SCM1612
Wi-Fi 6 和 BLE 5 低功耗 SoC

设备驱动开发指南

文档版本 1.3
发布日期 2025-3-3

联系方式
速通半导体科技有限公司 (www.senscomm.com)
江苏省苏州市工业园区苏州大道西 2 号国际大厦 303 室
销售或技术支持，请发送电子邮件至
support@senscomm.com

免责声明和注意事项

本文档仅按"现状"提供。速通半导体有限公司保留在无需另行通知的情况下对其或本文档中包含的任何规格进行更正、改进和其他变更的权利。

与使用本文档中的信息有关的一切责任，包括侵犯任何专有权利的责任，均不予承认。此处不授予任何明示或暗示、通过禁止或其他方式对任何知识产权的许可。

本文档中的所有第三方信息均按"现状"提供，不对其真实性和准确性提供任何保证。

本文档中提及的所有商标、商号和注册商标均为其各自所有者的财产，特此确认。

版本历史

版本	日期	描述
0.1	2023-11-23	初稿
1.0	2023-12-22	UART, ADC
1.1	2024-10-21	新增 I2S
1.2	2024-11-27	更新 SPI
1.3	2025-3-3	新增音频

目录

1 引言

本文档为开发者提供了一份全面的指南，主要介绍如何通过 SCM1612 设备驱动来实现 SoC（System on Chip，片上系统）外设的应用。文档的目的是帮助开发者充分利用 SCM1612 Wi-Fi 6 和 BLE 5 低功耗 SoC 的功能，重点指导如何有效使用设备驱动的应用程序接口（API），并将它们集成到对性能要求极高的应用中。

1.1 设备驱动 API

SCM1612 SDK 提供了两种不同层级的 API：

- 低层 API
  - 代码位置: hal/drivers/xxxx/xxxx.c
  - 头文件位置: include/hal/xxxx.h
  - 功能目的：这些 API 主要提供硬件抽象功能，尤其是直接访问硬件寄存器。
- 应用层 API（SCM API）
  - 代码位置: api/scm_xxxx.c
  - 头文件位置: api/include/scm_xxxx.h
  - 功能特点: SCM API 旨在通过通用功能函数简化对功能的访问，这些函数封装了对一个或多个低层 API、操作系统功能及基本错误处理的调用。

1.2 对性能要求极高的应用

尽管 SCM API 通过在低层 API 之上添加一个抽象层来简化开发过程，但这可能会带来一些性能上的开销。在资源密集型或对时间极为敏感的应用中，推荐直接使用低层 API。举个例子，在需要通过 SPI 和 I2C 接口与外部集成电路（IC）同时进行通信的自定义电路板中，建议开发一个专门针对 SPI 和 I2C 低层 API 的定制设备驱动程序。应用程序可以直接与这个定制驱动程序进行交互，而不是使用更高层次的 SCM API。

1.3 驱动程序事件通知回调

在 SCM1612 SDK 中，有些外设完成用户发起的操作可能需要一些时间。为此，SDK 提供了两种类型的 API：

同步 API (scm_xxxx)：这类 API 包含一个超时参数，适合于那些应用程序能够在指定时间内等待操作完成的场景。
异步 API (scm_xxxx_async)：这类 API 不包含超时参数，更适合那些不宜等待操作完成的场景。这些操作的完成情况，会通过特定的通知回调机制来告知。
在进行异步操作时，操作完成的通知是通过回调函数来传递的。这些回调函数在中断处理过程中被触发，并且应当迅速执行，以免影响到中断处理的效率。值得注意的是，由于这些回调函数运行的特定环境，其中使用其它 API 可能会受到限制。
通常情况下，出于简单易用和易于集成的考虑，大多数情况下推荐使用同步 API。

2 引脚控制

SoC（片上系统）的许多外设，例如 SPI、I2C、GPIO 和 PWM，都需要使用特定的引脚。

2.1 引脚复用选项

可用的引脚复用选项在以下文件中有详细列出：

- 文件位置：hal/drivers/pinctrl/pinctrl-scm2010.c

每个 SoC 引脚都可以被配置为若干预先定义好的功能之一。

以 UART0 和 UART1 为例，它们的引脚复用选项如下所示：

```
#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 引脚复用选择

引脚的选择依据具体的电路板设计而定，因为每种设计使用引脚的方式都不尽相同。比如，在 SCM2010 评估板中，UART 引脚的选择如下，文件位置为：

- 文件位置：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
```

一般来说，外设驱动程序，如 UART 驱动程序，会自动请求引脚驱动程序配置这些引脚复用。

3 UART

3.1 概述

SCM1612 SoC 支持使用通用异步接收发射器（UART）与外部设备通信。SoC 提供三个 UART 实例，其中一个可配置为控制台，以协助固件开发。

3.2 API 功能

- 初始化和配置：
  - scm_uart_init()：初始化 UART 模块。
  - scm_uart_deinit()：去初始化 UART 模块。
  - scm_uart_reset()：重置 UART 模块。
- 发送器和接收器：
  - 数据传输： scm_uart_tx()，scm_uart_tx_async()
  - 数据接收： scm_uart_rx()，scm_uart_rx_async()

3.3 开发指南

[配置结构]

```
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: 设置波特率（从 50 到 115200）。
- data_bits: 设置数据位长度（5、6、7 或 8 位）。
- parity: 配置奇偶校验（选项：无校验、奇校验、偶校验）。
- stop bits: 设置停止位长度（1 位或 2 位）。
- dma_en: 启用或禁用 DMA 传输。

[事件处理]

- SCM_UART_EVENT_TX_CMPL: 当使用异步 API 完成 UART 传输时触发。
- SCM_UART_EVENT_RX_CMPL: 当使用异步 API 完成 UART 接收时触发。

传输 (Tx) 和接收 (Rx) 的操作步骤：

1. 根据需要的配置初始化 UART。
2. 向连接的设备发送或从其接收数据。
3. 不再使用时反初始化 UART。

```
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 注意事项

4 SPI (串行外设接口)

4.1 概述

SCM1612 的 SPI 支持主模式和从模式，可以与外部设备进行通信。它可以配置为标准 SPI 或四线 SPI（QSPI），用于与兼容四线 SPI 的设备接口。

SPI 主模式支持命令、地址和数据阶段，且数据可以在无命令和地址阶段的情况下传输。
要通过 SPI 从设接收或发送数据，对等的主设备必须在发送或接收实际数据之前，通过单一 IO 发送 8 位命令和 8 位虚拟数据。

4.2 API 函数

- 初始化和配置:
  - `scm_spi_init()`: 初始化 SPI 模块。
  - `scm_spi_deinit()`: 去初始化 SPI 模块。
  - `scm_spi_configure()`: 设置 SPI 参数和选项。
  - `scm_spi_reset()`: 重置 SPI 模块。
- 主模式操作:
  - 数据传输: `scm_spi_master_tx()`, `scm_spi_master_tx_async()`
  - 数据接收: `scm_spi_master_rx()`,`scm_spi_master_rx_async()`
  - 综合传输和接收:`scm_spi_master_tx_rx()`,`scm_spi_master_tx_rx_async()`
  - 基于命令的操作:`scm_spi_master_tx_with_cmd()`,`scm_spi_master_rx_with_cmd()`,etc.
- 从模式操作:
  - 缓冲区管理: `scm_spi_slave_set_tx_buf()`, `scm_spi_slave_set_rx_buf()`, `scm_spi_slave_set_tx_rx_buf()`
  - 用户状态管理: `scm_spi_slave_set_user_state()`

4.3 开发指南

[配置]

```
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: 选择主模式或从模式
- mode: CPOL 和 CPHA 模式
  - SCM_SPI_MODE_0: active high, 奇数边沿采样
  - SCM_SPI_MODE_1: active high, 偶数边沿采样
  - SCM_SPI_MODE_2: active low, 奇数边沿采样
  - SCM_SPI_MODE_3: active low, 偶数边沿采样
- data_io_format: 选择 IO 格式
  - SPI_DATA_IO_FORMAT_SIGNLE
  - SPI_DATA_IO_FORMAT_DUAL
  - SPI_DATA_IO_FORMAT_QUAD

  注意: scm_spi_master_tx_rx 和 scm_spi_master_tx_rx_sync 不能在 DUAL 或 QUAD IO 格式下使用。
- bit_order: 选择位顺序
  - 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: 选择 SPI IO 时钟源
  - SPI_CLK_SRC_XTAL: XTAL 40Mhz
  - SPI_CLK_SRC_PLL: PLL 时钟。SPI0 和 SPI1 使用 240Mhz PLL 时钟，SPI2 使用 80Mhz PLL 时钟
- clk_div_2mul: 源时钟分频值。将设置值乘以 2 后除以。
  例如) clk_src = SPI_CLK_SRC_XTAL and clk_div_2mul = 20,
  SPI IO 时钟为 1Mhz。
- dma_en: 启用或禁用 DMA 传输

[事件]

- 主模式事件
  - SCM_SPI_EVENT_MASTER_TRANS_CMPL : SPI 传输完成事件。此事件为异步 API 触发
- 从模式事件
  - SCM_SPI_EVENT_SLAVE_TX_REQUEST : 接收到读取命令时发生。
  - SCM_SPI_EVENT_SLAVE_RX_REQUEST : 接收到读取命令时发生。
  - SCM_SPI_EVENT_SLAVE_USER_CMD: 接收到未定义命令时发生。
  - SCM_SPI_EVENT_SLAVE_TRAN_CMPL: SPI 传输完成事件。

主模式使用步骤：
步骤 1: 初始化 SPI

步骤 2: 将 SPI 配置为主模式

步骤 3: 向从设备发送或从从设备接收数据

步骤 4: 在不再使用时去初始化 SPI

```
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);
}
```

从模式使用步骤：

步骤 1: 初始化 SPI。

步骤 2: 将 SPI 配置为从模式。

步骤 3: 管理 Tx/Rx 缓冲区并响应事件。

步骤 4: 使用后去初始化 SPI。

```
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 注意事项

- SPI 实例:

SCM1612 提供三个 SPI 实例。SPI0 专用于闪存，不适用于其他用途。

- DMA 使用:

如果使用 DMA，请确保传输和接收缓冲区位于允许 DMA 的区域内。

- 从模式考虑:

从模式中的通知功能应该快速执行，以避免由于主时钟延迟导致的事务失败。

- 支持多个从机模式（作为从机）:

SPI 驱动应支持多个从机的连接模式，例如，通过为 **atcspi** 驱动启用 CONFIG_SPI_SUPPORT_MULTI_SLAVES_AS_A_SLAVE 配置项实现此功能。

  - 主机必须在发送命令和数据之间插入足够的**虚拟时钟周期**（Dummy Cycles），以确保从机有足够的时间响应。
  - 从机需要通过 **slave_extra_dummy_cycle** 参数配置虚拟时钟周期的数量，以匹配主机的设置。
  - 虚拟时钟周期的数量与 SPI 总线的时钟频率直接相关，时钟越快，需要的虚拟时钟周期越多。

- 支持多个从机模式（作为主机）

SPI 驱动应支持多个从机连接模式，例如，通过为 **atcspi** 驱动启用 CONFIG_SPI_SUPPORT_MULTI_SLAVES_AS_A_MASTER 配置项实现此功能。

  - 主机需要通过 **master_cs_bitmap** 参数提供用于选择从机的 GPIO 位图。
  - 位图中每一位对应一个 GPIO 引脚，用于控制某一从机的片选信号。例如：

- 如果需要使用 GPIO0 和 GPIO15 分别控制从机 1 和从机 2，则应将 master_cs_bitmap 配置为 **0x00008001**。

5 I2C (互联集成电路)

5.1 功能描述

SCM1612 SoC 支持 I2C 的主模式和从模式，便于与各种外部设备进行通信。I2C 接口旨在处理标准数据传输协议，确保与广泛的 I2C 兼容设备相容。

5.2 API 函数

- 初始化和配置:
  - `scm_i2c_init()`: 初始化 I2C 模块。
  - `scm_i2c_deinit()`: 去初始化 I2C 模块。
  - `scm_i2c_configure()`: 配置 I2C 参数和选项。
  - `scm_i2c_reset()`: 重置 I2C 模块。

- 主模式操作:
  - 数据传输: `scm_i2c_master_tx()`, `scm_i2c_master_tx_async()`
  - 数据接收: `scm_i2c_master_rx()`, `scm_i2c_master_rx_async()`
  - 综合传输和接收: `scm_i2c_master_tx_rx()`, `scm_i2c_master_tx_rx_async()`
  - 设备探测: `scm_i2c_master_probe()`

- 从模式操作:
  - 发送请求: `scm_i2c_slave_tx()`
  - 接收请求: `scm_i2c_slave_rx()`

5.3 开发指南

[事件]

- 主模式事件
  - SCM_I2C_EVENT_MASTER_TRANS_CMPL: 当主机完成传输时
- 从模式事件

- SCM_I2C_EVENT_SLAVE_RX_REQUEST: 当从机接收到 RX（主机到从机）数据请求时
- SCM_I2C_EVENT_SLAVE_TX_REQUEST: 当从机接收到 TX（从机到主机）数据请求时
- SCM_I2C_EVENT_SLAVE_RX_CMPL: 当从机完成接收时
- SCM_I2C_EVENT_SLAVE_TX_CMPL: 当从机完成传输时

主模式使用步骤：
步骤 1: 初始化 I2C。

步骤 2: 将 I2C 配置为主模式。

步骤 3: 执行发送/接收操作。

步骤 4: 使用后去初始化 I2C。

```
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);
}
```

从模式使用步骤：
步骤 1: 初始化 I2C

步骤 2: 将 I2C 配置为从模式。

步骤 3: 当 I2C 通知函数被调用时，准备 Tx 或 Rx 数据

步骤 4: 使用后去初始化 I2C

示例:

```
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 注意事项

对于从模式，如果用户提供的缓冲区不足，

- 当主机尝试读取超出缓冲区的内容时，将返回 0 给主机
- 当主机尝试写入超出缓冲区的内容时，数据将被静默丢弃。

6 计时器

6.1 概述

SCM1612 SoC 配备了强大的计时器功能，提供两个硬件计时器。每个计时器具有四个通道，可以配置成各种模式，这种灵活性允许创建四个独立的计时器，以满足不同的应用需求。

6.2 功能描述

- scm_timer_configure: 配置计时器模式
- scm_timer_start: 启动计时器
- scm_timer_stop: 停止计时器
- scm_timer_start_multi: 同时启动多个计时器
- scm_timer_stop_multi: 同时停止多个计时器
- scm_timer_value: 如果计时器配置为自由运行模式，读取计时器的值

6.3 开发指南

执行以下步骤:

步骤 1: 配置计时器

步骤 2: 启动计时器

步骤 3: 读取计时器

步骤 4: 停止计时器

示例:

```
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 注意事项

TIMER1 实例被电源管理和 BLE 子系统使用。当启用电源管理（PM）或 BLE 时，应用程序不应使用索引 SCM_TIMER_IDX_1 调用计时器 API。

7 GPIO (通用输入/输出)

7.1 概述

SCM1612 SoC 配备了 25 个通用输入/输出（GPIO）引脚。这些 GPIO 可以配置为输入或输出，为与外部硬件和传感器的接口提供多功能性。

7.2 API 函数

- 配置和控制:
  - `scm_gpio_configure()`: 配置 GPIO 引脚为输入或输出，包括上拉或下拉电阻的选项。
  - `scm_gpio_write()`: 设置 GPIO 引脚的输出电平。
  - `scm_gpio_read()`: 读取 GPIO 引脚的输入电平。
  - `scm_gpio_enable_interrupt()`: 启用 GPIO 中断，支持事件驱动编程。
  - `scm_gpio_disable_interrupt()`: 禁用 GPIO 中断。

7.3 开发指南

GPIO 使用的基本步骤：
步骤 1: 为预期的输入/输出功能配置 GPIO 引脚。

步骤 2: 对于输出使用，向 GPIO 引脚写入期望的电平。

步骤 3: 对于输入使用，根据需要从 GPIO 引脚读取电平。

步骤 4: 如果使用中断，启用它们并提供一个回调函数来处理事件。

示例代码:

```
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 注意事项

- 电源域:

GPIO 引脚 0 到 7 位于“常开”电源域，这意味着它们可以在低功耗模式下保持状态并接收唤醒事件。

- 中断处理：

应注意确保中断回调函数高效，并且不会阻塞关键任务。

8 eFuse

8.1 功能描述

SCM1612 的 eFuse（电子可编程保险丝）提供 1024 位的非易失性内存。这个功能通常用于存储系统关键数据，如安全引导密钥、设备身份和 RF 校准数据。

8.2 API 函数

- eFuse 操作:
  - `scm_efuse_read()`: 从指定的 eFuse 地址读取数据。
  - `scm_efuse_write()`: 向指定的 eFuse 地址写入数据。

8.3 开发指南

保留字段及其大小如下所示。

```
#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
```

示例:

```
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 注意事项

由于 eFuse 只能写入一次，因此在写入新值时需要格外小心。

eFuse 的每一位只能从 0 变成 1，不能反向变化。在特殊情况下，同一字段可以多次写入以将特定位设置为 1。

9 ADC

9.1 功能说明

SCM1612 SoC 配备了一个模拟数字转换器 (ADC)，它能够读取模拟电压水平并将其转换成数字格式。下表展示了物理 GPIO 引脚与对应 ADC 通道之间的对应关系。

ADC 通道	GPIO 引脚
4	4
5	7
6	0
7	1

9.2 API 功能

- ADC 测量
  - scm_adc_read(), scm_adc_read_aync()：从指定 ADC 通道读取数据。
  - scm_adc_reset()：重置 ADC。

9.3 开发指南

示例代码：

```
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);
}
```

这个示例展示了如何执行同步和异步 ADC 读取。scm_adc_read 函数用于从通道 0 进行同步读取，而 scm_adc_read_async 用于从通道 1 进行异步读取，配合回调函数使用。

9.4 注意事项

10 I2S

10.1 概述

SCM1612 的 I2S 支持主从模式，可以与外部音频 CODEC 交换音频样本数据。它支持多种格式和字长，具体描述如下。

10.2 API 函数

- 初始化和配置：
  - `scm_i2s_init()`: 初始化 I2S 模块。
  - `scm_i2s_deinit()`: 关闭 I2S 模块。
  - `scm_i2s_configure()`: 设置 I2S 参数和选项。
  - `scm_i2s_start()`: 启动指定方向的 I2S 数据流。
  - `scm_i2s_stop()`: 停止指定方向的 I2S 数据流。
  - `scm_i2s_read_block()`: 从 I2S 输入流中读取一个音频样本块，返回成功或错误代码。
  - `scm_i2s_write_block()`: 向 I2S 输出流写入一个音频样本块，返回成功或错误代码。
  - `scm_i2s_get_block_buffer_size()`: 返回配置的块缓冲区大小。

10.3 开发指南

[配置]

```
struct scm_i2s_cfg {
    enum scm_i2s_wl word_length; // 每个音频字（通道）的位数
    enum scm_i2s_fmt format;     // I/O 格式
    enum scm_i2s_role role;      // 总线角色（主/从）
    enum scm_i2s_direction dir;  // 数据传输方向（Tx/Rx）
    uint32_t fs;                 // 音频采样频率（Hz）
    int duration_per_block;      // 每个缓冲区块的时间持续（毫秒）
    int number_of_blocks;        // 要分配的最大缓冲区块数
    int timeout;                 // 读/写超时（毫秒）
}
```

- word_length:
  - SCM_I2S_WL_16: 16 位

- SCM_I2S_WL_20: 20 位
- SCM_I2S_WL_24: 24 位

注意：20 位或 24 位的字长在内存中将占用 32 位。

- format: 选择 I/O 格式
  - SCM_I2S_FMT_I2S
  - SCM_I2S_FMT_LJ
  - SCM_I2S_FMT_RJ
- role: 选择总线角色
  - SCM_I2S_ROLE_MASTER : 生成 I2S 位时钟和字时钟。
  - SCM_I2S_ROLE_SLAVE
- dir: 指定数据传输方向
  - SCM_I2S_RX
  - SCM_I2S_TX
- fs: 音频采样频率（Hz）
- duration_per_block: 每个缓冲区块对应的时间持续（毫秒）

注意：必须小于 1000（1 秒）。

- number_of_blocks: 要分配的最大缓冲区块数
- timeout: 读/写超时（毫秒）

传输使用步骤：

1: 初始化 I2S。
2: 配置 I2S 以支持 Tx 和 Rx 方向。
3: 在 Tx 方向启动 I2S 流。
4: 向 I2S Tx 流写入音频样本块。
5: 停止 Tx 流并在不再需要时停止 I2S 模块。

```
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();
}
```

接收的使用步骤：

1: 初始化 I2S。
2: 配置 I2S 以支持 Tx 和 Rx 方向。
3: 在 Rx 方向启动 I2S 流。
4: 从 I2S Rx 流中读取音频样本块。
5: 停止 Rx 流并在不再需要时停止 I2S 模块。

```
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 注意事项

- **内存使用：**

SCM1612 的 I2S 流处理可用的内存有限。请根据需要调整 cfg.duration_per_block 和 cfg.number_of_blocks，以防止内存溢出。

- **配置：**

无论实际数据传输方向如何，必须配置 I2S 的 Tx 和 Rx 方向。确保两个方向的参数相同。

11 音频

11.1 概述

本 CODEC API 用于控制外部音频 CODEC，通常与 I2S API 配合使用。

11.2 API 函数

- 初始化和配置:
  - `scm_audio_init()`: 初始化音频 CODEC。
  - `scm_audio_deinit()`: Deinitializes the Audio CODEC.
  - `scm_audio_configure()`: 设置音频 CODEC 参数。
  - `scm_audio_start()`: 启动指定的音频 CODEC 接口。
  - `scm_audio_stop()`: 停止指定的音频 CODEC 接口。
  - `scm_audio_get_volume()`: 获取当前音量。
  - `scm_audio_set_volume()`: 设置当前音量。
  - `scm_audio_mute()`: 对指定的音频 CODEC 接口执行静音操作。
  - `scm_audio_unmute()`: 取消对指定音频 CODEC 接口的静音操作。

11.3 开发指南

[配置]

```
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: MCLK 信号的频率（单位 Hz）
- word_length: 单个音频字（通道）所占位数

- SCM_I2S_WL_16: 16 位
- SCM_I2S_WL_20: 20 位
- SCM_I2S_WL_24: 24 位

- format: 选择 I/O 格式
  - SCM_I2S_FMT_I2S
  - SCM_I2S_FMT_LJ
  - SCM_I2S_FMT_RJ
- role: 选择在总线上的角色
  - SCM_I2S_ROLE_MASTER：生成 I2S 位时钟和字时钟
  - SCM_I2S_ROLE_SLAVE
- fs: 音频采样频率（单位 Hz）

使用步骤：

1: 初始化音频 CODEC.

2: 配置音频 CODEC.

3: 启动音频 CODEC 的输入和输出接口。

```
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);
}
```