

WS63V100 设备驱动

开发指南

文档版本 02

发布日期 2024-10-30

前言

概述

本文档主要介绍 WS63 设备驱动开发的相关内容主要包括工作原理、按场景描述接口使用方法和注意事项。

产品版本

产品名称	产品版本
WS63	V100


读者对象




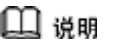
本文档主要适用于以下工程师：

- 技术支持工程师
- 软件开发工程师

符号约定

在本文中可能出现下列标志，它们所代表的含义如下。

符号	说明
 危险	表示如不可避免则将会导致死亡或严重伤害的具有高等级风险的危害。

符号	说明
 警告	表示如不避免则可能导致死亡或严重伤害的具有中等级风险的危害。
 注意	表示如不避免则可能导致轻微或中度伤害的具有低等级风险的危害。
 须知	用于传递设备或环境安全警示信息。如不避免则可能会导致设备损坏、数据丢失、设备性能降低或其它不可预知的结果。 “须知”不涉及人身伤害。
 说明	对正文中重点信息的补充说明。 “说明”不是安全警示信息，不涉及人身、设备及环境伤害信息。

修改记录

文档版本	发布日期	修改说明
02	2024-10-30	更新 “3.4 注意事项” 小节内容。
01	2024-04-10	第一次正式版本发布。
00B02	2024-03-29	<ul style="list-style-type: none">更新 “1.1 概述” 小节内容。更新 “1.3 开发指引” 小节内容。更新 “4.1 概述” 小节内容。更新 “5.2 功能描述” 小节内容。更新 “5.3 开发指引” 小节内容。更新 “9.1 概述” 小节内容。更新 “10.4 注意事项” 小节内容。更新 “12.1 概述” 小节内容。
00B01	2024-03-15	第一次临时版本发布。

目 录

前言i

1 Pinctrl1

1.1 概述1

1.2 功能描述1

1.3 开发指引2

1.4 注意事项2

2 GPIO3

2.1 概述3

2.2 功能描述3

2.3 开发指引4

2.4 注意事项5

3 UART6

3.1 概述6

3.2 功能描述7

3.3 开发指引7

3.4 注意事项11

4 SPI.....13

4.1 概述13

4.2 功能描述14

4.3 开发指引14

4.4 注意事项17

5 I2C.....18

5.1 概述18

5.2 功能描述18

5.3 开发指引	19
5.4 注意事项	20
6 ADC.....	21
6.1 概述	21
6.2 功能描述	21
6.3 开发指引	22
6.4 注意事项	23
7 DMA.....	24
7.1 概述	24
7.2 功能描述	25
7.3 开发指引	25
7.4 注意事项	27
8 PWM.....	28
8.1 概述	28
8.2 功能描述	28
8.3 开发指引	29
8.4 注意事项	30
9 WDT.....	32
9.1 概述	32
9.2 功能描述	32
9.3 开发指引	33
9.4 注意事项	34
10 Timer	35
10.1 概述	35
10.2 功能描述	35
10.3 开发指引	36
10.4 注意事项	37
11 SysTick.....	38
11.1 概述	38
11.2 功能描述	38
11.3 开发指引	39

11.4 注意事项.....40

12 TCXO41

12.1 概述41

12.2 功能描述41

12.3 开发指引42

12.4 注意事项42

13 SFC43

13.1 概述43

13.2 功能描述43

13.3 开发指引44

13.4 注意事项45

14 EFUSE46

14.1 概述46

14.2 功能描述46

14.3 开发指引47

14.4 注意事项47

1 Pinctrl

- 1.1 概述
- 1.2 功能描述
- 1.3 开发指引
- 1.4 注意事项

1.1 概述

Pinctrl 控制器用于控制 IO 管脚的复用功能，可配置规格如下：

- 支持配置 GPIO_00-GPIO_18 一组 IO 管脚。
- 支持配置 IO 驱动能力、IO 功能复用以及设置 IO 上下拉状态等功能。

1.2 功能描述

Pinctrl 驱动模块提供的接口及功能如下：

- uapi_pin_init：初始化 Pinctrl。
- uapi_pin_deinit：去初始化 Pinctrl。
- uapi_pin_set_mode：设置指定 IO 复用模式。
- uapi_pin_get_mode：获取指定 IO 的复用模式。
- uapi_pin_set_ds：设置指定 IO 驱动能力。
- uapi_pin_get_ds：获取指定 IO 驱动能力。
- uapi_pin_set_pull：设置指定 IO 的上拉/下拉状态。

- `uapi_pin_get_pull`: 获取指定 IO 的上拉/下拉状态。

1.3 开发指引

Pinctrl 接口使用遵循如下操作步骤（以下步骤根据实际需要可选）：

步骤 1 调用 `uapi_pin_set_mode`、`uapi_pin_get_mode` 接口，设置/查看指定 IO 的复用模式。

步骤 2 调用 `uapi_pin_set_ds`、`uapi_pin_get_ds` 接口，设置/查看指定 IO 的驱动能力。

步骤 3 调用 `uapi_pin_set_pull`、`uapi_pin_get_pull` 接口，设置/查看指定 IO 的上拉/下拉状态。

----结束

示例：

```
/* 设置GPIO_00的复用功能为gpio */
uapi_pin_set_mode(GPIO_00, PIN_MODE_0);
/* 设置GPIO_00的驱动能力为PIN_DS_2 */
uapi_pin_set_ds(GPIO_00, PIN_DS_2);
/* 设置GPIO_00为上拉模式 */
uapi_pin_set_pull(GPIO_00, PIN_PULL_TYPE_UP);
```

1.4 注意事项

配置 IO 复用功能时，应关注此 IO 是否支持目标功能或者已经被复用为其他功能，避免影响既有功能，IO 复用请参考

“`sdk\drivers\chips\ws63\include\platform_core_rom.h`” 源码中 “`pin_t`” 结构体的定义。

2 GPIO

- [2.1 概述](#)
- [2.2 功能描述](#)
- [2.3 开发指引](#)
- [2.4 注意事项](#)

2.1 概述

GPIO (General-purpose input/output) 是通用输入输出的缩写，是一种通用的 I/O 接口标准。可以配置为输入或输出模式，以便控制外部设备或与其他设备通信。可用于连接各种设备，如 LED 灯、传感器、执行器等。

GPIO 规格如下：

- 支持设置 GPIO 管脚方向、设置输出电平状态。
- 支持外部电平中断以及外部边沿中断上报。
- 支持每个 GPIO 独立中断。

2.2 功能描述

GPIO 模块提供的接口及功能如下：

- `uapi_gpio_init`：初始化 GPIO。
- `uapi_gpio_deinit`：去初始化 GPIO。
- `uapi_gpio_set_dir`：设置指定 GPIO 方向（输入/输出）。

- uapi_gpio_set_val: 设置指定 GPIO 电平状态。
- uapi_gpio_get_val: 获取指定 GPIO 电平状态。
- uapi_gpio_register_isr_func: 注册指定 GPIO 中断。
- uapi_gpio_unregister_isr_func: 去注册指定 GPIO 中断。
- uapi_gpio_disable_interrupt: 关闭 GPIO 中断。
- uapi_gpio_enable_interrupt: 使能 GPIO 中断。
- uapi_gpio_clear_interrupt: 清除 GPIO 中断。
- uapi_gpio_toggle: GPIO 输出电平状态翻转。

2.3 开发指引

GPIO 接口使用遵循如下操作步骤:

步骤 1 调用 uapi_pin_set_mode 接口, 将 PIN 复用为 GPIO 功能。

步骤 2 根据用户开发需求, 可设置 GPIO 接口为输出、输入和中断模式, 设置方法如下:

- 输出模式:
 - a. 调用 uapi_gpio_set_dir 接口, 设置 GPIO 方向为 OUT。
 - b. 调用 uapi_gpio_set_val 接口, 设置 GPIO 输出电平状态 (高/低) 。
- 输入模式:
 - a. 调用 uapi_gpio_set_dir 接口, 设置 GPIO 方向为 IN。
 - b. 调用 uapi_gpio_get_val 接口, 获取 GPIO 输入电平状态。
- 中断模式:
 - a. 调用 uapi_gpio_set_dir 接口, 设置 GPIO 方向为 IN。
 - b. 调用 uapi_gpio_register_isr_func 接口, 注册 GPIO 中断回调函数。
 - c. 调用 uapi_gpio_unregister_isr_func 接口, 注销 GPIO 中断回调函数 (去注册中断时调用) 。

----结束

示例:

```
#include "gpio.h"
void gpio_callback_func(pin_t pin, uintptr_t param)
{
```

```
        unused(param);
        osal_printk("PIN:%d interrupt success. \r\n", pin);
    }
errcode_t sample_gpio_test(pin_t pin)
{
    uapi_pin_init();
    uapi_gpio_init();

    uapi_pin_set_mode(pin, HAL_PIO_FUNC_GPIO); /* 设置指定IO复用为GPIO模式 */

    uapi_gpio_set_dir(pin, GPIO_DIRECTION_INPUT); /* 设置指定GPIO为输入模式 */

    /* 注册指定GPIO上升沿中断，回调函数为gpio_callback_func */

    if (uapi_gpio_register_isr_func(pin, GPIO_INTERRUPT_RISING_EDGE,
gpio_callback_func) != ERRCODE_SUCC) {
        uapi_gpio_unregister_isr_func(pin); /* 清理残留 */
        return ERRCODE_FAIL;
    }
    return ERRCODE_SUCC;
}
```

2.4 注意事项

- 在使用 GPIO 电平中断时，需要在回调函数控制输入电平触发中断时间，否则会导致系统一直处于中断处理中，无法执行其他功能。
- 触发方式（trigger）在没有明确需求场景时，推荐使用默认配置。

3 UART

- [3.1 概述](#)
- [3.2 功能描述](#)
- [3.3 开发指引](#)
- [3.4 注意事项](#)

3.1 概述

UART (Universal Asynchronous Receiver/Transmitter) 是通用异步收发器的缩写，是一种串行、异步、全双工的通信协议，用于设备间的数据传输。UART 是最常用的设备间通信协议之一，正确配置后，UART 可以配合许多不同类型的涉及发送和接收串行数据的串行协议工作。

WS63 芯片 MCU 侧提供了 3 个可配置的 UART 外设单元：UART0、UART1、UART2，UART 规格如下：

- 支持可编程数据位(5-8bit)、可编程停止位(1-2bit)、可编程校验位(奇/偶校验，无校验)。
- UART 支持无流控，RTS/CTS 流控模式
- 提供 64×8 的 TX，64×8 的 RX FIFO
- 支持接收 FIFO 中断、发送 FIFO 中断、接收超时中断、错误中断等中断屏蔽与响应。
- 支持 DMA 数据搬移方式。

3.2 功能描述

说明

若 UART 驱动需要支持 DMA 数据收发，需确保 DMA 驱动已完成初始化。

驱动代码在 include\driver\uart.h 声明了 UART 驱动相关函数，提供的接口及功能如下如下：

- uapi_uart_init：初始化 UART。
- uapi_uart_deinit：去初始化 UART。
- uapi_uart_read：读数据。
- uapi_uart_write：写数据。
- uapi_uart_set_flow_ctrl：配置 UART 硬流控。
- uapi_uart_set_software_flow_ctrl_level：配置软件流控的等级。
- uapi_uart_get_attr：获取 UART 配置参数。
- uapi_uart_set_attr：设置 UART 配置参数。
- uapi_uart_has_pending_transmissions：查询 UART 是否正在传输数据。
- uapi_uart_register_rx_callback：注册接收回调函数，此回调函数会根据触发条件和 Size 触发。
- uapi_uart_unregister_rx_callback：取消注册接收回调函数。
- uapi_uart_register_parity_error_callback：注册奇偶校验错误处理的回调函数。
- uapi_uart_register_frame_error_callback：注册帧错误处理回调函数。
- uapi_uart_write_int：使用中断模式将数据发送到已打开的 UART 上，当数据发送完成，会调用回调函数。
- uapi_uart_write_by_dma：通过 DMA 发送数据。
- uapi_uart_flush_rx_data：刷新 UART 接收 Buffer 中的数据。
- uapi_uart_get_rx_data_count：获取当前接收 Buffer 中的数据。
- uapi_uart_rx_fifo_is_empty：判断 RX FIFO 是否为空。

3.3 开发指引

以应用 UART0 为例，数据收发流程如下：

步骤 1 配置 IO 复用。将对应的 IO 分别复用为 UART1 的 TX、RX、RTS、CTS 功能。

如果不需要支持硬件流控，仅配置 TX、RX 即可。

```
void usr_uart_io_config(void)
{
    /* 如下IO复用配置，也可集中在SDK中的usr_io_init函数中进行配置 */
    uapi_pin_set_mode(S_AGPIO5, HAL_PIO_FUNC_UART_H0_M1); /* uart1 rtx */
    uapi_pin_set_mode(S_AGPIO6, HAL_PIO_FUNC_UART_H0_M1); /* uart1 ctx */
    uapi_pin_set_mode(S_AGPIO12, HAL_PIO_FUNC_UART_H0_M1); /* uart1 tx */
    uapi_pin_set_mode(S_AGPIO13, HAL_PIO_FUNC_UART_H0_M1); /* uart1 rx */
}
```

步骤 2 UART 初始化。配置 UART 的波特率、数据位等属性，并使能 UART。

```
#define TEST_UART_RX_BUFF_SIZE 0x1 /* 定义 UART 接收缓存区大小 */
unsigned char g_uart_rx_buff[TEST_UART_RX_BUFF_SIZE] = { 0 };
uart_buffer_config_t g_uart_buffer_config = {
    .rx_buffer = g_uart_rx_buff,
    .rx_buffer_size = TEST_UART_RX_BUFF_SIZE
};
errcode_t usr_uart_init_config(void)
{
    errcode_t errcode;
    uart_attr_t attr = {
        .baud_rate = 115200, /* 波特率 */
        .data_bits = 8,      /* 数据位 */
        .stop_bits = 1,      /* 停止位 */
        .parity = 0          /* 校验位 */
    };
    uart_pin_config_t pin_config = {
        .tx_pin = S_AGPIO5, /* uart1 tx */
        .rx_pin = S_AGPIO6, /* uart1 rx */
        .cts_pin = S_AGPIO12, /* 流控功能，可选 */
        .rts_pin = S_AGPIO13 /* 流控功能，可选 */
    };
    errcode = uapi_uart_init(UART_BUS_1, &pin_config, &attr, NULL, &g_uart_buffer_config);
    if (errcode != ERRCODE_SUCC) {
        osal_printk("uart init fail\r\n");
    }
    return errcode;
}
```

步骤 3 UART 数据收发。调用 UART 轮询读写数据接口，进行数据收发。

```
void usr_uart_read_data(void)
{
    int len;
    unsigned char g_test_uart_rx_buffer[64];
    len = uapi_uart_read(UART_BUS_0, g_test_uart_rx_buffer, 64, 0);
    if(len > 0) {
        /* process */
    }
}

int usr_uart_write_data(unsigned int size, char* buff)
{
    unsigned char tx_buff[10] = { 0 };
    if (memcpy_s(tx_buff, 10, buff, size) != EOK) {
        return ERRCODE_FAIL;
    }
    int ret = uapi_uart_write(UART_BUS_0, tx_buff, size, 0);
    if(ret == -1) {
        return ERRCODE_FAIL;
    }
    return ERRCODE_SUCC;
}
```

----结束

UART DMA 模式发送数据流程如下：

步骤 1 配置 IO 复用。将对应的 IO 复用为 UART 的 TX、RX、RTS、CTS 功能。

如果不需要支持硬件流控，仅配置 TX、RX 即可。

```
void usr_uart_io_config(void)
{
    /* 如下IO复用配置，也可集中在SDK中的usr_io_init函数中进行配置 */
    uapi_pin_set_mode(S_AGPIO5, HAL_PIO_FUNC_UART_H0_M1); /* uart1 rtx */
    uapi_pin_set_mode(S_AGPIO6, HAL_PIO_FUNC_UART_H0_M1); /* uart1 ctx */
    uapi_pin_set_mode(S_AGPIO12, HAL_PIO_FUNC_UART_H0_M1); /* uart1 tx */
    uapi_pin_set_mode(S_AGPIO13, HAL_PIO_FUNC_UART_H0_M1); /* uart1 rx */
}
```

步骤 2 UART 初始化。配置 UART 的波特率、数据位等属性，并使能 UART。

```
errcode_t usr_uart_init_config(void)
{

```

```

errcode_t errcode;
uart_attr_t attr = {
    .baud_rate = 115200, /* 波特率 */

    .data_bits = 8,      /* 数据位 */

    .stop_bits = 1,      /* 停止位 */

    .parity = 0          /* 校验位 */
};
uart_pin_config_t pin_config = {
    .tx_pin = S_AGPI05, /* uart1 tx */
    .rx_pin = S_AGPI06, /* uart1 rx */

    .cts_pin = S_AGPI012, /* 流控功能, 可选 */

    .rts_pin = S_AGPI013 /* 流控功能, 可选 */
};
uart_extra_attr_t ext_config = {
    .tx_dma_enable = true,
    .tx_int_threshold = 0x4,
}
errcode = uapi_uart_init(UART_BUS_1, &pin_config, &attr, &ext_config,
&g_uart_buffer_config);
if (errcode != ERRCODE_SUCC) {
    osal_printk("uart init fail\n");
}
return errcode;
}

```

步骤 3 UART DMA 数据发。

```

#define TEST_UART_DMA_SEND_BUFF_SIZE 1024
#define HAL_DMA_TRANSFER_WIDTH_8 0
#define HAL_DMA_BURST_TRANSACTION_LENGTH_4 1
static errcode_t test_uart_write_by_dma()
{
    uint8_t dma_buff[TEST_UART_DMA_SEND_BUFF_SIZE] = { 0 };
    if (memset_s(dma_buff, TEST_UART_DMA_SEND_BUFF_SIZE, 0xA5,
TEST_UART_DMA_SEND_BUFF_SIZE) != 0) {
        return ERRCODE_FAIL;
    }
    uart_write_dma_config_t dma_cfg = {
        .src_width = HAL_DMA_TRANSFER_WIDTH_8,          /* 0代表8bit */
        .dest_width = HAL_DMA_TRANSFER_WIDTH_8,         /* 0代表8bit */
    }
}

```

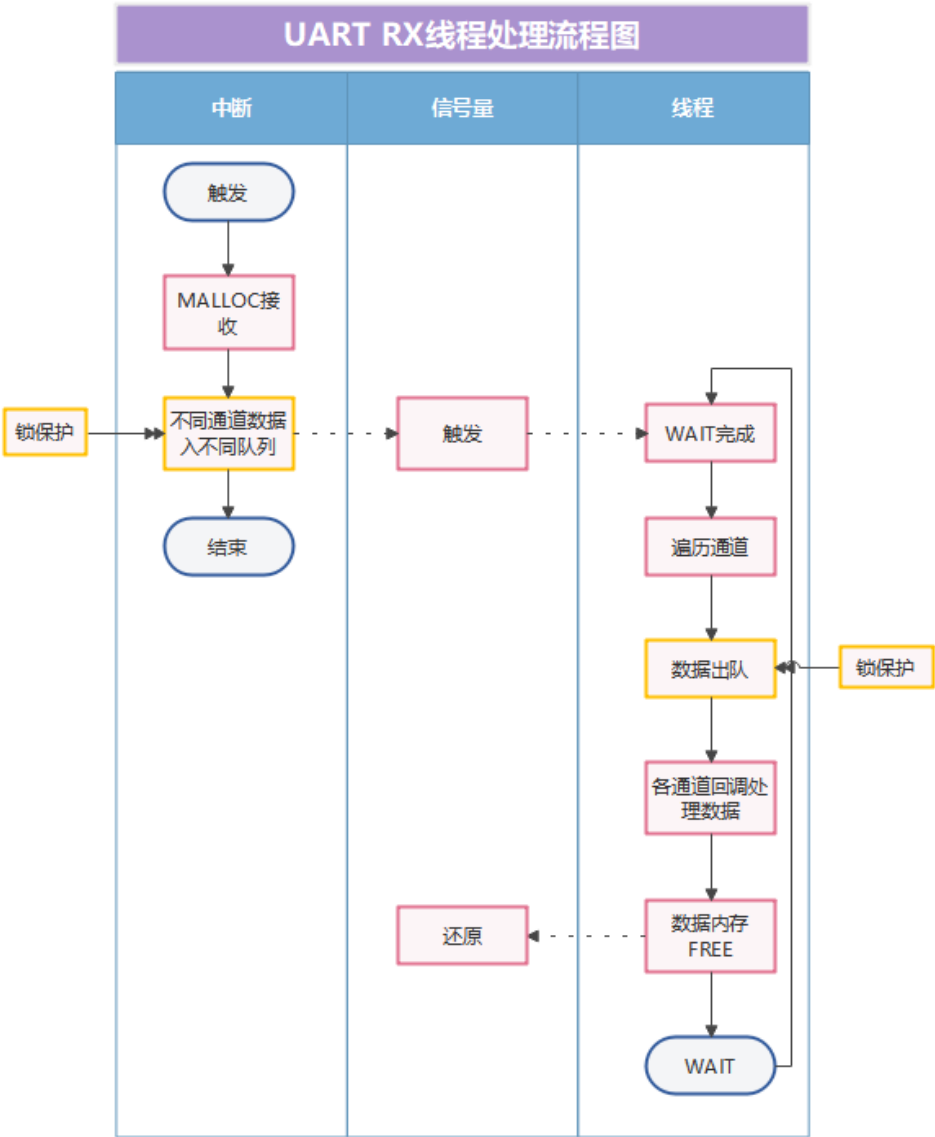


```
.burst_length = HAL_DMA_BURST_TRANSACTION_LENGTH_4, /* 代表4字节 */  
  
.priority = 0 /* 优先级0 */  
};  
if (uapi_uart_write_by_dma(UART_BUS_0, dma_buff, len, &dma_cfg) != len) {  
    osal_printk("[UART] *** memory t-o uart fail!\r\n");  
    return ERRCODE_FAIL;  
}  
return ERRCODE_SUCC;  
}
```

----结束

3.4 注意事项

- SDK 中，UART0 默认作为程序烧写和 DebugKites 工具维测数据通道。
- SDK 中，UART1 默认作为 Testsuite、AT 以及数据打印共享串口。
- SDK 中 drivers/chips/ws63/include/platform_core.h 文件定义了 UART 使用情况，TEST_SUITE_UART_BUS 定义了 testsuite 调试使用的串口，LOG_UART_BUS 定义了 HSO 工具使用的串口。
- SDK 中，UART 提供了特性宏 CONFIG_UART_SUPPORT_RX_THREAD。
 - 开启后 CONFIG_UART_SUPPORT_RX_THREAD，可配置串口 RX 数据通过线程处理，用于避免硬件中断直调串口回调函数导致耗时过长时，出现丢中断操作，最后导致 RX 数据丢包的现象；
 - 特性原理为：新建线程，线程被触发时对各串口数据队列进行处理，处理方式为调用各串口回调函数；中断中取消直调串口回调函数的操作，改为仅接收数据并存入队列并触发线程处理，缩短中断耗时，降低丢中断概率；



- 该特性开启方式为：

- 步骤 1 进入 menuconfig 配置，进入(Top) → Drivers → Drivers → UART → Uart Configuration。
- 步骤 2 在 UART support RX 下面打开 UART support RX thread。
- 步骤 3 配置线程相关参数，包括 uart rx 线程栈大小，rx 数据流控流水线大小以及线程优先级，完成后保存即可。

----结束

4 SPI

- [4.1 概述](#)
- [4.2 功能描述](#)
- [4.3 开发指引](#)
- [4.4 注意事项](#)

4.1 概述

SPI (Serial Peripheral Interface) 是一种高速、全双工、同步的通信总线。它可以使 MCU 与各种外围设备以串行方式进行通信以交换信息。SPI 总线可直接与各个厂家生产的多种标准外围器件相连, 包括 FLASH、RAM、网络控制器、LCD 显示驱动器、A/D 转换器和 MCU 等。标准 SPI 总线一般使用 4 条线: 串行时钟线 (SCLK)、主机输入/从机输出数据线 MISO、主机输出/从机输入数据线 MOSI 和低电平有效的从机选择线 NSS。

WS63 提供 SPI0-1 共 2 组可配置的全双工标准 SPI 外设, SPI 规格如下:

- 支持 SPI 帧格式, 分为以下三种:
 - Motorola 帧格式
 - TI (Teaxs Instruments) 帧格式
 - National Microwire 帧格式
- 每个 SPI 具有收发分开的位宽为 32bit×8 的 FIFO。
- 支持最大传输位宽为 32bit。

4.2 功能描述

SPI 主模式支持轮询模式读写、中断模式读写以及 DMA 模式读写；从模式支持中断模式读写和 DMA 模式读写。

如果 SPI 驱动想配置 DMA 模式读写数据，需要确保 DMA 驱动已经初始化。DMA 初始化请参考“7 DMA”进行配置。

SPI 模块提供的接口及功能如下：

- `uapi_spi_init`：初始化 SPI（包括：主从设备、极性、相性、帧协议、传输频率、传输位宽等设定）。
- `uapi_spi_deinit`：去初始化 SPI（关闭相应的 SPI 单元，释放资源）。
- `uapi_spi_get_attr`：获取 SPI 的基础配置参数（主从模式、时钟极性、时钟相位、时钟分频系数、SPI 工作频率、串行传输协议、SPI 帧格式、SPI 帧长度、SPI 传输模式等）。
- `uapi_spi_set_attr`：设置 SPI 的基础配置参数。
- `uapi_spi_get_extra_attr`：获取 SPI 的高级配置参数（SPI 是否使用 DMA 发送数据、SPI 是否使用 DMA 接收数据、QSPI 参数等）。
- `uapi_spi_set_extra_attr`：设置 SPI 的高级配置参数。
- `uapi_spi_select_slave`：片选。
- `uapi_spi_master_write`：SPI 主机半双工发送数据。
- `uapi_spi_master_read`：SPI 主机半双工接收数据。
- `uapi_spi_master_writeread`：SPI 主机全双工收发数据。
- `uapi_spi_slave_read`：SPI 从机半双工接收数据。
- `uapi_spi_slave_write`：SPI 从机半双工发送数据。

4.3 开发指引

SPI 用于对接支持 SPI 协议的设备，SPI 单元可以作为主设备或从设备。以 SPI 单元作为主设备为例，写数据操作如下：

步骤 1 通过 IO 复用，复用 SPI 功能用到的管脚为 SPI 功能。

管脚复用各功能请参考 `platform_core.h` 中各个管脚功能的定义。

```
#define SPI_PIN_MISO_PINMUX    HAL_PIO_FUNC_SPI2_M1
```

```

#define SPI_PIN_MOSI_PINMUX      HAL_GPIO_FUNC_SPI2_M1
#define SPI_PIN_CLK_PINMUX      HAL_GPIO_FUNC_SPI2_M1
#define SPI_PIN_CS_PINMUX       HAL_GPIO_FUNC_SPI2_M1
#define SPI_PIN_MISO            S_MGPIO16
#define SPI_PIN_MOSI            S_MGPIO17
#define SPI_PIN_CLK             S_MGPIO18
#define SPI_PIN_CS              S_MGPIO19
void usr_spi_io_init(void)
{
    /* 设置spi pinmux */

    uapi_pin_set_mode(SPI_PIN_MISO, SPI_PIN_MISO_PINMUX);    /* 设置 spi miso
pinmux */
    uapi_pin_set_mode(SPI_PIN_MOSI, SPI_PIN_MOSI_PINMUX);    /* 设置 spi mosi
pinmux */
    uapi_pin_set_mode(SPI_PIN_CLK, SPI_PIN_CLK_PINMUX);      /* 设置 spi clk pinmux
*/
    uapi_pin_set_mode(SPI_PIN_CS, SPI_PIN_CS_PINMUX);        /* 设置 spi cs pinmux
*/
}

```

步骤 2 调用 uapi_spi_init, 初始化 SPI 资源, 选择 SPI 功能单元以及配置 SPI 参数。

```

#define TEST_SPI                SPI_BUS_2
#define BUS_CLOCK                32000000    /* 32M */
#define SPI_FREQUENCY            2
errcode_t usr_spi_init(void)
{
    spi_attr_t config = { 0 };
    spi_extra_attr_t ext_config = { 0 };
    ext_config.sspi_param.wait_cycles = 0x10;
    usr_spi_io_init();

    config.freq_mhz = SPI_FREQUENCY;          /* spi 分频值 */

    config.is_slave = false;                  /* 主机模式 */

    config.frame_size = HAL_SPI_FRAME_SIZE_8; /* spi 帧大小, 使用8位
*/

    config.salve_num = 1;                    /* 使用片选 0 */

    config.spi_frame_format = HAL_SPI_FRAME_FORMAT_STANDARD; /* spi传输模式: 标
准spi */
}

```

```

    config.bus_clk = BUS_CLOCK;                                /* spi传输速率 */

    config.frame_format = SPI_CFG_FRAME_FORMAT_MOTOROLA_SPI;    /* spi协议格式：摩
托罗拉SPI协议格式 */

    config.tmod = HAL_SPI_TRANS_MODE_TXRX;                    /* spi传输模式：收发
模式 */

    config.clk_phase = SPI_CFG_CLK_CPHA_0;                    /* spi相位：
SPI_CFG_CLK_CPHA_0 */

    config.clk_polarity = SPI_CFG_CLK_CPOL_0;                /* spi极性：
SPI_CFG_CLK_CPOL_0 */

    /* 初始化spi */

    errcode_t err = uapi_spi_init(TEST_SPI, &config, &ext_config);
    return err;
}

```

步骤 3 调用 uapi_spi_master_writeread，进行 SPI 主设备读写操作。

以主设备读写数据为例：

```

errcode_t usr_spi_writeread(uint8_t *wdata, uint8_t wlen, uint8_t *rdata, uint8_t rlen)
{
    spi_xfer_data_t spi_rcv_xfer = { 0 };

    spi_rcv_xfer.tx_buff = wdata;                                /* 设置 tx buff */

    spi_rcv_xfer.tx_bytes = wlen;                                /* 设置 tx buff 长度 */

    spi_rcv_xfer.rx_buff = rdata;                                /* 设置 rx buff */

    spi_rcv_xfer.rx_bytes = rlen;                                /* 设置 rx buff 长度 */

    spi_porting_set_rx_mode(TEST_SPI, rlen);                    /* 设置 读写接口的 rx 接
收模式*/

    return uapi_spi_master_writeread(TEST_SPI, &spi_rcv_xfer, 100); /* 读取数据 */
}

```

----结束

4.4 注意事项

- 当不再使用 SPI 时，必须调用 `uapi_spi_deinit` 进行资源释放，否则在进行初始化时将返回错误。
- 使用 microwire 帧协议时，由于 microwire 帧协议限制，主设备只能发送 8bit 位宽数据。
- 芯片作为主设备时，如果从设备速率较慢，主设备在每次调用读写接口后进行适当延时，避免从设备因读写数据太慢导致数据出错。

5 I2C

- [5.1 概述](#)
- [5.2 功能描述](#)
- [5.3 开发指引](#)
- [5.4 注意事项](#)

5.1 概述

IIC (Inter-Integrated Circuit) 也叫做 I2C, 译作集成电路总线, 是一种串行通信总线, 使用主从架构, 便于 MCU 与周边设备组件之间的通讯。

I2C 总线包含两条线: SDA (Serial Data Line) 和 SCL (Serial Clock Line), 其中 SDA 是数据线, SCL 是时钟线。I2C 总线上的每个设备都有一个唯一的地址, 主机可以通过该地址与设备进行通信。

WS63 提供了 I2C0 ~ I2C1 共 2 组支持 Master 模式的 I2C 外设, I2C 规格如下:

- 支持标速、快速二种工作模式, 在串行 8 位双向数据传输场景下, 标准模式下可达 100kbit/s, 快速模式下可达 400kbit/s。
- 支持位宽为 32bit×8 的 FIFO。
- 支持 7bit/10bit 寻址模式。

5.2 功能描述

I2C 模块提供的接口及功能如下:

- uapi_i2c_master_init: 初始化该 I2C 设备为主机，需要传入的参数有总线号、波特率、高速模式主机码（WS63 不支持高速模式，传入 0 即可）。
- uapi_i2c_deinit: 去初始化 I2C 设备，支持主从机。
- uapi_i2c_master_write: I2C 主机将数据发送到目标从机上，使用轮询模式。
- uapi_i2c_master_read: 主机接收来自目标 I2C 从机的数据，使用轮询模式。
- uapi_i2c_master_writeread: 主机发送数据到目标 I2C 从机，并接收来自此从机的数据，使用轮询模式。
- uapi_i2c_set_baudrate: 对已初始化的 I2C 重置波特率，支持主从机。

5.3 开发指引

I2C 用于对接支持 I2C 协议的设备，I2C 单元可以作为主设备。以 I2C 单元作为主设备为例：

步骤 1 通过 IO 复用，将用到的管脚复用为 I2C 功能。

步骤 2 调用 uapi_i2c_init 接口，初始化 I2C 资源，此处以初始化 I2C 主机为例：

```
#define TEST_I2C                I2C_BUS_0
#define I2C_BAUDRATE            400000    /* 400kHz */
#define I2C_PIN_CLK_PINMUX      PIN_MODE_2
#define I2C_PIN_DAT_PINMUX      PIN_MODE_2
#define I2C_PIN_CLK             GPIO_18
#define I2C_PIN_DAT             GPIO_17
errcode_t sample_i2c_init(void)
{
    /* 设置 i2c pinmux */

    uapi_pin_set_mode(I2C_PIN_CLK, I2C_PIN_CLK_PINMUX);    /* 设置 i2c clk pinmux */

    uapi_pin_set_mode(I2C_PIN_DAT, I2C_PIN_DAT_PINMUX);    /* 设置 i2c dat pinmux */

    /* 初始化 i2c */

    return uapi_i2c_master_init(TEST_I2C, I2C_BAUDRATE, 0); /* 初始化 i2c0 */
}
```

步骤 3 调用 uapi_i2c_master_write 接口，实现主机发送数据。

```
errcode_t sample_i2c_write(uint8_t *data, uint8_t len, uint16_t addr)
{
    i2c_data_t i2c_send_data = { 0 };
}
```

```
i2c_send_data.send_buf = data;                                /* 设置 tx buff */  
i2c_send_data.send_len = len;                                /* 设置 tx buff 长度 */  
return uapi_i2c_master_write(TEST_I2C, addr, &i2c_send_data); /* 发送数据 */  
}
```

----结束

5.4 注意事项

- 函数 `uapi_i2c_set_baudrate` 需要先初始化，再调用，方便修改波特率。如果 I2C 未经过初始化，直接调用 `uapi_i2c_set_baudrate` 函数，会返回错误码 `ERRCODE_I2C_NOT_INIT`。
- 需要主动确保数据发送指针 `send_buf` 和数据接收指针 `receive_buf` 不可传入空指针。
- 当发送的数据大于对接设备的可接受范围时，会发送失败；如果发送数据失败，再切换另一个 I2C 设备继续发送时，会造成总线挂死，所有 I2C 设备都无法正确发送数据。
- `uapi_i2c_master_init` 不能多次初始化，使用完成后需要调用 `uapi_i2c_deinit` 进行去初始化。

6 ADC

- [6.1 概述](#)
- [6.2 功能描述](#)
- [6.3 开发指引](#)
- [6.4 注意事项](#)

6.1 概述

ADC（Analog-to-Digital Converter）模/数转换器，是指将连续变化的模拟信号转换为离散的数位信号的器件。

真实世界的模拟信号，例如温度、压力、声音或者图像等，需要转换成更容易储存、处理和发送的数字信号，模/数转换器可以实现这个功能，可应用于电量检测、按键检测等。

6.2 功能描述

ADC 模块提供的接口及功能如下：

- uapi_adc_init：初始化 ADC。
- uapi_adc_deinit：去初始化 ADC。
- uapi_adc_power_en：对 adc 进行校准。
- uapi_adc_open_channel：对通道进行管脚复用配置。
- uapi_adc_close_channel：关闭对应通道的管脚复用

- uapi_adc_auto_scan_ch_enable: 对需要扫描的通道进行配置, 并开始扫描
- uapi_adc_auto_scan_disable: 停止扫描

6.3 开发指引

示例:

ADC 使能:

```
int hadc_power_on_demo(void)
{
    uapi_adc_power_en(AFE_HADC_MODE, true);
    return TEST_OK;
}
```

ADC 闲时关闭

```
int hadc_power_off_demo(void)
{
    uapi_adc_power_en(AFE_HADC_MODE, false);
    return TEST_OK;
}
```

示例代码:

```
void test_adc_callback(uint8_t ch, uint32_t *buffer)
{
    for (uint32_t i = 0; i < length; i++) {
        printf("channel: %d, voltage: %dmv\r\n", ch, buffer[i]);
    }
}

void test_adc_stop_auto_scan(uint8_t channel)
{
    uapi_adc_auto_scan_ch_disable(channel);
}

void test_adc_start_auto_scan(uint8_t channel, adc_scan_config_t config, adc_callback_t callback)
{
    uapi_adc_init(ADC_CLOCK_500KHZ);
    uapi_adc_power_en(AFE_SCAN_MODE_MAX_NUM, true);
    uapi_adc_auto_scan_ch_enable((uint8_t)channel, config, test_adc_callback);
}
```

```
void test_adc()
{
    adc_scan_config_t config = {.type = 0, .freq = 1};
    test_adc_start_auto_scan(0, config, test_adc_callback);
    test_adc_stop_auto_scan(0); //通过该接口，调用打印接口输出adc转换结果
}
```

6.4 注意事项

量程问题如下：

- 模拟输入电压范围

受限于数模复用的 GPIO 供电电压，ADC 参考电压为 0-3.3V，有六个端口可以输入电压值。

7 DMA

- [7.1 概述](#)
- [7.2 功能描述](#)
- [7.3 开发指引](#)
- [7.4 注意事项](#)

7.1 概述

DMA (Directory Memory Access) 直接存储器访问是一种完全由硬件执行数据交换的工作方式。在这种方式中，直接存储器访问控制器 DMAC (Directory Memory Access Controller) 直接在存储器和外设、外设和外设、存储器和存储器之间进行数据传输，减少处理器的干涉和开销。

DMA 方式一般用于高速传输成组的数据。DMAC 在收到 DMA 传输请求后根据 CPU 对通道的配置启动总线主控制器，向存储器和外设发出地址和控制信号，对传输数据的个数计数，并以中断方式向 CPU 报告传输操作的结束或错误。

提供的 DMA 规格如下：

- 支持存储器到存储器、存储器到外设、外设到存储器 传输类型。
- MCU 侧 DMA 支持 8 个通道，16 个硬件握手接口，且通道参数优先级可配置。
- 所有通道支持单个包长度最大 4095 个数据。
- 支持大小端可配置。

7.2 功能描述

说明

如果需要在 SPI/UART 中使用 DMA 传输数据，需要在系统启动时进行 DMA 初始化。

DMA 模块提供的接口及功能如下：

- uapi_dma_init：初始化 DMA。
- uapi_dma_deinit：去初始化 DMA。
- uapi_dma_open：打开 DMA。
- uapi_dma_close：关闭 DMA。
- uapi_dma_start_transfer：启动指定通道的 DMA 传输。
- uapi_dma_end_transfer：停止指定通道的 DMA 传输。
- uapi_dma_transfer_memory_single：通过 DMA 通道传输类型为内存到内存的数据。
- uapi_dma_configure_peripheral_transfer_single：通过 DMA 通道传输类型为内存到外设或外设到内存的数据。
- uapi_dma_enbale_lli：启用 DMA 链表传输。
- uapi_dma_transfer_memory_lli：通过 DMA 通道以链表模式传输类型为内存到内存的数据。
- uapi_dma_configure_peripheral_transfer_lli：通过 DMA 通道以链表模式传输类型为内存到外设或外设到内存的数据。

7.3 开发指引

DMA 接口仅对外提供存储器到存储器的拷贝功能（其他拷贝方式可参考本文档内对应外设驱动开发指引），操作步骤如下：

步骤 1 调用 uapi_dma_init 接口，初始化 DMA 模块。

步骤 2 调用 uapi_dma_open_ch 接口，打开 DMA 通道。

步骤 3 调用 uapi_dma_transfer 接口，DMA 开始传输，通过参数 block 可以设置是否为阻塞模式。

----结束

示例:

```
#include "dma.h"
#include "hal_dma.h"

/* 传输完成后回调函数处理 */

static bool g_dma_trans_done;
static bool g_dma_trans_succ;
void test_dma_trans_done_callback(uint8_t intr, uint8_t channel, uintptr_t arg)
{
    unused(channel);
    unused(arg);
    switch (intr) {
        case HAL_DMA_INTERRUPT_TFR:
            g_dma_trans_done = true;
            g_dma_trans_succ = true;
            break;
        case HAL_DMA_INTERRUPT_BLOCK:
            g_dma_trans_done = true;
            g_dma_trans_succ = true;
            break;
        case HAL_DMA_INTERRUPT_ERR:
            g_dma_trans_done = true;
            g_dma_trans_succ = false;
            break;
        default:
            break;
    }
    osal_printk("[DMA] int_type is %d. \r\n", intr);
}

static void test_fill_test_buffer(void *data, unsigned int length)
{
    for (unsigned int i = 0; i < length; i++) {
        *((unsigned char *)data + i) = (unsigned char)i;
    }
}

static void test_clear_test_buffer(void *data, unsigned int length)
{
    memset_s(data, length, 0, length);
}

errcode_t test_dma_mem_to_mem_single(void)
{
    dma_ch_user_memory_config_t transfer_config;

    /* 填充源地址要发送的数据 */
```



```
test_fill_test_buffer((void*)(uintptr_t)g_dma_src_data, sizeof(g_dma_src_data));
/* 清空目的地址的数据 */

test_clear_test_buffer((void*)(uintptr_t)g_dma_desc_data, sizeof(g_dma_desc_data));
/* 初始化DMA */

uapi_dma_init();
/* 开启DMA模块 */

uapi_dma_open();
/* 源地址 */

transfer_config.src = ((uint32_t)(uintptr_t)g_dma_src_data);
/* 目的地址 */

transfer_config.dest = ((uint32_t)(uintptr_t)g_dma_desc_data);
/* 传输数目 */

transfer_config.transfer_num = 100;
/* 优先级0-3, 0最低 */

transfer_config.priority = 0;
/* 传输宽度 0:1字节 1:2字节 2:4字节 */

transfer_config.width = 0;
/* 调用接口按块发送函数, 并注册回调函数 */

if (uapi_dma_transfer_memory_single(&transfer_config, test_dma_trans_done_callback, 0) !=
    ERRCODE_SUCC) {
    return ERRCODE_FAIL;
}
/* 等待发送完成 */

while (!g_dma_trans_done) {}
if (!g_dma_trans_succ) {
    return ERRCODE_FAIL;
}

return ERRCODE_SUCC;
}
```

7.4 注意事项

建议仅在需要非阻塞进行数据拷贝的场景下使用 DMA, 此时可让出 CPU, 传输完成之后 CPU 会上报中断, 可以在回调函数中根据事件类型判断传输成功与失败。传输阻塞场景下, 仍建议使用 memcpy_s 进行数据拷贝。

8 PWM

- [8.1 概述](#)
- [8.2 功能描述](#)
- [8.3 开发指引](#)
- [8.4 注意事项](#)

8.1 概述

PWM (Pulse Width Modulation) 脉宽调制模块通过对一系列脉冲的宽度进行调制，等效出所需波形。即对模拟信号电平进行数字编码，通过调节频率、占空比的变化来调节信号的变化。

PWM 规格如下：

- 支持 8 路 PWM 输出，寄存器单独可配置。
- 支持 0 电平宽度和 1 电平宽度可调。
- 支持固定周期数发送模式。
- 支持发送完成中断，支持中断清除和中断查询。

8.2 功能描述

PWM 模块提供的接口及功能如下：

- `uapi_pwm_init`：初始化 PWM。
- `uapi_pwm_deinit`：去初始化 PWM。

- uapi_pwm_open: 打开 PWM 通道。
- uapi_pwm_close: 关闭 PWM 通道。
- uapi_pwm_register_interrupt: 为 PWM 注册中断回调。
- uapi_pwm_unregister_interrupt: 去 PWM 注册中断回调。
- uapi_pwm_start: 启动 PWM 信号输出。
- uapi_pwm_stop: 停止 PWM 信号输出。

8.3 开发指引

PWM 利用微处理器的数字输出对模拟电路进行控制，操作步骤如下：

步骤 1 将 IO 复用为 PWM 功能。

步骤 2 调用 uapi_pwm_init 对 PWM 进行初始化。

步骤 3 调用 uapi_pwm_open，配置 PWM 参数，打开指定通道。

步骤 4 调用 uapi_pwm_register_interrupt 接口，注册 PWM 中断的回调函数。

步骤 5 调用 uapi_pwm_start 接口，开启指定 ID 的 PWM 信号输出。

步骤 6 调用 uapi_pwm_close 接口，停止指定 ID 的 PWM 信号输出。

步骤 7 调用 uapi_pwm_deinit 接口，去初始化指定 ID 的 PWM。

---结束

示例：

```
#include "pwm.h"
#include "pwm_porting.h"
#define TEST_MAX_TIMES 10
#define TEST_DELAY_MS 1000
/* PWM注册中断回调函数 */
static errcode_t pwm_test_callback(pwm_channel_t channel)
{
    osal_printk("PWM channel number is %d, func of interrupt start. \r\n", channel);
    uapi_pwm_isr(channel);
    return ERRCODE_SUCC;
}
void test_pwm_sample(pin_t pin, pin_mode_t mode, pwm_channel_t channel)
```

```
{
    /* 设置循环次数 */
    unsigned int test_times;

    /* 配置 low_time、high_time、cycles, repeat. 当repeat为true时候, cycles无效 */

    /* offset_time 未使用到配置为0 */

    pwm_config_t cfg_repeat = { 100, 100, 0, 0, true };

    /* 设置可作为PWM IO的模式 */
    uapi_pin_set_mode(pin, mode);
    uapi_pwm_init();

    /* 打开指定channel的PWM */
    uapi_pwm_open(channel, &cfg_repeat);

    /* 注册回调函数 */
    uapi_pwm_register_interrupt(channel, pwm_test_callback);

    /* 启动指定channel pwm输出 */
    uapi_pwm_start(channel);

    /* 当前设置为循环输出, 循环TEST_MAX_TIMES次, 每次delay TEST_DELAY_MS,后关闭
    pwm输出 */
    for (test_times = 0; test_times <= TEST_MAX_TIMES; test_times++) {
        if (test_times == TEST_MAX_TIMES) {
            uapi_pwm_close(channel);
            osal_printk("now close the pwm output and trigger interrupt \r\n");
        }
        osal_mdelay(TEST_DELAY_MS);
    }
    uapi_pwm_deinit();
    return;
}
```

8.4 注意事项

- 在调用 uapi_pwm_deinit 接口之前, 需要先调用 uapi_pwm_close 接口。
- PWM 调用 uapi_pwm_stop/uapi_pwm_close 时不支持在中断中调用。
- PWM 不支持占空比为 0。
- PWM 不支持多路互补输出。

- 深睡唤醒之后需要先配置复用关系，再调用 `uapi_pwm_deinit` 接口去初始化，之后调用 `uapi_pwm_init` 接口初始化。

9 WDT

- [9.1 概述](#)
- [9.2 功能描述](#)
- [9.3 开发指引](#)
- [9.4 注意事项](#)

9.1 概述

WDT (Watch Dog Timer)

看门狗计时器，一般用于 CPU 运行异常时实现异常恢复，如果系统正常运行，会定期喂狗，以防止计时器超时。如果系统由于某种原因停止运行或无法正常喂狗，导致计时器在设定的超时时间内未被重置，此时看门狗会认为系统出现故障，触发相应的处理措施，如复位系统或执行特定的错误处理程序。

WDT 规格如下：

- 拥有一个 CPU 看门狗以及一个 PMU 看门狗，其中 PMU 看门狗不对用户开放使用。
- CPU 看门狗超时时间支持 2s ~ 108s 可调。
- CPU 看门狗支持直接复位以及中断后复位两种工作模式。

9.2 功能描述

WDT 模块提供的接口及功能如下：

- uapi_watchdog_init: 初始化 Watchdog 功能, 设置看门狗超时时间, 单位 s。
- uapi_watchdog_deinit: 去初始化 Watchdog 功能。
- uapi_watchdog_set_time: 设置看门狗超时时间, 单位 s (如不设置, 默认时间是 8s)。
- uapi_watchdog_enable: 使能看门狗。
- uapi_watchdog_kick: 重新启动计数器。
- uapi_watchdog_disable: 关闭看门狗。
- uapi_watchdog_get_left_time: 获取看门狗剩余时间, 单位 ms。

9.3 开发指引

WDT 一般用于检测是否死机, 如果超过喂狗等待的时间没有进行喂狗操作, 根据 Watchdog 配置的使能模式产生一个系统复位或者上报狗中断。参考代码如下:

步骤 1 调用 uapi_watchdog_init 初始化并设置看门狗超时时间。

步骤 2 调用 uapi_watchdog_enable, 使能看门狗模块, 可配置复位模式和中断模式。

步骤 3 调用 uapi_watchdog_kick, 进行喂狗操作, 此时在 idle 任务中已实现了喂狗操作。

步骤 4 调用 uapi_watchdog_get_left_time, 获取看门狗定时器剩余时间 (可选)。

步骤 5 调用 uapi_watchdog_disable, 关闭看门狗 (正常情况下不建议关闭看门狗)。

----结束

示例:

```
#include "watchdog.h"
#include "watchdog_porting.h"
void sample_wdt(void)
{
    uint32_t sample_remain_ms;
    /* 设置看门狗超时时间 */

    uapi_watchdog_init(CHIP_WDT_TIMEOUT_32S);/* 设置超时时间 */

    uapi_watchdog_enable(WDT_MODE_RESET);/* 使能看门狗 */

    osal_mdelay(5000); /* delay 5000 ms */

    uapi_watchdog_kick(); /* 喂狗 */
}
```

```
uapi_watchdog_get_left_time(&sample_remain_ms); /* 获取剩余超时时间 */  
osal_printk("sample_remain_ms = %x! \n", sample_remain_ms);  
uapi_watchdog_disable(); /* 关闭看门狗 */  
}
```

9.4 注意事项

- 如果获取时间为 0xFFFFFFFF，说明看门狗处于未使能状态。
- 看门狗在 SDK 中已经使能且已存在喂狗动作，在特殊场景下，如需长时间占用 CPU，可将看门狗去使能或在业务代码中增加喂狗操作，避免正常业务场景引起看门狗复位。
- 看门狗默认超时时间为 8s，SSB 中将狗重新配置为 15s，使用时请注意。一般情况下，不建议修改超时时间。

10 Timer

- [10.1 概述](#)
- [10.2 功能描述](#)
- [10.3 开发指引](#)
- [10.4 注意事项](#)

10.1 概述

Timer 是一种用来计时和产生定时事件的重要模块。它通常由一个计数器和一些相关的寄存器组成。定时器的核心功能是根据设定的时钟源和预设的计数值来进行计数，并在特定条件下产生中断或触发其他事件。

Timer 规格如下：

- 提供 3 个定时器（Timer0 ~ 2），其中 Timer0 用于支撑系统时钟，Timer1 和 Timer2 提供给业务使用。
- Timer1 提供 6 个软件定时器，Timer2 提供 4 软件定时器。
- 每个定时器提供一个 32 位寄存器用于计数。
- 支持超时中断以及重装载值。

10.2 功能描述

Timer 模块提供的接口及功能如下：

- `uapi_timer_adapter`：适配定时器配置。

- uapi_timer_init: 初始化 Timer。
- uapi_timer_deinit: 去初始化 Timer。
- uapi_timer_create: 创建定时器。
- uapi_timer_delete: 删除指定定时器。
- uapi_timer_start: 开启指定高精度定时器，开始计时。
- uapi_timer_stop: 停止当前定时器计时。
- uapi_get_time_us: 获取当前计时值。

10.3 开发指引

使用 Timer 驱动接口创建一个 5ms 周期触发中断的定时器，参考步骤如下：

步骤 1 调用 uapi_timer_adapter 接口，配置定时器索引、定时器中断号和中断优先级。

步骤 2 调用 uapi_timer_init 接口，初始化定时器功能。

步骤 3 调用 uapi_timer_create 接口，创建一个高精度定时器，函数参数句柄为唯一定时器标识。

步骤 4 调用 uapi_timer_start 接口，设置超时时间、超时回调函数、回调函数入参以及启动定时器。

步骤 5 调用 uapi_timer_stop 接口，停止当前定时器计时。

步骤 6 调用 uapi_timer_delete 接口，删除当前定时器。

----结束

示例：

```
#include "timer.h"
#define DELAY_5MS      5000
#define DELAY_1S       1000000
#define TIMER_IRQ_PRIO 3    /* 中断优先级范围，从高到低： 0~7 */

static timer_handle_t timer1_handle = 0;
static void timer1_callback(uintptr_t data);
void timer1_callback(uintptr_t data)
{
    unused(data);
    osal_printf("Timer1 5ms int test!\r\n");
}
```

```
/* 开启下一次timer中断 */
uapi_timer_start(timer1_handle, DELAY_5MS, timer1_callback, 0);
}
errcode_t test_timer_sample(void)
{
    errcode_t ret;
    /* timer 软件初始化 */
    uapi_timer_init();
    /* 设置 timer1 硬件初始化, 设置中断号, 配置优先级 */
    ret = uapi_timer_adapter(TIMER_INDEX_1, TIMER_1_IRQN, TIMER_IRQ_PRIO);
    /* 创建 timer1 软件定时器控制句柄 */
    uapi_timer_create(TIMER_INDEX_1, &timer1_handle);
    /* 启动定时器 */
    uapi_timer_start(timer1_handle, DELAY_5MS, timer1_callback, 0);
    osal_mdelay(DELAY_1S);
    /* 停止定时器 */
    uapi_timer_stop(timer1_handle);
    /* 删除定时器 */
    uapi_timer_delete(timer1_handle);
    return ret;
}
```

10.4 注意事项

- Timer 创建的定时器超时时间单位为 μs 。
- 默认最多可同时创建 2 个高精度定时器 (Timer1~2), Timer1 提供 6 个软件定时器, Timer2 提供 4 软件定时器。
- Timer0 默认作为 liteos 的系统时钟源, 禁止使用 uapi 接口配置。
- 在非低功耗模式下, Timer 可配置的最大计数值为 $2^{32}-1\text{s}$ 。
- 确定不需要使用当前高精度定时器后, 需要调用 uapi_timer_delete 接口, 释放该定时器资源。
- 禁止在 Timer 的回调函数中调用 uapi_timer_stop、uapi_timer_delete 等接口。

11 SysTick

- 11.1 概述
- 11.2 功能描述
- 11.3 开发指引
- 11.4 注意事项

11.1 概述

SysTick 是单片机系统中的一种硬件设备或功能模块，用于提供精确的时间基准和定时功能。

系统定时规格如下：

- SysTick 提供了一个 32 位和一个 16 的寄存器用于存放计数值，最高可计数到 $2^{48}-1$ 。
- 可使用外部 32.768kHz 晶振或内部 32kHz 时钟作为时钟源。

11.2 功能描述

SysTick 模块提供的接口及功能如下：

- `uapi_systick_init`：初始化 SysTick。
- `uapi_systick_deinit`：去初始化 SysTick。
- `uapi_systick_count_clear`：清除 SysTick 计数。
- `uapi_systick_get_count`：获取 SysTick 计数值。

- uapi_systick_get_s: 获取 Systick 计数秒值。
- uapi_systick_get_ms: 获取 Systick 计数毫秒值。
- uapi_systick_get_us: 获取 Systick 计数微秒值。
- uapi_systick_delay_count: 按 count 计数延时。
- uapi_systick_delay_s: 按秒数延时。
- uapi_systick_delay_ms: 按毫秒数延时。
- uapi_systick_delay_us: 按微秒数延时。

11.3 开发指引

步骤 1 调用 uapi_systick_init 接口，初始化 Systick 模块。

步骤 2 调用 uapi_systick_get_count 接口，获取当前 Systick 计数值。

步骤 3 调用 uapi_systick_delay_ms 接口，延迟传入的时间。

步骤 4 再次调用 uapi_systick_get_count 接口，获取当前 Systick 计数值。

----结束

示例:

```
#include "systick.h"
void test_systick_sample(void)
{
    uint64_t count_before_delay_count;
    uint64_t count_after_delay_count;
    /* systick模块初始化 */
    uapi_systick_init();
    /* 通过count数差值验证延迟时间 */
    count_before_delay_count= uapi_systick_get_count();
    uapi_systick_delay_ms(1000);
    count_after_delay_count = uapi_systick_get_count();
    osal_printk("test case delay count %lu.\r\n", count_before_delay_count -
count_after_delay_count);
}
```

11.4 注意事项

- SysTick 时钟源使用外部 32.768k 或者内部 32k 时钟，最小计数单元为 30 μ s 左右，使用 μ s 延时接口请注意。
- SysTick 一般用于提供了一个稳定的时钟信号，作为整个单片机系统的基准时钟。高精度延时则使用 TCXO。

12 TCXO

- 12.1 概述
- 12.2 功能描述
- 12.3 开发指引
- 12.4 注意事项

12.1 概述

TCXO (Temperature Compensated Crystal Oscillator) 是一种温度补偿晶体振荡器, 通过在电路中引入温度传感器和温度补偿电路, 以降低温度对振荡频率的影响, 从而提供更稳定的时钟信号。WS63 芯片内置了一块 TCXO 晶振及其计数单元, 用于计数和延时, 用户也可以修改时钟配置, 使用外部晶振作为 TCXO 计数单元的时钟输入, WS63 TCXO 规格如下:

- 内部 TCXO 高达 32M, 最小计数单元约为 32ns。
- TCXO 计数器提供了两个 32 位的寄存器用于存放计数值, 最高可计数到 $2^{64}-1$ 。

12.2 功能描述

TCXO 模块提供的接口及功能如下:

- `uapi_tcxo_init`: 初始化 TCXO。
- `uapi_tcxo_deinit`: 去初始化 TCXO。
- `uapi_tcxo_get_count`: 获取 TCXO 计数值。
- `uapi_tcxo_get_ms`: 获取 TCXO 计数毫秒值。

- uapi_tcxo_get_us: 获取 TCXO 计数微秒值。
- uapi_tcxo_delay_ms: 设置延迟毫秒数。
- uapi_tcxo_delay_us: 设置延迟微秒数。

12.3 开发指引

步骤 1 调用 uapi_tcxo_init 接口，初始化 TCXO 模块。

步骤 2 调用 uapi_tcxo_get_count 接口，获取当前 TCXO 计数值。

步骤 3 调用 uapi_tcxo_delay_ms 接口，延迟传入的时间。

----结束

示例:

```
#include "tcxo.h"
void test_tcxo_sample(void)
{
    uint64_t count_before_delay_count;
    uint64_t count_after_delay_count;
    /* tcxo模块初始化 */
    uapi_tcxo_init();
    /* 通过count差值验证延迟时间 */
    count_before_delay_count = uapi_tcxo_get_count();
    uapi_tcxo_delay_ms(1000);
    count_after_delay_count= uapi_tcxo_get_count();
    osal_printk("test case delay count %lu.\r\n", count_before_delay_count -
count_after_delay_count);
    return;
}
```

12.4 注意事项

无。

13 SFC

- [13.1 概述](#)
- [13.2 功能描述](#)
- [13.3 开发指引](#)
- [13.4 注意事项](#)

13.1 概述

Flash 是一种非易失快闪记忆体技术，又称为闪存，通常支持 SPI 协议。Flash 可以通过 SPI 协议实现读、写和擦除等多种命令，部分 Flash 支持 XIP 模式。WS63 芯片可以通过片上 XIP 外设结合 SPI 接口连接外部 Flash 芯片。通过 XIP 和 QSPI，WS63 芯片可以直接以总线的方式从 Flash 中读取指令和数据。

13.2 功能描述

Flash 模块提供的接口及功能如下：

- uapi_sfc_init：初始化 Flash。
- uapi_sfc_init_rom：按照单线读写 512KB 初始化 Flash。
- uapi_sfc_deinit：去初始化 Flash。
- uapi_sfc_reg_read：提供寄存器模式读功能，读取的数据将按字节存入 read_buffer 中。
- uapi_sfc_reg_write：提供寄存器模式写功能，预计写入的数据按字节存入 write_data 中。

- uapi_sfc_reg_erase: 使用寄存器模式进行对 Flash 的擦除, 不使能写回时强制要求地址和大小按扇区对齐。
- uapi_sfc_reg_erase_chip: 使用寄存器模式对整片 Flash 进行擦除。
- uapi_sfc_reg_other_flash_opt: 使用寄存器模式对 Flash 属性进行读写。。
- uapi_sfc_dma_read: 提供 DMA 模式读功能, 读取的数据将按字节存入 read_buffer 中。
- uapi_sfc_dma_write: 提供寄存器模式写功能, 预计写入的数据按字节存入 write_data 中。
- uapi_sfc_suspend: 挂起 SFC。
- uapi_sfc_resume: 恢复 SFC。

13.3 开发指引

示例:

步骤 1 对 Flash 做初始化。

```
const sfc_flash_config_t sfc_cfg = {
    .read_type = FAST_READ_QUAD_OUTPUT,
    .write_type = PAGE_PROGRAM,
    .mapping_addr = 0x200000,
    .mapping_size = 0x800000,
};

static uint32_t sfc_flash_init(void)
{
    return uapi_sfc_init((sfc_flash_config_t *)&sfc_cfg);
}
```

步骤 2 向 Flash 指定地址读取数据。

```
ret = (uint32_t)memcpy_s((void *) (uintptr_t)FLASHBOOT_RAM_ADDR, BOOT_MAX_LEN,
    (void *) (uintptr_t) (g_flash_info.part_info.addr_info.addr + FLASH_START_ADDR),
    LOADER_BOOT_SIGN_HEAD_LEN);
```

----结束

13.4 注意事项

- 对 Flash 进行读写擦操作时，请确保 Flash 已完成初始化。

14 EFUSE

- [14.1 概述](#)
- [14.2 功能描述](#)
- [14.3 开发指引](#)
- [14.4 注意事项](#)

14.1 概述

eFuse 的全称是“电子熔断器” (electronic fuse)，是一种可编程电子保险丝，是一种用于存储信息和保护芯片的非易失性存储器件。ws63 只提供操作用户预留空间接口。

14.2 功能描述

eFuse 模块提供的接口及功能如下：

- uapi_efuse_user_read_bit：从用户预留的 eFuse 空间中读取一位。
- uapi_efuse_user_read_buffer：从用户预留的 eFuse 空间中读取多个字节，进入提供的缓冲区。
- uapi_efuse_user_write_bit：向用户预留 eFuse 空间中的对应 bit 写 1。
- uapi_efuse_user_write_buffer：从提供的缓冲区向用户预留的 eFuse 空间写入多个字节。

14.3 开发指引

示例：

步骤 1 对 EFUSE 做初始化。

```
uapi_efuse_init();
```

步骤 2 按照 buffer 读取 efuse 值。

```
uint8_t efuse_data[8] = {0};  
uint32_t byte_number = 1;  
uint16_t length = 8;  
// 从2048 bit空间中的第10byte开始读取8个字节  
uapi_efuse_user_read_buffer(byte_number, efuse_data, length);
```

----结束

14.4 注意事项

- ws63 用户预留 efuse 空间共 128bit，读写操作不能超过整个 efuse 预留空间。