.data_io_format = SCM_SPI_DATA_IO_FORMAT_SINGLE;
    cfg.bit_order = SCM_SPI_BIT_ORDER_MSB_FIRST;
    cfg.clk_src = SCM_SPI_CLK_SRC_XTAL,
    cfg.clk_div_2mul = 20,
    cfg.dma_en = 0,

    scm_spi_init(SCM_SPI_IDX_0);
    scm_spi_configure(SCM_SPI_IDX_0, &cfg, NULL, NULL);

    scm_spi_master_tx(SCM_SPI_IDX_0, 0, tx_buf, tx_len, 1000);
    scm_spi_master_rx(SCM_SPI_IDX_0, 0, rx_buf, rx_len, 1000);
    scm_spi_master_tx_rx(SCM_SPI_IDX_0, 0, tx_buf, tx_len, rx_buf, rx_len,
     1000);

    scm_spi_deinit(SCM_SPI_IDX_0);
}
```

Usage Steps for Slave Mode:

Step 1: Initialize the SPI.

Step 2: Configure SPI as Slave.

Step 3: Manage Tx/Rx buffers and respond to events.

Step 4: Deinitialize SPI after use.

```
int volatile g_spi_complete;

void spi_slave_complete_wait(void)
{
    g_spi_complete = 0;
    while (1) {
        if (g_spi_complete) {
        }
    }
}

Int spi_slave_notify(struct scm_spi_event *event, void *ctx)
{
```

```
    Switch (event->type) {
    case SCM_SPI_EVENT_SLAVE_TRANS_CMPL:
        g_spi_complete = 1;
        break;
    default:
        break;
    }

    return 0;
}

void sample_spi(void)
{
    struct scm_spi_cfg cfg;
    int ret;

    memset(&cfg, 0, sizeof(cfg));
    cfg.role = SCM_SPI_ROLE_SLAVE;
    cfg.mode = SCM_SPI_MODE_0;
    cfg.data_io_format = SCM_SPI_DATA_IO_FORMAT_SINGLE;
    cfg.bit_order = SCM_SPI_BIT_ORDER_MSB_FIRST;
    cfg.clk_src = SCM_SPI_CLK_SRC_XTAL,
    cfg.clk_div_2mul = 20,
    cfg.dma_en = 0,

    scm_spi_init(SCM_SPI_IDX_0);
    scm_spi_configure(SCM_SPI_IDX_0, &cfg, spi_slave_notify, NULL);

    scm_spi_slave_set_tx_buf(SCM_SPI_IDX_0, tx_buf, tx_len);
    scm_spi_slave_complete_wait();

    scm_spi_slave_set_rx_buf(SCM_SPI_IDX_0, rx_buf, rx_len);
    scm_spi_slave_complete_wait();

    scm_spi_slave_set_tx_rx_buf(SCM_SPI_IDX_0, tx_buf, tx_len,
                                rx_buf, rx_len);
    scm_spi_slave_complete_wait();


    scm_spi_deinit(SCM_SPI_IDX_0);
}
```


4.4 Precautions

- SPI Instances:

SCM1612 provides three SPI instances. SPI0 is dedicated to flash memory and is not available for other uses.

- DMA Usage:

If using DMA, ensure transmission and reception buffers are in the DMA-allowed area.

- Slave Mode Considerations:

The notification function in Slave mode should execute quickly to avoid transaction failures due to master clock delays.

- Multiple slaves support as a slave:

The SPI driver should support this feature, e.g., by enabling CONFIG_SPI_SUPPORT_MULTI_SLAVES_AS_A_SLAVE for atcspi driver.
And the master must provide enough dummy cycles between command and data, which should also be configured in the slave as slave_extra_dummy_cycle parameter.
Number of dummy cycles to be required depends on SPI clock speed.

- Multiple slaves support as a master:

The SPI driver should support this feature, e.g., by enabling CONFIG_SPI_SUPPORT_MULTI_SLAVES_AS_A_MASTER for atcspi driver.
And the master must be supplied with the bitmap of GPIO numbers via master_cs_bitmap parameter.
For example, if GPIO0 and GPIO15 are to be used to control slave1 and slave2, respectively, master_cs_bitmap must be 0x00008001.

5 I2C (Inter-Integrated Circuit)

5.1 Function Description

The SCM1612 SoC supports I2C in both master and slave modes, facilitating communication with various external devices. The I2C interface is designed to handle standard data transfer protocols, ensuring compatibility with a wide range of I2C-compatible devices.

5.2 API Functions

- Initialization and Configuration:
  - `scm_i2c_init()`: Initializes the I2C module.
  - `scm_i2c_deinit()`: Deinitializes the I2C module.
  - `scm_i2c_configure()`: Configures I2C parameters and options.
  - `scm_i2c_reset()`: Resets the I2C module.

- Master Mode Operations:
  - Data Transmission: `scm_i2c_master_tx()`, `scm_i2c_master_tx_async()`
  - Data Reception: `scm_i2c_master_rx()`, `scm_i2c_master_rx_async()`
  - Combined Transmission and Reception: `scm_i2c_master_tx_rx()`, `scm_i2c_master_tx_rx_async()`
  - Device Probing: `scm_i2c_master_probe()`

- Slave Mode Operations:
  - Transmit Request: `scm_i2c_slave_tx()`
  - Receive Request: `scm_i2c_slave_rx()`

5.3 Development Guide

[Events]

- Master events
  - SCM_I2C_EVENT_MASTER_TRANS_CMPL: when the master completes the trasfer
- Slave events

- SCM_I2C_EVENT_SLAVE_RX_REQUEST: when the slave receives the RX (master to slave) data request
- SCM_I2C_EVENT_SLAVE_TX_REQUEST: when the slave receives the TX (slave to master) data request
- SCM_I2C_EVENT_SLAVE_RX_CMPL: when the slave completes the reception
- SCM_I2C_EVENT_SLAVE_TX_CMPL: when the slave completes the transmission.

Usage Steps for Master Mode:
Step 1: Initialize the I2C

Step 2: Configure I2C as Master.

Step 3: Perform Transmit/Receive operations.

Step 4: Deinitialize I2C after use.

```
static int i2c_master_notify(struct scm_i2c_event *event, void *ctx)
{
    return 0;
}

void sample_i2c(void)
{
    struct scm_i2c_cfg cfg;
    int ret;

    memset(&cfg, 0, sizeof(cfg));
    cfg.role = SCM_I2C_ROLE_MASTER;
    cfg.master_clock = 400 * 1000;
    cfg.pull_up_en = 1;

    scm_i2c_init(SCM_I2C_IDX_0);
    scm_i2c_configure(SCM_I2C_IDX_0, &cfg, i2c_master_notify, NULL);

    scm_i2c_master_tx(SCM_I2C_IDX_0, SLAVE_DEVICE_ADDR, tx_buf, tx_len, 1000);
    scm_i2c_master_tx_rx(SCM_I2C_IDX_0, SLAVE_DEVICE_ADDR,
                         tx_buf, tx_len, buf, len, 1000);
    scm_i2c_master_rx(SCM_I2C_IDX_0, SLAVE_DEVICE_ADDR, buf, len, 1000);

    scm_i2c_deinit(SCM_I2C_IDX_0);
```

```
}
```

Usage Steps for Slave Mode:
Step 1: Initialize the I2C

Step 2: Configure I2C as Slave.

Step 3: When I2C notify functions is called, prepare Tx or Rx data

Step 4: Deinitialize I2C after use.

Sample:

```
static int i2c_slave_notify(struct scm_i2c_event *event, void *ctx)
{
    switch (event->type) {
    case SCM_I2C_EVENT_SLAVE_TX_REQUEST:
              scm_i2c_slave_tx(SCM_I2C_IDX_1, tx_buf, tx_len);
    break;
    case SCM_I2C_EVENT_SLAVE_RX_REQUEST:
           scm_i2c_slave_rx(SCM_I2C_IDX_1, rx_buf, rx_len);
    break;
    case SCM_I2C_EVENT_SLAVE_TX_CMPL:
        len = event->data.slave_tx_cmpl.len;
        /* do something about tx data */
    break;
    case SCM_I2C_EVENT_SLAVE_RX_CMPL:
        len = event->data.slave_rx_cmpl.len;
        /* do something about rx data */
    break;
    default:
    break;
    }
    return 0;
}

void sample_i2c(void)
{
    struct scm_i2c_cfg cfg;
    int ret;

    memset(&cfg, 0, sizeof(cfg));
    cfg.role = SCM_I2C_ROLE_SLAVE;
    cfg.pull_up_en = 1;
```

```
    scm_i2c_init(SCM_I2C_IDX_0);
    scm_i2c_configure(SCM_I2C_IDX_1, &cfg, i2c_slave_notify, NULL);

    ...

    scm_i2c_deinit(SCM_I2C_IDX_1);
}
```

5.4 Precautions

For the slave role, if the user supplied buffer is not enough,

- When master tries to read beyond the buffer, 0 will be returned to master
- When master tries to write beyond the buffer, the data is silently dropped.

6 Timer

6.1 Overview

The SCM1612 SoC is equipped with robust timer functionalities, offering two hardware timers. Each timer features four channels, enabling them to be configured in various modes. This flexibility allows the creation of four independent timers for diverse application needs.

6.2 Function Description

- scm_timer_configure: Configures the timer mode
- scm_timer_start: Starts the timer
- scm_timer_stop: Stops the timer
- scm_timer_start_multi: Starts multiple timers at the same time
- scm_timer_stop_multi: Stops multiple timers at the same time
- scm_timer_value: Reads the value of timer if it is configured as free-run mode

6.3 Development Guide

Perform the following steps.

Step 1: Configure the timers

Step 2: Start the timers

Step 3: Read the timers

Step 4: Stop the timers

Sample:

```
static uint8_t timer_id[3] = { 0, 1, 2};

static int timer_notify(uint32_t pin, void *ctx)
{
```

```
    uint8_t timer_id = *((uint8_t *)ctx);
    /* differentiate the notification based on the context provided */
    return 0;
}

void sample_timer(void)
{
    struct scm_timer_cfg cfg0, cfg1, cfg2;
    uint32_t value;

    cfg0.mode = SCM_TIMER_MODE_PERIODIC;
    cfg0.intr_en = 1;
    cfg0.data.periodic.duration = 1000000;

    cfg1.mode = SCM_TIMER_MODE_ONESHOT;
    cfg1.intr_en = 1;
    cfg1.data.oneshot.duration = 3000000;

    cfg2.mode = SCM_TIMER_MODE_FREERUN;
    cfg2.data.freerun.freq = 1000000;

    scm_timer_configure(SCM_TIMER_IDX_0, SCM_TIMER_CH_0, timer_notify, &cfg0,
&timer_id[0]);
    scm_timer_configure(SCM_TIMER_IDX_1, SCM_TIMER_CH_1, timer_notify, &cfg1,
&timer_id[1]);

    scm_timer_start(SCM_TIMER_IDX_0, SCM_TIMER_CH_0);
    scm_timer_start(SCM_TIMER_IDX_0, SCM_TIMER_CH_1);

    ...
    scm_timer_value(SCM_TIMER_IDX_0, SCM_TIMER_CH_2, &value);
    ...

    scm_timer_stop_multi(SCM_TIMER_IDX_0,  1  <<  SCM_TIMER_CH_0  |  1  <<
SCM_TIMER_CH_1);
}
```

6.4 Precautions

The TIMER1 instance is used by the power management and BLE subsystems. When PM or BLE is used. When PM or BLE is enabled, the application should not call the timer API with the index SCM_TIMER_IDX_1

7 GPIO (General Purpose Input/Output)

7.1 Overview

The SCM1612 SoC is equipped with 25 General Purpose Input/Output (GPIO) pins. These GPIOs can be configured as inputs or outputs, offering versatile functionality for interfacing with external hardware and sensors.

7.2 API Functions

- Configuration and Control:
  - `scm_gpio_configure()`: Configures a GPIO pin as input or output, with options for pull-up or pull-down resistors.
  - `scm_gpio_write()`: Sets the output level of a GPIO pin.
  - `scm_gpio_read()`: Reads the input level of a GPIO pin.
  - `scm_gpio_enable_interrupt()`: Enables GPIO interrupts, allowing for event-driven programming.
  - `scm_gpio_disable_interrupt()`: Disables GPIO interrupts.

7.3 Development Guide

Basic Steps for GPIO Usage:
Step 1: Configure the GPIO pin for the intended input/output functionality.

Step 2: For output usage, write the desired level to the GPIO pin.

Step 3: For input usage, read the level from the GPIO pin as required.

Step 4: If using interrupts, enable them and provide a callback function to handle events.

Sample Code:

```
static int scm_cli_gpio_notify(uint32_t pin, void *ctx)
{
    gpio_stats[pin]++;
    return 0;
}

void sample_gpio(void)
{
    /* set PIN 6 as gpio output */
    scm_gpio_configure(6, SCM_GPIO_PROP_OUTPUT);
    scm_gpio_write(6, 1);

    /* set PIN 7 as gpio input, and enable interrupt */
    scm_gpio_configure(7, SCM_GPIO_PROP_INPUT_PULL_DOWN);
    scm_gpio_enable_interrupt(7, SCM_GPIO_INT_BOTH_EDGE,
                              scm_cli_gpio_notify, NULL);
}
```

7.4 Precautions

- Power Domains:

GPIO pins 0 to 7 are in the "Always ON" power domain, meaning they can maintain their state and receive wake-up events during low power modes.

- Interrupt Handling:

Care should be taken to ensure that interrupt callback functions are efficient and do not block critical tasks.

8 eFuse

8.1 Function Description

SCM1612's eFuse (electrically programmable fuses) provides 1024 bits of non-volatile memory. This feature is typically used for storing system-critical data such as secure boot keys, device identity, and RF calibration data.

8.2 API Functions

- eFuse Operations:
  - `scm_efuse_read()`: Reads data from a specified eFuse address.
  - `scm_efuse_write()`: Writes data to a specified eFuse address.

8.3 Development Guide

The reserved fields and their size are shown below.

```
#define SCM_EFUSE_ADDR_ROOT_KEY        0
#define SCM_EFUSE_SIZE_ROOT_KEY        128

#define SCM_EFUSE_ADDR_PARITY          128
#define SCM_EFUSE_SIZE_PARITY          1

#define SCM_EFUSE_ADDR_HARD_KEY        129
#define SCM_EFUSE_SIZE_HARD_KEY        1

#define SCM_EFUSE_ADDR_FLASH_PROT      130
#define SCM_EFUSE_SIZE_FLASh_PROT      1

#define SCM_EFUSE_ADDR_SECURE_BOOT     131
#define SCM_EFUSE_SIZE_SECURE_BOOT     1

#define SCM_EFUSE_ADDR_AR_BL_EN        132
#define SCM_EFUSE_SIZE_AR_BL_EN        1

#define SCM_EFUSE_ADDR_SDIO_OCR_EN     132
#define SCM_EFUSE_SIZE_SDIO_OCR_EN     1
```

```
#define SCM_EFUSE_ADDR_AR_FW_EN        133
#define SCM_EFUSE_SIZE_AR_FW_EN        1

#define SCM_EFUSE_ADDR_CUST_ID         160
#define SCM_EFUSE_SIZE_CUST_ID         8

#define SCM_EFUSE_ADDR_CHIP_ID         192
#define SCM_EFUSE_SIZE_CHIP_ID         32

#define SCM_EFUSE_ADDR_PK_HASH         224
#define SCM_EFUSE_SIZE_PK_HASH         256

#define SCM_EFUSE_ADDR_SDIO_OCR        544
#define SCM_EFUSE_SIZE_SDIO_OCR        20

#define SCM_EFUSE_ADDR_WLAN_MAC_ADDR    576
#define SCM_EFUSE_SIZE_WLAN_MAC_ADDR    48

#define SCM_EFUSE_ADDR_BLE_MAC_ADDR     624
#define SCM_EFUSE_SIZE_BLE_MAC_ADDR     48

#define SCM_EFUSE_ADDR_RF_CAL          672
#define SCM_EFUSE_SIZE_RF_CAL          64

#define SCM_EFUSE_ADDR_AL_BL_VER       736
#define SCM_EFUSE_SIZE_AL_BL_VER       64

#define SCM_EFUSE_ADDR_RESERVED        800
#define SCM_EFUSE_SIZE_RESERVED        224
```

Sample:

```
uint8_t mac[8];
uint8_t custom_data[2];

void sample_efuse(void)
{
    scm_efuse_read(SCM_EFUSE_ADDR_WLAN_MAC_ADDR,
                SCM_EFUSE_SIZE_WLAN_MAC_ADDR,
                mac);

    scm_efuse_write(800, 16, &custom_data);
}
```

8.4 Precautions

As eFuse can be written only once, care must be taken when writing new values.

Each bit of eFuse can be turned from 0 to 1, not the other way around. In special circumstances, the same field can be written multiple times to set the specific bits into 1.

9 ADC

9.1 Function Description

The SCM1612 SoC features an Analog to Digital Converter (ADC) capable of reading analog voltage levels and converting these values to digital format. The table below presents the mapping between the physical GPIO pins and their corresponding ADC channels.

ADC Channel	GPIO Pin
4	4
5	7
6	0
7	1

9.2 API Functions

- ADC measurement:
  - `scm_adc_read()`, `scm_adc_read_aync()`: Reads data from a specified ADC channel.
  - `scm_adc_reset()` : Resets the ADC

9.3 Development Guide

Sample Code:

```
static int adc_notify(void *ctx)
{
    return 0;
}

uint16 ch0_buf[8];
uint16 ch1_buf[16];

void sample_efuse(void)
```

```
{
    scm_adc_read(SCM_ADC_SINGLE_CH_0, ch0_buf, 8);
    scm_adc_read_async(SCM_ADC_SINGLE_CH_1, ch1_buf, 16, adc_notify, NULL);
}
```

This sample demonstrates how to perform synchronous and asynchronous ADC readings. The scm_adc_read function is used for synchronous reading from channel 0, while scm_adc_read_async is utilized for asynchronous reading from channel 1 with a callback function.

9.4 Precautions

10 I2S

10.1 Overview

The I2S of SCM1612 supports both master and slave modes, enabling exchange of audio sample data with an external audio CODEC. It supports various formats and word lengths on I2S bus as described below.

10.2 API Functions

- Initialization and Configuration:
  - `scm_i2s_init()`: Initializes the I2S module.
  - `scm_i2s_deinit()`: Deinitializes the I2S module.
  - `scm_i2s_configure()`: Sets up I2S parameters and options.
  - `scm_i2s_start()`: Starts I2S data stream(s) in specified direction(s).
  - `scm_i2s_stop()`: Stops I2S data stream(s) in specified direction(s).
  - `scm_i2s_read_block()`: Reads one block of audio samples from the I2S input stream, returning success or error codes.
  - `scm_i2s_write_block()`: Writes one block of audio samples to the I2S output stream, returning success or error codes.
  - `scm_i2s_get_block_buffer_size()`: Returns configured block buffer size.

10.3 Development Guide

[Configuration]

```
struct scm_i2s_cfg {
    enum scm_i2s_wl word_length; // Number of bits per audio word (channel)
    enum scm_i2s_fmt format;     // I/O format
    enum scm_i2s_role role;      // Bus role (master/slave)
    enum scm_i2s_direction dir;  // Data transfer direction (Tx/Rx)
    uint32_t fs;                 // Audio sampling frequency in Hz
    int duration_per_block;      // Time duration per buffer block in ms
    int number_of_blocks;        // Max number of buffer blocks to allocate
    int timeout;                 // Read/write timeout in ms
}
```

- word_length: Number of bits per a single audio word, i.e., channel
  - SCM_I2S_WL_16: 16 bits
  - SCM_I2S_WL_20: 20 bits
  - SCM_I2S_WL_24: 24 bits

Note that word length of 20 or 24 bits will occupy 32 bits in memory.

- format: Select io format
  - SCM_I2S_FMT_I2S
  - SCM_I2S_FMT_LJ
  - SCM_I2S_FMT_RJ
- role: Select a role on the bus
  - SCM_I2S_ROLE_MASTER : Generates I2S bit clock and word clock.
  - SCM_I2S_ROLE_SLAVE
- dir: Specify data transfer direction
  - SCM_I2S_RX
  - SCM_I2S_TX
- fs: Audio sampling frequency in Hz
- duration_per_block: Time duration corresponding to one buffer block in milliseconds
  Note: Must be less than 1000 (1 second).
- number_of_blocks: Maximum number of buffer blocks to allocate
- timeout: Read/Write timeout in milliseconds

Usage Steps for Transmission:

1: Initialize the I2S.

2: Configure the I2S both for Tx and Rx directions.

3: Start I2S stream in Tx direction.

4: Write audio sample blocks to the I2S Tx stream.

5: Stop Tx stream and deinitialize I2S when no longer used.

```
void sample_i2s_write(uint32 pattern)
{
    struct scm_i2s_cfg cfg;
    int bufsz;
    uint8_t *buf;

    scm_i2s_init();

    memset(&cfg, 0, sizeof(cfg));
    cfg.word_length = SCM_I2S_WL_16;
    cfg.format = SCM_I2S_FMT_I2S;
    cfg.role = SCM_I2S_ROLE_MASTER;
    cfg.fs = 44100;
    cfg.duration_per_block = 100;
    cfg.number_of_blocks = 5;
    cfg.timeout = 3000;

    cfg.dir = SCM_I2S_RX;
    scm_i2s_configure(&cfg);
```

```
    cfg.dir = SCM_I2S_TX;
    scm_i2s_configure(&cfg);

    bufsz = scm_i2s_get_block_buffer_size(&cfg);
    buf = zalloc(bufsz);

    for (int i = 0; i < bufsz / 4; i = i + 4) {
        memcpy(buf + i, &pattern, sizeof(uint32_t));
    }

    scm_i2s_start(SCM_I2S_TX);

    for (int i = 0; i < cfg.number_of_blocks; i++) {
        scm_i2s_write_block(buf, bufsz);
    }

    scm_i2s_stop(SCM_I2S_TX);

    free(buf);

    scm_i2s_deinit();
}
```

Usage Steps for Reception:

1: Initialize the I2S.

2: Configure the I2S both for Tx and Rx directions.

3: Start I2S stream in Rx direction.

4: Read audio sample blocks from the I2S Rx stream.

5: Stop Rx stream and deinitialize I2S when no longer used.

```
void sample_i2s_read(void)
{
    struct scm_i2s_cfg cfg;
    int bufsz, len;
    uint8_t *buf;

    scm_i2s_init();

    memset(&cfg, 0, sizeof(cfg));
    cfg.word_length = SCM_I2S_WL_16;
    cfg.format = SCM_I2S_FMT_I2S;
    cfg.role = SCM_I2S_ROLE_MASTER;
    cfg.fs = 44100;
    cfg.duration_per_block = 100;
    cfg.number_of_blocks = 5;
```

```
    cfg.timeout = 3000;

    cfg.dir = SCM_I2S_RX;
    scm_i2s_configure(&cfg);

    cfg.dir = SCM_I2S_TX;
    scm_i2s_configure(&cfg);

    bufsz = scm_i2s_get_block_buffer_size(&cfg);
    buf = zalloc(bufsz);

    scm_i2s_start(SCM_I2S_RX);

    while (1) {
        ret = scm_i2s_read_block(buf, &len);
        if (ret) {
            if (ret == WISE_ERR_NO_MEM) {
                printf("I2S is not running.\n");
            }
            Break;
        }
        /* Do something with buf */
    }
    Scm_i2s_stop(SCM_I2S_RX);

    free(buf);

    scm_i2s_deinit();
}
```

10.4 Precautions

- **Memory usage:**
  SCM1612 has limited memory available for I2S streaming. Adjust cfg.duration_per_block and cfg.number_of_blocks accordingly to prevent memory overflow.
- **Configuration:**
  Both I2S Tx and Rx directions must be configured regardless of the actual data transfer direction. Ensure that the parameters for both directions are identical.

11 AUDIO

11.1 Overview

The CODEC API is provided to control an external audio CODEC.
It will usually be used together with the I2S API.

11.2 API Functions

- Initialization and Configuration:
  - `scm_audio_init()`: Initializes the Audio CODEC.
  - `scm_audio_deinit()`: Deinitializes the Audio CODEC.
  - `scm_audio_configure()`: Sets up Audio CODEC parameters.
  - `scm_audio_start()`: Starts the specified Audio CODEC interface.
  - `scm_audio_stop()`: Stops the specified Audio CODEC interface.
  - `scm_audio_get_volume()`: Get the current volume.
  - `scm_audio_set_volume()`: Set the current volume.
  - `scm_audio_mute()`: Mutes the specified Audio CODEC interface.
  - `scm_audio_unmute()`: Unmutes the specified Audio CODEC interface.

11.3 Development Guide

[Configuration]

```
struct scm_audio_cfg {
    uint32_t mclk_freq;           /* MCLK frequency */
    enum scm_i2s_wl word_length;  /* word length in bits */
    enum scm_i2s_fmt format;      /* I2S data format */
    enum scm_i2s_role role;       /* Role on I2S bus */
    uint32_t fs;                  /* Sampling frequency */
};
```

- mclk_freq: Frequency of the MCLK signal in Hz
- word_length: Number of bits per a single audio word, i.e., channel
  - SCM_I2S_WL_16: 16 bits
  - SCM_I2S_WL_20: 20 bits
  - SCM_I2S_WL_24: 24 bits
- format: Select io format
  - SCM_I2S_FMT_I2S
  - SCM_I2S_FMT_LJ

  - SCM_I2S_FMT_RJ
- role: Select a role on the bus
  - SCM_I2S_ROLE_MASTER : Generates I2S bit clock and word clock.
  - SCM_I2S_ROLE_SLAVE
- fs: Audio sampling frequency in Hz

Usage Steps:

1: Initialize the Audio CODEC.

2: Configure the Audio CODEC.

3: Start Audio CODEC's Input and output interface.

```
void start_audio_input(void)
{
    struct scm_audio_cfg cfg;

    scm_audio_init();

    memset(&cfg, 0, sizeof(cfg));
    cfg.mclk_freq = 12000000;
    cfg.word_length = SCM_I2S_WL_16;
    cfg.format = SCM_I2S_FMT_I2S;
    cfg.role = SCM_I2S_ROLE_SLAVE;
    cfg.fs = 44100;

    scm_audio_configure(&cfg);

    scm_audio_start(SCM_AUDIO_INPUT);
    scm_audio_start(SCM_AUDIO_OUTPUT);
}
```