



✍ moliam.zhang 📅 2021年11月17日 晚上  
📖 2.3k 字 🕒 35 分钟 👁 3822 次

我会详细的解释各个模块的作用，如果只需要代码，则直接看完整代码章节。

## ESP32S2 ADC简介

ESP32S2的ADC有两个控制器，一个是RTC控制器，一个是DIG控制器。官方的adc1\_get\_raw是基于RTC控制器，是已经设置过的最快的采集速度。由于RTC模式下的最大采样速度只有200kHz，正常情况下是能满足需求的，但是如果需要更快的速度，则需要使用DIG控制器，并且使用DMA进行传输。DIG控制器可达到2MHz。

### 5 ADC 特性

表 10: ADC 特性

| 参数          | 描述                            | 最小值 | 最大值 | 单位   |
|-------------|-------------------------------|-----|-----|------|
| DNL (差分非线性) | RTC 控制器; ADC 外接 100 nF 电容;    | -7  | 7   | LSB  |
| INL (积分非线性) | 输入为 DC 信号; 常温 25 °C; Wi-Fi 关闭 | -12 | 12  | LSB  |
| 采样速度        | RTC 控制器                       | —   | 200 | kSPS |
|             | DIG 控制器                       | —   | 2   | MSps |

ADC特性

### RTC控制器

RTC控制器下的初始化就比较简单了，只需要使用两个函数就能初始化完成。

```
adc1_config_width(ADC_WIDTH_BIT_13); //RTC控制器下只支持13位
adc1_config_channel_atten(ADC1_CHANNEL_8, ADC_ATTEN_DB_11); //最大量程为2.6V
```

RTC控制下的取值也很简单

```
value = (adc1_get_raw(ADC1_CHANNEL_8) & 0x1FFF); //取末尾13位数据
```

## DIG控制器

### TEST\_CASE测试

DIG控制器下工作就比较繁杂了，因为esp-idf-4.2版本并没有给出该模式下的API，因此需要自己进行移植。根据TEST\_CASE进行移植。

ADC\_DMA的TEST\_CASE使用

```
#编译driver的test_case，烧录并进行监控
idf.py -T "driver" build && idf.py flash && idf.py monitor
```

监控模式下按下 98+  就是ADC的测试输出结果。

监控例程源码在components/driver/test/adc\_dma\_test目录下。记得查看文件名带s2的文件。

## 移植

### ADC的初始化代码

```
int adc_dig_bsp_init()
{
    adc_digi_config_t config = {
        .conv_limit_en = false,
        .conv_limit_num = 0,
        /** Sample rate = APB_CLK(80 MHz) / (dig_clk.div_num+ 1) / TRIGGER_INTERVAL / 2. */
        .interval = 40,
        .dig_clk.use_apll = 0, // APB clk
        .dig_clk.div_num = 9,
        .dig_clk.div_b = 0,
        .dig_clk.div_a = 0,
        .dma_eof_num = SAR_SIMPLE_NUM*2, //采样的数目*2
    };
    /**
    所谓的样式表就是ADC的采集列表。设置样式表数组的话，就是根据数组依次采集，然后依次放入buf中，比如你创建了样式表有通道1和通道2两个元素，则buf[0]存放通道1的来说，这个样式表对于多通道无顺序采集还是挺方便的
    */
    //设置样式表，可数组
    adc_digi_pattern_table_t adc1_patt = {0};
    //样式表长度
    config.adc1_pattern_len = 1;
    //样式表地址配置赋值
    config.adc1_pattern = &adc1_patt;
    //样式表的量程
    adc1_patt.atten = ADC_ATTEN_11db;
    //样式表的通道
    adc1_patt.channel = ADC1_CHANNEL_8;
    //该通道的引脚初始化
    adc_gpio_init(ADC_UNIT_1, ADC1_CHANNEL_8 );
    //转换模式，单次，见下面的转换模式图
    config.conv_mode = ADC_CONV_SINGLE_UNIT_1;
    //DMA模式下使用数据格式1因此为12位的ADC
    config.format = ADC_DIGI_FORMAT_12BIT;
    //配置的初始化
    adc_digi_controller_config(&config);
    //中断队列
    if (que_adc == NULL) {
        que_adc = xQueueCreate(5, sizeof(adc_dma_event_t));
    } else {
        xQueueReset(que_adc);
    }
    //DMA的中断函数寄存器等
    uint32_t int_mask = SPI_IN_SUC_EOF_INT_ENA;
    uint32_t dma_addr = adc_dma_linker_init();
    adc_dac_dma_isr_register(adc_dma_isr, NULL, int_mask);
    return 0;
}
```

表 169: 样式表寄存器的字段信息

| 样式表寄存器 [7:0] |          |            |
|--------------|----------|------------|
| ch_sel[3:0]  | null     | atten[1:0] |
| 扫描通道         | reserved | 衰减         |

用户可在 [APB\\_SARADC\\_WORK\\_MODE](#) 中配置扫描模式，定义两个 DIG ADC 控制器的工作方式，包括两个控制器各自完全独立工作，或交替工作，或同步工作：

- 单通道模式：SAR ADC1 和 SAR ADC2 各自按照自己的样式表独立进行工作。
- 双通道模式：SAR ADC1 和 SAR ADC2 同时进行采样，即两个 DIG ADC 控制器同步逐条读取各自样式表。
- 交替模式：SAR ADC1 和 SAR ADC2 交替采样，即两个 DIG ADC 控制器交替逐条读取各自样式表。

ADC 最终向 DMA 传递 16 位数据，包括 12/11 位的 ADC 转换结果，及一些因扫描模式不同而有所差别的相关信息，具体为：

- 单通道模式：仅增加 4 位通道选择信息。
- 双通道模式或交替模式：增加 4 位通道选择信息，及 1 位 SAR ADC 选择信息。

每种扫描模式均有其对应的数据格式，即 I 型和 II 型。有关这两种数据格式的具体描述，请见表 170 和表 171。

表 170: I 型 DMA 数据格式

| I 型 DMA 数据格式 [15:0] |            |
|---------------------|------------|
| ch_sel[3:0]         | data[11:0] |
| 通道                  | SAR ADC 信息 |

表 171: II 型 DMA 数据格式

| II 型 DMA 数据格式 [15:0]          |             |            |
|-------------------------------|-------------|------------|
| sar_sel                       | ch_sel[3:0] | data[10:0] |
| SAR ADC $n$ ( $n = 1$ 或 $2$ ) | 通道          | SAR ADC 信息 |

DIG ADC 支持的分辨率最高为 12 位，其中 I 型数据格式最高支持 12 位分辨率，II 型数据格式最高支持 11 位分辨率。

[https://blog.csdn.net/weixin\\_44529321](https://blog.csdn.net/weixin_44529321)

模式解释以及数据格式图

## DMA

很明显，上面有的函数，是没有定义的，需要自己写。

```
uint32_t adc_dma_linker_init(void)
{
    dma1 = (lldesc_t) {
        .size = SAR_SIMPLE_NUM*2*2, //DMA采样数据的两倍
        .owner = 1, //我也不知道，写1就对了，哈哈哈哈，如果你知道，可以评论告诉我
        .buf = &link_buf[0], //DMA的buf，需是8位，但是最后采集的数据需要转成16位
        .qe.stqe_next = NULL, //如果循环采样的话，该值为&dma1
    };
    return (uint32_t)&dma1;
}
```

中断函数注册

```

typedef struct adc_dac_dma_isr_handler_ {
    uint32_t mask;
    intr_handler_t handler;
    void* handler_arg;
    SLIST_ENTRY(adc_dac_dma_isr_handler_) next;
} adc_dac_dma_isr_handler_t;

static SLIST_HEAD(adc_dac_dma_isr_handler_list_, adc_dac_dma_isr_handler_) s_adc_dac_dma_isr_handler_list =
    SLIST_HEAD_INITIALIZER(s_adc_dac_dma_isr_handler_list);
portMUX_TYPE s_isr_handler_list_lock = portMUX_INITIALIZER_UNLOCKED;
static intr_handle_t s_adc_dac_dma_isr_handle;

static IRAM_ATTR void adc_dac_dma_isr_default(void* arg)
{
    uint32_t status = REG_READ(SPI_DMA_INT_ST_REG(3));
    adc_dac_dma_isr_handler_t* it;
    portENTER_CRITICAL_ISR(&s_isr_handler_list_lock);
    SLIST_FOREACH(it, &s_adc_dac_dma_isr_handler_list, next) {
        if (it->mask & status) {
            portEXIT_CRITICAL_ISR(&s_isr_handler_list_lock);
            (*it->handler)(it->handler_arg);
            portENTER_CRITICAL_ISR(&s_isr_handler_list_lock);
        }
    }
    portEXIT_CRITICAL_ISR(&s_isr_handler_list_lock);
    REG_WRITE(SPI_DMA_INT_CLR_REG(3), status);
}

static esp_err_t adc_dac_dma_isr_ensure_installed(void)
{
    esp_err_t err = ESP_OK;
    portENTER_CRITICAL(&s_isr_handler_list_lock);
    if (s_adc_dac_dma_isr_handle) {
        goto out;
    }
    REG_WRITE(SPI_DMA_INT_ENA_REG(3), 0);
    REG_WRITE(SPI_DMA_INT_CLR_REG(3), UINT32_MAX);
    err = esp_intr_alloc(ETS_SPI3_DMA_INTR_SOURCE, 0, &adc_dac_dma_isr_default, NULL, &s_adc_dac_dma_isr_handle);
    if (err != ESP_OK) {
        goto out;
    }
out:
    portEXIT_CRITICAL(&s_isr_handler_list_lock);
    return err;
}

esp_err_t adc_dac_dma_isr_register(intr_handler_t handler, void* handler_arg, uint32_t intr_mask)
{
    esp_err_t err = adc_dac_dma_isr_ensure_installed(); //确认是否安装，该函数也需
    if (err != ESP_OK) {
        return err;
    }

    adc_dac_dma_isr_handler_t* item = malloc(sizeof(*item));
    if (item == NULL) {
        return ESP_ERR_NO_MEM;
    }
    item->handler = handler;
    item->handler_arg = handler_arg;
    item->mask = intr_mask;
    portENTER_CRITICAL(&s_isr_handler_list_lock);
    SLIST_INSERT_HEAD(&s_adc_dac_dma_isr_handler_list, item, next);
    portEXIT_CRITICAL(&s_isr_handler_list_lock);
    return ESP_OK;
}

```

关于SPI3的问题，因为SPI3和ADC是公用的DMA，因此不能同时进行，所以再使用的时候，需要关闭SPI3相关寄存器。也需要包含SPI的一些头文件

### 30.2.5.2 DMA

DIG ADC 控制器允许通过 SPI3 的 DMA 实现直接内存访问，由 DIG ADC 专用定时器产生触发信号。因此，SPI3 DMA 已经在使用时，则 DIG ADC 控制器不可用。用户可通过软件配置 `APB_SARADC_APB_ADC_TRANS` 将 DMA 的数据通路切换到 DIG ADC。关于 DMA 的具体配置，参考 SPI3 的 DMA 控制。

# 数据处理

## DMA的中断处理

```
/** ADC-DMA ISR handler. */
static IRAM_ATTR void adc_dma_isr(void *arg)
{
    uint32_t int_st = REG_READ(SPI_DMA_INT_ST_REG(3));
    int task_awoken = pdFALSE;
    REG_WRITE(SPI_DMA_INT_CLR_REG(3), int_st);
    if (int_st & SPI_IN_SUC_EOF_INT_ST_M) {
        adc_evt.int_msk = int_st;
        xQueueSendFromISR(que_adc, &adc_evt, &task_awoken);
    }
    if (int_st & SPI_IN_DONE_INT_ST) { //完成DMA采集
        adc_evt.int_msk = int_st;
        xQueueSendFromISR(que_adc, &adc_evt, &task_awoken);
    }

    if (task_awoken == pdTRUE) {
        portYIELD_FROM_ISR();
    }
}
```

## 中断完成之后的数据处理

```
esp_err_t adc_get_value(uint16_t *buf, const int num)
{
    adc_dma_event_t evt;
    //链接DMA采样
    adc_dac_dma_linker_start((void *)&dma1, SPI_IN_SUC_EOF_INT_ENA);
    //开始采样
    adc_digi_start();
    while (1) { //等待采样完成，此处有bug，DMA不完成则会一直卡在这里，但是我当前还没有进行修改。
        xQueueReceive(que_adc, &evt, 10 / portTICK_RATE_MS);
        if (evt.int_msk & SPI_IN_SUC_EOF_INT_ENA) {
            break;
        }
    }
    //停止采样
    adc_digi_stop();
    //指针转换，因为DMA的buf只能为8位，但是根据数据格式1，采集到的数据是16为数据，因此需要进行转换
    uint16_t *buf1 = (uint16_t *)link_buf;
    for(int i=0; i<num; i++){
        //DMA数据格式1，忽略通道，直接进行取值。
        buf[i] = (buf1[i] & 0xFFF);
    }
    return ESP_OK;
}
```

# 完整代码

## C文件

```

#include "esp_system.h"
#include "esp_intr_alloc.h"
#include "freertos/FreeRTOS.h"
#include "freertos/task.h"
#include "freertos/queue.h"
#include "driver/adc.h"
#include "driver/dac.h"
#include "driver/rtc_io.h"
#include "driver/gpio.h"
#include "unity.h"
#include "esp_system.h"
#include "esp_event.h"
#include "esp_wifi.h"
#include "esp_log.h"
#include "nvs_flash.h"
#include "adc.h"
#include "soc/adc_periph.h"

#include "soc/system_reg.h"
#include "soc/lldesc.h"
#include "adc.h"

#include "sysCfg.h"
#include "esp_log.h"

static const char *TAG = "adc";
#define debug_i(format,...)      ESP_LOGI(TAG,format,##__VA_ARGS__)
#define debug_w(format,...)      ESP_LOGW(TAG,format,##__VA_ARGS__)
#define debug_e(format,...)      ESP_LOGE(TAG,format,##__VA_ARGS__)

uint8_t link_buf[SAR_SIMPLE_NUM*2*2] = {0};
static lldesc_t dma1 = {0};
static QueueHandle_t que_adc = NULL;
static adc_dma_event_t adc_evt;

/** ADC-DMA ISR handler. */
static IRAM_ATTR void adc_dma_isr(void *arg)
{
    uint32_t int_st = REG_READ(SPI_DMA_INT_ST_REG(3));
    int task_awoken = pdFALSE;
    REG_WRITE(SPI_DMA_INT_CLR_REG(3), int_st);
    if (int_st & SPI_IN_SUC_EOF_INT_ST_M) {
        adc_evt.int_msk = int_st;
        xQueueSendFromISR(que_adc, &adc_evt, &task_awoken);
    }
    if (int_st & SPI_IN_DONE_INT_ST) {
        adc_evt.int_msk = int_st;
        xQueueSendFromISR(que_adc, &adc_evt, &task_awoken);
    }

    if (task_awoken == pdTRUE) {
        portYIELD_FROM_ISR();
    }
}

uint32_t adc_dma_linker_init(void)
{
    dma1 = (lldesc_t) {
        .size = SAR_SIMPLE_NUM*2*2,
        .owner = 1,
        .buf = &link_buf[0],
        .qe.stqe_next = NULL,
    };
    return (uint32_t)&dma1;
}

typedef struct adc_dac_dma_isr_handler_ {
    uint32_t mask;
    intr_handler_t handler;
    void* handler_arg;
    SLIST_ENTRY(adc_dac_dma_isr_handler_) next;
} adc_dac_dma_isr_handler_t;

static SLIST_HEAD(adc_dac_dma_isr_handler_list_, adc_dac_dma_isr_handler_) s_adc_dac_dma_isr_handler_list =
    SLIST_HEAD_INITIALIZER(s_adc_dac_dma_isr_handler_list);
portMUX_TYPE s_isr_handler_list_lock = portMUX_INITIALIZER_UNLOCKED;
static intr_handle_t s_adc_dac_dma_isr_handle;

static IRAM_ATTR void adc_dac_dma_isr_default(void* arg)

```

```

{
    uint32_t status = REG_READ(SPI_DMA_INT_ST_REG(3));
    adc_dac_dma_isr_handler_t* it;
    portENTER_CRITICAL_ISR(&s_isr_handler_list_lock);
    SLIST_FOREACH(it, &s_adc_dac_dma_isr_handler_list, next) {
        if (it->mask & status) {
            portEXIT_CRITICAL_ISR(&s_isr_handler_list_lock);
            (*it->handler)(it->handler_arg);
            portENTER_CRITICAL_ISR(&s_isr_handler_list_lock);
        }
    }
    portEXIT_CRITICAL_ISR(&s_isr_handler_list_lock);
    REG_WRITE(SPI_DMA_INT_CLR_REG(3), status);
}

static esp_err_t adc_dac_dma_isr_ensure_installed(void)
{
    esp_err_t err = ESP_OK;
    portENTER_CRITICAL(&s_isr_handler_list_lock);
    if (s_adc_dac_dma_isr_handle) {
        goto out;
    }
    REG_WRITE(SPI_DMA_INT_ENA_REG(3), 0);
    REG_WRITE(SPI_DMA_INT_CLR_REG(3), UINT32_MAX);
    err = esp_intr_alloc(ETS_SPI3_DMA_INTR_SOURCE, 0, &adc_dac_dma_isr_default, NULL, &s_adc_dac_dma_isr_handle);
    if (err != ESP_OK) {
        goto out;
    }
}

out:
    portEXIT_CRITICAL(&s_isr_handler_list_lock);
    return err;
}

esp_err_t adc_dac_dma_isr_register(intr_handler_t handler, void* handler_arg, uint32_t intr_mask)
{
    esp_err_t err = adc_dac_dma_isr_ensure_installed();
    if (err != ESP_OK) {
        return err;
    }

    adc_dac_dma_isr_handler_t* item = malloc(sizeof(*item));
    if (item == NULL) {
        return ESP_ERR_NO_MEM;
    }
    item->handler = handler;
    item->handler_arg = handler_arg;
    item->mask = intr_mask;
    portENTER_CRITICAL(&s_isr_handler_list_lock);
    SLIST_INSERT_HEAD(&s_adc_dac_dma_isr_handler_list, item, next);
    portEXIT_CRITICAL(&s_isr_handler_list_lock);
    return ESP_OK;
}

void adc_dac_dma_linker_start( void *dma_addr, uint32_t int_msk)
{
    REG_SET_BIT(DPORT_PERIP_CLK_EN_REG, DPORT_APB_SARADC_CLK_EN_M);
    REG_SET_BIT(DPORT_PERIP_CLK_EN_REG, DPORT_SPI3_DMA_CLK_EN_M);
    REG_SET_BIT(DPORT_PERIP_CLK_EN_REG, DPORT_SPI3_CLK_EN);
    REG_CLR_BIT(DPORT_PERIP_RST_EN_REG, DPORT_SPI3_DMA_RST_M);
    REG_CLR_BIT(DPORT_PERIP_RST_EN_REG, DPORT_SPI3_RST_M);
    REG_WRITE(SPI_DMA_INT_CLR_REG(3), 0xFFFFFFFF);
    REG_WRITE(SPI_DMA_INT_ENA_REG(3), int_msk | REG_READ(SPI_DMA_INT_ENA_REG(3)));

    REG_SET_BIT(SPI_DMA_IN_LINK_REG(3), SPI_INLINK_STOP);
    REG_CLR_BIT(SPI_DMA_IN_LINK_REG(3), SPI_INLINK_START);
    SET_PERI_REG_BITS(SPI_DMA_IN_LINK_REG(3), SPI_INLINK_ADDR, (uint32_t)dma_addr, 0);
    REG_SET_BIT(SPI_DMA_CONF_REG(3), SPI_IN_RST);
    REG_CLR_BIT(SPI_DMA_CONF_REG(3), SPI_IN_RST);
    REG_CLR_BIT(SPI_DMA_IN_LINK_REG(3), SPI_INLINK_STOP);
    REG_SET_BIT(SPI_DMA_IN_LINK_REG(3), SPI_INLINK_START);
}

esp_err_t adc_get_value_group(uint16_t *buf, const int num)
{
    adc_dma_event_t evt;

    adc_dac_dma_linker_start((void *)&dma1, SPI_IN_SUC_EOF_INT_ENA);
    adc_digi_start();
    while (1) {
        xQueueReceive(que_adc, &evt, 10 / portTICK_RATE_MS);
        if (evt.int_msk & SPI_IN_SUC_EOF_INT_ENA) {
            break;
        }
    }
}

```



```

    adc_digi_stop();
    uint16_t *buf1 = (uint16_t *)link_buf;
    for(int i=0;i<num;i++){
        buf[i] = (buf1[i] & 0xFF);
    }
    return ESP_OK;
}

int adc_dig_bsp_init()
{
    adc_digi_config_t config = {
        .conv_limit_en = false,
        .conv_limit_num = 0,
        .interval = 40,
        .dig_clk.use_apll = 0, // APB clk
        .dig_clk.div_num = 9,
        .dig_clk.div_b = 0,
        .dig_clk.div_a = 0,
        .dma_eof_num = SAR_SIMPLE_NUM*2,
    };
    adc_digi_pattern_table_t adc1_patt = {0};
    config.adc1_pattern_len = 1;
    config.adc1_pattern = &adc1_patt;
    adc1_patt.atten = ADC_ATTEN_11db;
    adc1_patt.channel = ADC1_CHANNEL_8;
    adc_gpio_init(ADC_UNIT_1, ADC1_CHANNEL_8);
    config.conv_mode = ADC_CONV_SINGLE_UNIT_1;
    config.format = ADC_DIGI_FORMAT_12BIT;
    adc_digi_controller_config(&config);
    if (que_adc == NULL) {
        que_adc = xQueueCreate(5, sizeof(adc_dma_event_t));
    } else {
        xQueueReset(que_adc);
    }
    uint32_t int_mask = SPI_IN_SUC_EOF_INT_ENA;
    uint32_t dma_addr = adc_dma_linker_init();
    adc_dac_dma_isr_register(adc_dma_isr, NULL, int_mask);
    //先采集一次
    uint16_t buf[SAR_SIMPLE_NUM];
    adc_dac_dma_linker_start((void *)dma_addr, int_mask);
    adc_get_value_group(buf, 6);

    return 0;
}

```

## h文件

```

#pragma once

#include "stdint.h"

#define SAR_SIMPLE_NUM 12 //需要采样的bug

typedef struct dma_msg {
    uint32_t int_msk;
    uint8_t *data;
    uint32_t data_len;
} adc_dma_event_t; //用于队列的事件
//外部获取的ADC值的API
esp_err_t adc_get_value_group(uint16_t *buf, const int num);
//外部初始化函数
int adc_dig_bsp_init(void);

```

## 总结

移植ADC\_DMA在浏览器上相关的移植记录很少，而官方又默认使用速率较慢的RTC控制器，在某些应用场景并不能适用，因此需要使用DIG控制器。目前这个代码是使用的保证准确度的分频系数。但是在config初始化的值dig\_clk.div\_num为9，官方介绍该值可以为0~255，低于9可能会导致数据不准，如果还想再提高速率的话，可以考虑将该值减少。

在初始化的DMA的大小时，需要为采样值的四倍以上，否则会进入不到中断，导致卡死在读值的死循环。这个大小让我调试了很久才找出来的。

🔗 [experience](#) [esp32s2](#) 📄 [c](#) [esp32s2](#)

本博客所有文章除特别声明外，均采用 [CC BY-SA 4.0 协议](#)，转载请注明出处！

◀ [数据结构和算法](#)

[消抖](#) ▶

xiaolaba

邮箱

[https://github.com/xiaolaba/ESP32\\_DMA\\_ADC\\_te](https://github.com/xiaolaba/ESP32_DMA_ADC_te)

say something



提交

## 1 评论



xiaolaba

Firefox 105.0

Windows 10.0

5 秒前

回复

非常好的解說.

試了一下, 無法直接移植到 ARDUINO 的 IDE 做編譯.

Powered By [Valine](#)  
v1.4.14