GR551x外设示例用户手册

版本： **1.8**

发布日期： **2021-04-25**

深圳市汇顶科技股份有限公司

商标声明

GOODiX 和其他汇顶商标均为深圳市汇顶科技股份有限公司的商标。本文档提及的其他所有商标或注册商标，由各自的所有人持有。

免责声明

本文档中所述的器件应用信息及其他类似内容仅为您提供便利，它们可能由更新之信息所替代。确保应用符合技术规范，是您自身应负的责任。

深圳市汇顶科技股份有限公司（以下简称“GOODIX”）对这些信息不作任何明示或暗示、书面或口头、法定或其他形式的声明或担保，包括但不限于针对其使用情况、质量、性能、适销性或特定用途的适用性的声明或担保。GOODIX对因这些信息及使用这些信息而引起的后果不承担任何责任。

未经GOODIX书面批准，不得将GOODIX的产品用作生命维持系统中的关键组件。在GOODIX知识产权保护下，不得暗中或以其他方式转让任何许可证。

深圳市汇顶科技股份有限公司

总部地址：深圳市福田保税区腾飞工业大厦B座2层、13层

电话：+86-755-33338828　　传真：+86-755-33338099

网址：www.goodix.com

前言

编写目的

本文档介绍如何使用和修改GR551x SDK中的外设示例，旨在帮助用户快速进行二次开发。

读者对象

本文适用于以下读者：

- GR551x用户
- GR551x开发人员
- GR551x测试人员
- 开发爱好者
- 文档工程师

版本说明

本文档为第6次发布，对应的产品系列为GR551x。

修订记录

版本	日期	修订内容
1.0	2019-12-08	首次发布
1.3	2020-03-16	删除sysTick相关的工程，更新DMA的instance为channel等
1.5	2020-05-30	补充“ADC常见问题”章节处理方法中的计算公式
1.6	2020-06-30	• 更正“测试验证”章节固件名称 • 更新ADC参考电压值，涉及“ADC Battery”和“ADC Temperature”章节 • 优化示例工程流程的相关描述，涉及“HMAC”和“PKC”章节
1.7	2020-12-18	• “PWM”章节增加方式相关的描述 • “RNG”章节补充RNG_OUTPUT_FR0_S0的使用限制描述
1.8	2021-04-25	“AES”章节去掉DMA描述

目录

1 简介

本文档描述了GR551x SDK中外设示例的使用及修改方法，可帮助用户快速了解SDK外设接口的使用方法，加快开发进程。

使用和修改外设示例之前，可参考如表 1-1 文档。

表 1-1 文档参考

名称	描述
GR551x开发者指南	GR551x 软硬件介绍、快速使用及资源总览
GR551x Datasheet	GR551x芯片数据手册
J-Link用户指南	J-Link使用说明：www.segger.com/downloads/jlink/UM08001_JLink.pdf
Keil用户指南	Keil详细操作说明：www.keil.com/support/man/docs/uv4/

2 初次运行

本章介绍如何快速验证GR551x SDK中的外设示例。

说明:

SDK_Folder为GR551x SDK的根目录。

2.1 准备工作

验证外设示例之前，需要完成以下准备工作。

- 硬件准备

表 2-1 硬件准备

名称	描述
GR5515 Starter Kit开发板	部分示例需要两块GR5515 Starter Kit开发板

- 软件准备

表 2-2 软件准备

名称	描述
Windows	Windows 7/Windows 10 操作系统
J-Link Driver	J-Link驱动程序，下载网址：www.segger.com/downloads/jlink/
Keil MDK5	IDE工具，下载网址：www.keil.com/download/product/
GRUart（Windows）	GR551x串口调试工具，位于SDK_Folder\tools\GRUart

2.2 硬件连接

如图 2-1所示，使用Micro USB 2.0数据线连接GR5515 Starter Kit开发板与计算机。

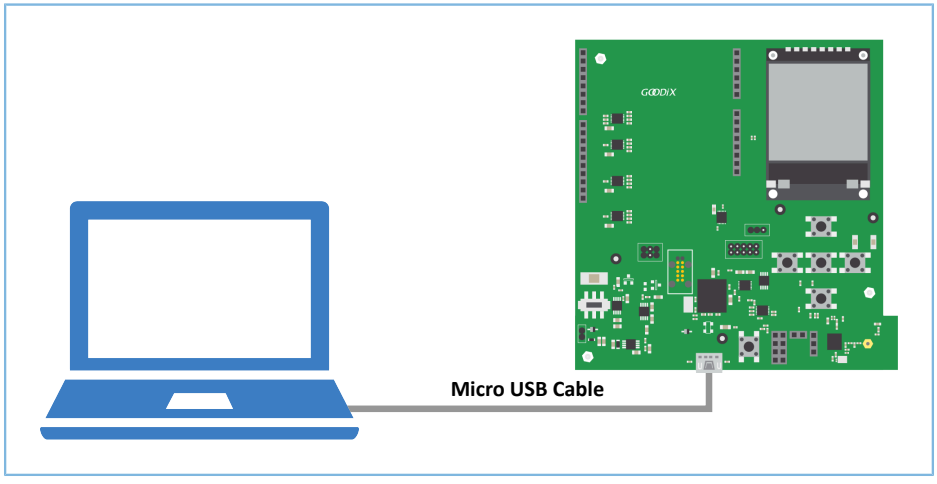

图 2-1 硬件连接示意图

2.3 下载固件

以LED外设示例为例，下载示例的*gpio_led_sk_r2_fw.bin*固件至开发板。具体操作方法请参考《GProgrammer用户手册》。

说明:

*gpio_led_sk_r2_fw.bin*位于：

`SDK_Folder\projects\peripheral\gpio\gpio_led\build\`。

2.4 串口设置

启动GRUart，按照表 2-3 中的参数配置串口。

表 2-3 GRUart串口配置参数

PortName	BaudRate	DataBits	Parity	StopBits	Flow Control
根据实际选择	115200	8	None	1	不勾选

3 应用示例概述

SDK中所有外设示例路径：`SDK_Folder\projects\peripheral\`。

以ADC示例为例，双击打开*adc.uvprojx*工程文件，在Keil中的目录结构如图 3-1所示：

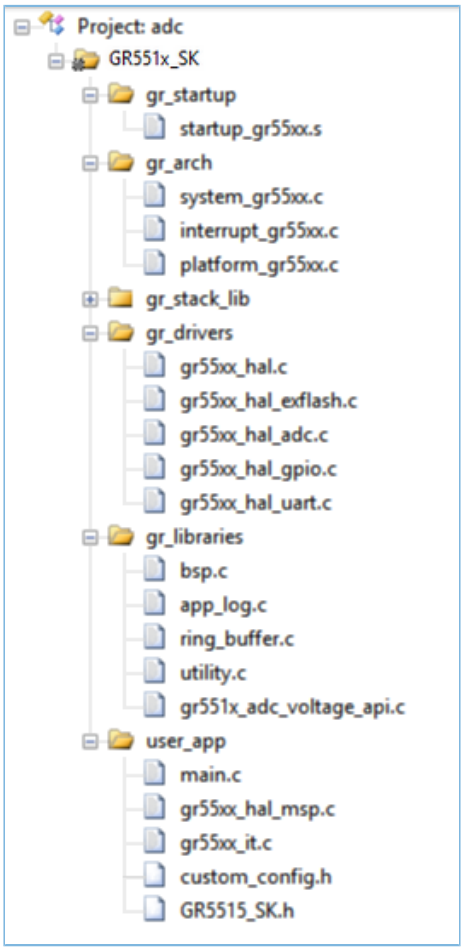

图 3-1 工程目录结构

主要的文件说明如表 3-1 所示：

表 3-1 外设示例主要文件说明

Group	文件	描述
user_app	main.c	Main()入口函数，对应模块的初始化代码和执行代码
	gr55xx_hal_msp.c	外设初始化，对应模块的GPIO、中断、DMA配置
	gr55xx_it.c	中断入口，对应模块与相关联模块的中断入口函数
	GR5515_SK.h	硬件宏定义，对应模块的常用GPIO等硬件宏定义

- gr55xx_hal_msp.c

  该文件主要实现了ADC模块中GPIO、中断等硬件的初始化与反初始化。

  hal_adc_msp_init()：GPIO、中断等硬件初始化函数。

  1. 调用NVIC_ClearPendingIRQ()与hal_nvic_enable_irq()接口清除DMA Pending中断并使能DMA中断。

2. 调用hal_msio_init()接口将ADC_P_INPUT_PIN(MSIO_PIN_0)、ADC_N_INPUT_PIN(MSIO_PIN_1)引脚初始化为模拟输入引脚。

3. 调用hal_dma_ini()接口初始化DMA模块。

代码如下所示：

```
void hal_adc_msp_init(adc_handle_t *hadc)
{
    msio_init_t msio_config = MSIO_DEFAULT_CONFIG;

    NVIC_ClearPendingIRQ(DMA_IRQn);
    hal_nvic_enable_irq(DMA_IRQn);

    /* Config input GPIO */
    msio_config.pin  = ADC_P_INPUT_PIN | ADC_N_INPUT_PIN;
    msio_config.mode = MSIO_MODE_ANALOG;
    hal_msio_init(&msio_config);

    /* Configure the DMA handler for Transmission process */
    hadc->p_dma = &s_dma_handle;
    s_dma_handle.p_parent = hadc;
    hadc->p_dma->channel           = DMA_Channel0;
    hadc->p_dma->init.src_request   = DMA_REQUEST_SNSADC;
    hadc->p_dma->init.direction     = DMA_PERIPH_TO_MEMORY;
    hadc->p_dma->init.src_increment = DMA_SRC_NO_CHANGE;
    hadc->p_dma->init.dst_increment = DMA_DST_INCREMENT;
    hadc->p_dma->init.src_data_alignment = DMA_SDATAALIGN_WORD;
    hadc->p_dma->init.dst_data_alignment = DMA_DDATAALIGN_WORD;
    hadc->p_dma->init.mode          = DMA_NORMAL;
    hadc->p_dma->init.priority      = DMA_PRIORITY_LOW;

    hal_dma_init(hadc->p_dma);
}
```

hal_adc_msp_deinit()：GPIO、中断等硬件反初始化函数。

1. 调用hal_msio_deinit()接口将ADC_P_INPUT_PIN(MSIO_PIN_0)、ADC_N_INPUT_PIN(MSIO_PIN_1)引脚恢复为普通引脚状态；

2. 调用hal_dma_deinit()反初始化DMA。

代码如下所示：

```
void hal_adc_msp_deinit(adc_handle_t *hadc)
{
    hal_msio_deinit(ADC_P_INPUT_PIN | ADC_N_INPUT_PIN);
    hal_dma_deinit(hadc->p_dma);
}
```

- gr55xx_it.c

  该文件主要实现了ADC模块与其他相关联模块的中断入口处理函数。

  DMA_IRQHandler()：DMA中断入口处理函数，运行结果与状态通过用户回调函数hal_adc_conv_cplt_callback()返回，用户在此回调函数内可自定义操作。

  代码如下所示：

```
void DMA_IRQHandler(void)
{
    hal_dma_irq_handler(g_adc_handle.p_dma);
}
__WEAK void hal_adc_conv_cplt_callback(adc_handle_t *p_adc)
{
}
```

3.1 ADC

模拟数字转换器（Analog-to-Digital Converter，即ADC）是用于将连续变化的模拟信号转换为离散的数字信号的器件。一般用于检测外设的电压值、温度等信息。

以下章节主要介绍使用ADC测量MSIO输入信号、电池电压、芯片内部温度。

3.1.1 ADC

ADC示例实现了ADC的两种采样模式：单端采样和差分采样。

- 单端采样时，用户需将有效信号连接至MSIO1引脚，单端输入信号幅值范围在0 ～ 2*Vref，且不大于VBAT；
- 差分采样时，用户需输入差分信号至MSIO0与MSIO1，差分输入信号幅值范围在-2*Vref ～ +2*Vref，共模值大于0.8 V。

ADC示例源代码和工程文件位于SDK_Folder\projects\peripheral\adc\adc，其中工程文件在文件夹Keil_5下。

3.1.1.1 代码理解

示例工程流程图如图 3-2所示：

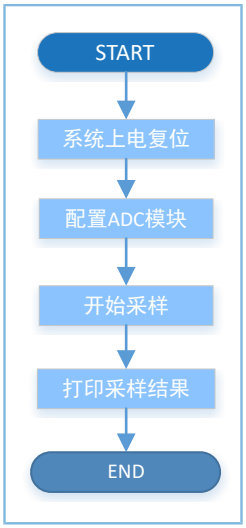

图 3-2 ADC示例工程流程图

1. 配置ADC模块。

```
g_adc_handle.init.channel_p  = ADC_INPUT_SRC_IO0;

g_adc_handle.init.channel_n  = ADC_INPUT_SRC_IO1;

g_adc_handle.init.input_mode = ADC_INPUT_SINGLE;
g_adc_handle.init.ref_source = ADC_REF_SRC_BUF_INT;
g_adc_handle.init.ref_value  = ADC_REF_VALUE_1P6;
g_adc_handle.init.clock      = ADC_CLK_1P6M;
```

- init.channel_p、init.channel_n：通道P（channel_p）和通道N（channel_n）设置，可选择ADC_INPUT_SRC_IO0 ～ ADC_INPUT_SRC_IO4、ADC_INPUT_SRC_TMP、ADC_INPUT_SRC_BAT。此处选择ADC_INPUT_SRC_IO0与ADC_INPUT_SRC_IO1，即映射至MSIO0与MSIO1脚。用户可修改映射至其他MSIO引脚。
- init.input_mode：输入模式设置，可选择ADC_INPUT_SINGLE、ADC_INPUT_DIFFERENTIAL。此处选择ADC_INPUT_SINGLE即单端模式，表示只采样N通道。用户可修改为差分模式。
- init.ref_source：参考源设置，可选择ADC_REF_SRC_BUF_INT、ADC_REF_SRC_IO0 ～ ADC_REF_SRC_IO3。此处选择ADC_REF_SRC_BUF_INT即内部参考源。
- init.ref_value：内部参考源参考电压设置，可选择ADC_REF_VALUE_0P8、ADC_REF_VALUE_1P2、ADC_REF_VALUE_1P6。此处选择ADC_REF_VALUE_1P6即1.6 V，输入量程为0 ～ 3.2 V。
- init.clock：ADC时钟设置，可选择ADC_CLK_16M、ADC_CLK_1P6M、ADC_CLK_8M、ADC_CLK_4M、ADC_CLK_2M、ADC_CLK_1M。此处选择ADC_CLK_1P6M 即1.6 MHz，用户可修改为其他频率。

2. 本示例中ADC转换采用DMA方式读取数据。

(1) 调用hal_adc_start_dma()接口用DMA方式进行ADC采样。

采样结果通过DMA读取至内存中，代码如下所示：

```
hal_adc_start_dma(&g_adc_handle, conversion, TEST_CONV_LENGTH);
```

由于为非阻塞方式，需等待ADC状态恢复。在本示例中，ADC采样结果通过hal_adc_conv_cplt_callback()接口返回，在该回调函数内用户可自定义可执行操作。

代码如下所示：

```
void hal_adc_conv_cplt_callback(adc_handle_t *hadc)
{
    printf("DMA conversion is done.\r\n");
}
```

(2) 调用hal_gr551x_adc_voltage_intern()接口将得到的ADC数值转换为电压值，代码如下所示：

```
hal_gr551x_adc_voltage_intern(&g_adc_handle, conversion, voltage,
                              TEST_CONV_LENGTH);
```

(3) 如果参考源采用外部参考时，可调用如下接口将得到ADC的数值转换为电压值：

```
void hal_gr551x_adc_voltage_extern(adc_handle_t *hadc, double vref, uint16_t
                                   *inbuf, double *outbuf, uint32_t buflen)
```

3.1.1.2 测试验证

1. 用GProgrammer下载*adc_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并设置GRUart。
3. 串口输出ADC等模块的开机信息等日志。
4. 在GRUart的“Receive Data”窗口中显示计算后的ADC电压值。

3.1.2 ADC DMA UART

ADC DMA UART示例实现了使用DMA方式将ADC采样数据通过UART直接传输至PC或其他外围设备。数据传输无需CPU参与，有助于高速数据传输。

ADC DMA UART示例的源代码和工程文件位于SDK_Folder\projects\peripheral\adc\adc_dma_uart，其中工程文件在文件夹Keil_5下。

3.1.2.1 代码理解

示例工程流程图如图 3-3所示：

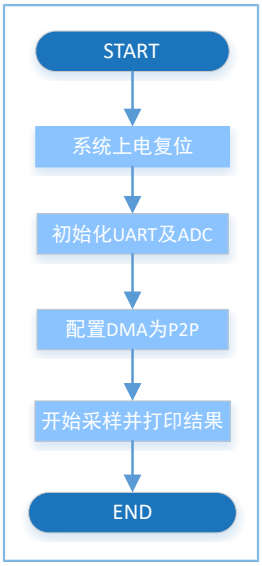

图 3-3 ADC DMA UART示例工程流程图

1. 配置ADC模块。

```
g_adc_handle.init.channel_p  = ADC_INPUT_SRC_IO0;
g_adc_handle.init.channel_n  = ADC_INPUT_SRC_IO1;
g_adc_handle.init.input_mode = ADC_INPUT_SINGLE;
g_adc_handle.init.ref_source = ADC_REF_SRC_BUF_INT;
g_adc_handle.init.ref_value  = ADC_REF_VALUE_1P6;
g_adc_handle.init.clock = ADC_CLK_1P6M;
```

ADC配置参数详情请参考3.1.1 ADC。

2. UART发送阈值及DMA传输单元大小配置。

```
ll_adc_set_thresh(16);
ll_uart_set_tx_fifo_threshold(uart_handle.p_instance, LL_UART_TX_FIFO_TH_CHAR_2);
ll_dma_set_source_burst_length(DMA, hadc->p_dma->channel,
                               LL_DMA_SRC_BURST_LENGTH_8);
ll_dma_set_destination_burst_length(DMA, hadc->p_dma->channel,
                               LL_DMA_DST_BURST_LENGTH_8);
```

- ADC阈值配置为16，即ADC采样到16个数据时请求DMA传输。
- UART发送阈值配置为2，即UART TX FIFO中的数据少于2个时请求DMA传输。
- DMA burst长度配置为8，即一次传输8个数据，用于减少DMA的读取次数，提高DMA效率。

3. 数据采样。

```
__HAL_ADC_ENABLE_CLOCK(hadc);
hal_dma_start(hadc->p_dma, (uint32_t)&MCU_SUB->SENSE_ADC_FIFO,
              (uint32_t)&UART1->RBR_DLL_THR, TEST_CONV_LENGTH >> 1);
hal_dma_poll_for_transfer(hadc->p_dma, 1000);
__HAL_ADC_DISABLE_CLOCK(hadc);
```

- 打开ADC时钟。

- 调用hal_dma_start()开始DMA传输。
- hal_dma_poll_for_transfer()以polling方式等待DMA传输完成。结果将通过UART1传输至GRUart。

3.1.2.2 测试验证

1. 用GProgrammer下载*adc_dma_uart_sk_r2_fw.bin*至开发板。
2. 将开发板串口UART0，UART1连接至PC端，打开并设置两个GRUart。
3. 连接UART0的GRUart打印程序的调试信息，连接UART1的GRUart打印ADC采样值的16进制数据。

3.1.3 ADC Battery

ADC Battery示例实现了ADC对电池电压的测量。

ADC Battery示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\adc\battery`，其中工程文件在文件夹Keil_5下。

3.1.3.1 代码理解

示例工程流程图如图 3-4所示：

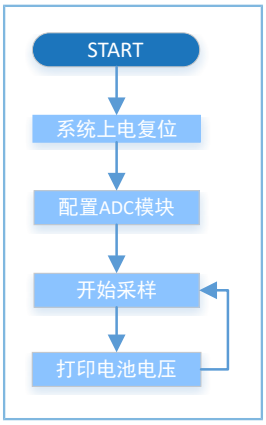

图 3-4 ADC Battery示例工程流程图

1. 配置ADC模块。

```
g_adc_handle.init.channel_p = ADC_INPUT_SRC_BAT;
g_adc_handle.init.channel_n = ADC_INPUT_SRC_BAT;
g_adc_handle.init.input_mode = ADC_INPUT_SINGLE;
g_adc_handle.init.ref_source = ADC_REF_SRC_BUF_INT;
g_adc_handle.init.ref_value = ADC_REF_VALUE_0P8;
g_adc_handle.init.clock = ADC_CLK_1P6M;
```

ADC配置参数细节请参考3.1.1 ADC。此示例中init.channel_p、init.channel_n选择ADC_INPUT_SRC_BAT。init.ref_value选择ADC_REF_VALUE_0P8即0.85 V。

说明:

可检测的VBAT范围在2.0 V ～ 3.8 V，当选择ADC的channel通道为BAT，测量的信号来自内部的分压电路（确保输入在0 ～ 1.6 V），所以参考电压应选择0.85 V，以提升测量精度。

2. 调用hal_gr551x_vbat_init()接口初始化ADC BATTERY模块。
3. 调用hal_gr551x_vbat_read()接口读取电池电压。接口返回值即为电池当前电压，单位伏（V）。

3.1.3.2 测试验证

1. 用GProgrammer下载*battery_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示转化后的电池电压值。

3.1.4 ADC Temperature

ADC Temperature示例实现了ADC对芯片内部温度的测量。

ADC Temperature示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\adc\temperature`，其中工程文件在文件夹Keil_5下。

3.1.4.1 代码理解

示例工程流程图如图 3-5所示：

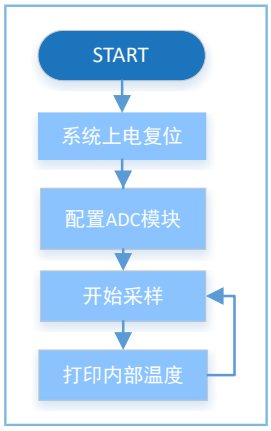

图 3-5 ADC Temperature示例工程流程图

1. 配置ADC模块。

```
g_adc_handle.init.channel_p = ADC_INPUT_SRC_TMP;
g_adc_handle.init.channel_n = ADC_INPUT_SRC_TMP;
g_adc_handle.init.input_mode = ADC_INPUT_SINGLE;
g_adc_handle.init.ref_source = ADC_REF_SRC_BUF_INT;
g_adc_handle.init.ref_value = ADC_REF_VALUE_0P8;
```

```
g_adc_handle.init.clock = ADC_CLK_1P6M;
```

ADC配置参数细节请参考3.1.1 ADC。此示例中init.channel_p、init.channel_n选择ADC_INPUT_SRC_TMP。init.ref_value选择ADC_REF_VALUE_0P8即0.85 V。

2. 调用hal_gr551x_temp_init()接口初始化ADC TEMPERATURE模块。
3. 调用hal_gr551x_temp_read()接口读取温度值。接口返回值即为芯片内部温度，单位摄氏度（℃）。

3.1.4.2 测试验证

1. 用GProgrammer下载*temp_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示转化后的温度值。

3.2 AES

高级加密标准（Advanced Encryption Standard，即AES），又称Rijndael加密法，是一种区块加密标准，经过多年的发展已成为对称密钥加密中最流行的算法之一。一般在数据加解密和文件加解密过程中使用。

3.2.1 AES

AES示例实现了AES的两种加密模式：ECB和CBC，以及三种加密密钥长度：128 bits，192 bits和256 bits。

在示例工程中明文和密文为直接输出，同一条明文应用不同长度的密钥加密可得到对应不同的密文。

- 加密操作时，将明文作为输入，AES加密输出密文数据，并将其与相应原始密文作对比，验证AES加密的正确性。
- 解密操作时，将加密输出的密文作为输入，AES解密输出明文数据，将其与原始明文作对比，验证AES解密的正确性。

以下章节主要介绍使用AES的轮询和中断两种方式对数据进行加解密。

AES示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\aes\aes`，其中工程文件在文件夹Keil_5下。

3.2.1.1 代码理解

示例工程流程图如图 3-6所示：

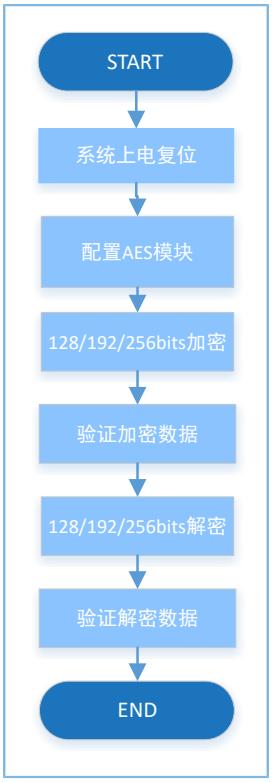

图 3-6 AES示例工程流程图

1. 配置AES模块。

   AES模块配置代码（ECB）如下：

```
g_aes_handle.p_instance = AES;
g_aes_handle.init.key_size = AES_KEYSIZE_128BITS;
g_aes_handle.init.p_key = (uint32_t *)g_key128_ecb;
g_aes_handle.init.chaining_mode = AES_CHAININGMODE_ECB;
g_aes_handle.init.p_init_vector = NULL;
g_aes_handle.init.dpa_mode = DISABLE;
g_aes_handle.init.p_seed = (uint32_t *)g_seed;

hal_aes_deinit(&g_aes_handle);
hal_aes_init(&g_aes_handle);
```

   AES模块配置代码（CBC）如下：

```
g_aes_handle.p_instance         = AES;
g_aes_handle.init.key_size      = AES_KEYSIZE_128BITS;
g_aes_handle.init.p_key         = (uint32_t *)g_key128_cbc;
g_aes_handle.init.chaining_mode = AES_CHAININGMODE_CBC;
g_aes_handle.init.p_init_vector = (uint32_t *)g_iv_cbc;
g_aes_handle.init.dpa_mode = DISABLE;
g_aes_handle.init.p_seed = (uint32_t *)g_seed;

hal_aes_deinit(&g_aes_handle);
hal_aes_init(&g_aes_handle);
```

- init.key_size：密钥长度设置，可选择AES_KEYSIZE_128BITS、AES_KEYSIZE_192BITS、AES_KEYSIZE_256BITS。此示例选择AES_KEYSIZE_128BITS即128位。
- init.p_key：加密/解密密钥。由用户提供。
- init.chaining_mode：加解密方式设置，可选择AES_CHAININGMODE_ECB、AES_CHAININGMODE_CBC。
- init.p_init_vector：CBC模式中初始化向量，ECB模式中不用配置。
- dpa_mode：配置是否使能DPA功能。
- p_seed：随机序列种子，由用户提供。

2. 使用AES加解密API。

   以ECB模式为例，CBC模式的加解密可参考ECB加解密接口使用。

   - 轮询方式加解密

```
hal_aes_ecb_encrypt(&g_aes_handle, (uint32_t *)g_plaintext_ecb,
    sizeof(g_plaintext_ecb), (uint32_t *)g_encrypt_result, 5000)
hal_aes_ecb_decrypt(&g_aes_handle, (uint32_t *)g_encrypt_result,
    sizeof(g_encrypt_result), (uint32_t *)g_decrypt_result, 5000)
```

   (1) 调用hal_aes_ecb_encrypt()接口实现明文的轮询方式加密，加密完成后返回函数执行结果，密文数据在参数g_encrypt_result中。

   (2) 调用hal_aes_ecb_decrypt()接口实现密文的轮询方式解密，解密完成后返回函数执行结果，明文数据在参数g_decrypt_result中。

   - 非轮询方式加解密

```
hal_aes_ecb_encrypt_it(&g_aes_handle, (uint32_t *)g_plaintext_ecb,
    sizeof(g_plaintext_ecb), (uint32_t *)g_encrypt_result)
hal_aes_ecb_decrypt_it(&g_aes_handle, (uint32_t *)g_encrypt_result,
    sizeof(g_encrypt_result), (uint32_t *)g_decrypt_result)

void hal_aes_done_callback(aes_handle_t *haes)
{
    g_int_done_flag = 1;
}

void hal_aes_error_callback(aes_handle_t *haes)
{
    printf("\r\nGet an Error!\r\n");
}
```

   (1) 调用hal_aes_ecb_encrypt_it()接口实现明文的非轮询方式加密，加密状态即结果通过回调函数hal_aes_done_callback()和hal_aes_error_callback()返回，用户在此函数内可以自定义操作。

(2) 调用hal_aes_ecb_decrypt_it()接口实现密文的非轮询方式解密，解密状态即结果通过回调函数hal_aes_done_callback()和hal_aes_error_callback()返回，用户在此函数内可以自定义操作。

3.2.1.2 测试验证

1. 用GProgrammer下载*aes_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示AES加解密结果。

3.3 AON_GPIO

AON_GPIO通用型输入输出端口，该模块是基于低速时钟工作的IO模块，可用于输入输出或其他特殊功能（例如唤醒睡眠态下的MCU，触发中断）。

以下章节主要介绍使用AON_GPIO的输入输出和中断唤醒功能。

3.3.1 AON_GPIO Input & Output

AON_GPIO Input & Output示例实现了AON_GPIO的输入输出功能。

- 配置为普通输出时，通过程序AON_GPIO Input & Output示例控制IO上电平的变化。
- 配置为普通输入时，通过程序AON_GPIO Input & Output示例读取IO上电平的高低。

AON_GPIO Input & Output示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\aon_gpio\aon_gpio_output_input`，其中工程文件在文件夹Keil_5下。

3.3.1.1 代码理解

示例工程流程图如图 3-7所示：

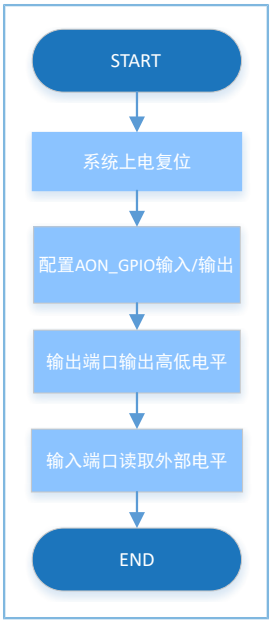

图 3-7 AON_GPIO Input & Output示例工程流程图

1. 配置AON_GPIO模块输入/输出模式。

```
#define AON_GPIO_DEFAULT_CONFIG                         \
{                                                       \
    .pin     = AON_GPIO_PIN_ALL,                  \
    .mode    = AON_GPIO_MODE_INPUT,               \
    .pull    = AON_GPIO_PULLDOWN,                 \
    .mux     = AON_GPIO_MUX_7,                    \
}

aon_gpio_init_t aon_gpio_init = AON_GPIO_DEFAULT_CONFIG;

aon_gpio_init.pin  = AON_GPIO_PIN_7;
aon_gpio_init.mode = AON_GPIO_MODE_OUTPUT;
hal_aon_gpio_init(&aon_gpio_init);

aon_gpio_init.pin  = AON_GPIO_PIN_6;
aon_gpio_init.mode = AON_GPIO_MODE_INPUT;
hal_aon_gpio_init(&aon_gpio_init);
```

- init.pin：引脚配置ID，可选择AON_GPIO_PIN_0 ～ AON_GPIO_PIN_7任意组合。
- init.mode：引脚工作模式，可选择AON_GPIO_MODE_INPUT、AON_GPIO_MODE_OUTPUT、AON_GPIO_MODE_MUX、AON_GPIO_MODE_IT_RISING、AON_GPIO_MODE_IT_FALLING、AON_GPIO_MODE_IT_HIGH、AON_GPIO_MODE_IT_LOW。
- init.pull：上下拉电阻配置，可选择AON_GPIO_NOPULL、AON_GPIO_PULLUP、AON_GPIO_PULLDOWN。

- init.mux：pin_mux配置，参考《GR551x Datasheet》中的pin_mux配置表，输入/输出时需配置为AON_GPIO_MUX_7。

当需要配置为输入时，只需要修改mode：

```
aon_gpio_init.mode = AON_GPIO_MODE_INPUT;
```

2. 设置输出引脚电平。

```
hal_aon_gpio_write_pin(AON_GPIO_PIN_7, AON_GPIO_PIN_RESET);
```

参数AON_GPIO_PIN_RESET表示设置为低电平，AON_GPIO_PIN_SET表示设置为高电平。

3. 读取输入引脚电平。

```
pin_level = hal_aon_gpio_read_pin(AON_GPIO_PIN_6);
```

pin_level为0表示低电平，为1表示高电平。

3.3.1.2 测试验证

1. 用GProgrammer下载*aon_gpio_io_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示IO口的输入/输出状态信息。

3.3.2 AON_GPIO Wakeup

AON_GPIO Wakeup示例实现了AON_GPIO的中断检测功能。

当AON_GPIO配置为外部输入中断模式并使能中断时，通过IO上电平的变化（上升/下降沿，高/低电平）触发中断处理程序；AON_GPIO也可以配置为唤醒源，用于唤醒睡眠的系统。

AON_GPIO Wakeup示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\aon_gpio\aon_gpio_wakeup`，其中工程文件在文件夹Keil_5下。

3.3.2.1 代码理解

示例工程流程图如图 3-8和图 3-9所示：

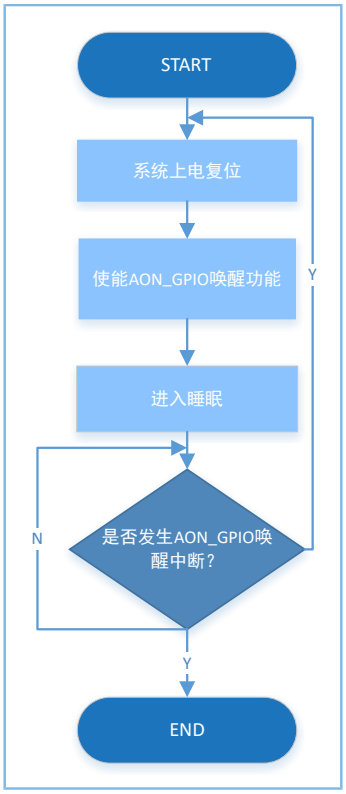

图 3-8 AON_GPIO Wakeup示例工程流程图（睡眠）

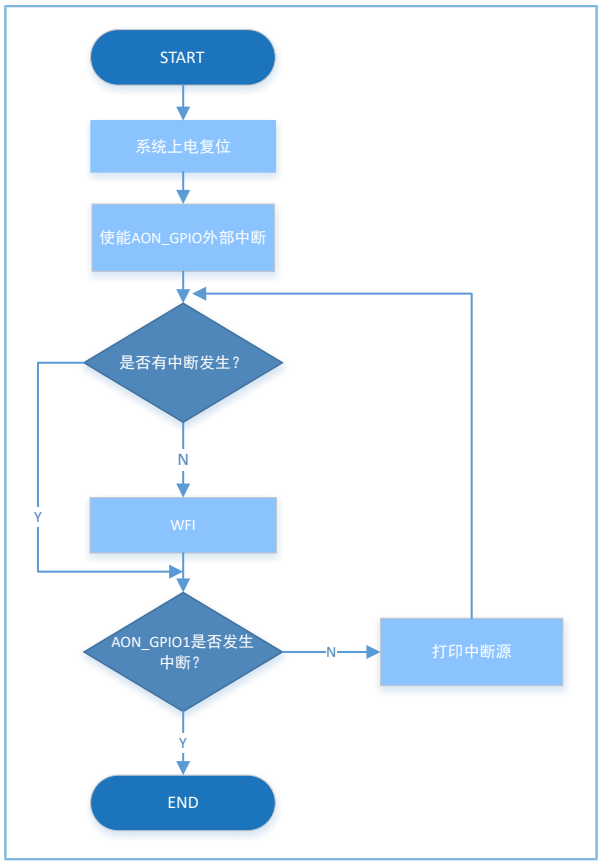

图 3-9 AON_GPIO Wakeup示例工程流程图（WFI/WFE）

1. 为AON_GPIO模块配置外部中断。

```
aon_gpio_init_t aon_gpio_init = AON_GPIO_DEFAULT_CONFIG;
aon_gpio_init.pin  = KEY_OK_PIN;
aon_gpio_init.mode = KEY_ANO_TRIGGER_MODE;
aon_gpio_config.pull = AON_GPIO_PULLUP;
hal_aon_gpio_init(&aon_gpio_init);
```

AON_GPIO配置参数细节请参考3.3.1 AON_GPIO Input & Output。此示例中pin选择KEY_OK_PIN即AON_GPIO_PIN_1。

2. 清除并使能AON_GPIO中断。代码如下所示：

```
hal_nvic_clear_pending_irq(EXT2_IRQn);
hal_nvic_enable_irq(EXT2_IRQn);
```

3. 系统进入睡眠。代码如下所示：

```
while (!g_exit_flag)
{
    printf("\r\nEnter sleep.\r\n");
    SCB->SCR |= 0x04;
    __WFI();
    printf("Wakeup from sleep.\r\n");
}
```

4. 按“OK”键唤醒系统并结束程序。

3.3.2.2 测试验证

1. 用GProgrammer下载*aon_gpio_wakeup_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示通过IO口唤醒系统以及设备睡眠的调试信息。

3.4 GPIO

通用型输入输出端口，该模块是基于系统时钟工作的IO模块，可用于输入/输出或其他特殊功能（例如：唤醒睡眠态下的MCU，触发中断）。一般使用GPIO控制外围设备或者用中断方式检测外围设备的输入状态。

以下章节主要介绍使用GPIO的输入/输出和中断唤醒功能。

3.4.1 GPIO Input & Output

GPIO Input & Output示例实现了GPIO的普通输入/输出功能。

- 配置为普通输出时，通过程序能控制IO上电平的变化。
- 配置为普通输入时，能通过程序读取IO上电平的高低。

GPIO Input & Output示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\gpio\gpio_output_input`，其中工程文件在文件夹Keil_5下。

3.4.1.1 代码理解

示例工程流程图如图 3-10所示：

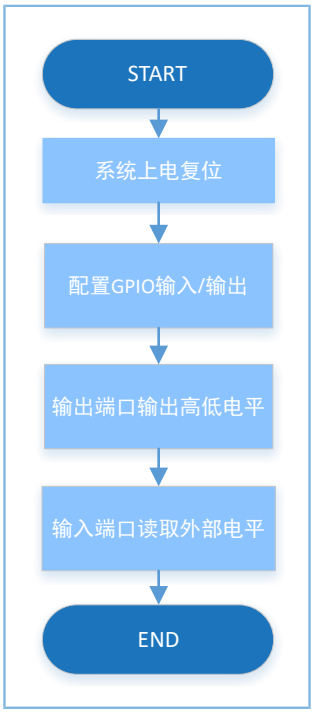

图 3-10 GPIO Input & Output示例工程流程图

1. 配GPIO模块输入/输出模式。

```
#define GPIO_DEFAULT_CONFIG                         \
{                                                   \
    .pin        = GPIO_PIN_ALL,                     \
    .mode       = GPIO_MODE_INPUT,                  \
    .pull       = GPIO_PULLDOWN,                   \
    .mux        = GPIO_PIN_MUX_GPIO,                \
}

gpio_init_t gpio_config = GPIO_DEFAULT_CONFIG;
gpio_config.mode = GPIO_MODE_OUTPUT;
gpio_config.pin  = GPIO_PIN_12;
hal_gpio_init(GPIO1, &gpio_config);
```

- pin：引脚配置ID，可选择GPIO_PIN_0 ～ GPIO_PIN_15任意组合。
- mode：引脚工作模式，可选择GPIO_MODE_INPUT、GPIO_MODE_OUTPUT、GPIO_MODE_MUX、GPIO_MODE_IT_RISING、GPIO_MODE_IT_FALLING、GPIO_MODE_IT_HIGH、GPIO_MODE_IT_LOW。
- pull：上下拉电阻配置，可选择GPIO_NOPULL、GPIO_PULLUP、GPIO_PULLDOWN。
- mux：pin_mux配置，参考《GR551x Datasheet》中的pin_mux配置表，输入/输出时需配置为GPIO_MUX_7。

当需要配置为输入时，只需要修改mode：

```
gpio_init.mode = GPIO_MODE_INPUT;
```

2. 设置输出引脚电平。

```
hal_gpio_write_pin(GPIO1, GPIO_PIN_12, GPIO_PIN_RESET);
```

参数GPIO_PIN_RESET表示设置为低电平，GPIO_PIN_SET表示设置为高电平

3. 读取输入引脚电平。

```
pin_level = hal_gpio_read_pin(GPIO1, GPIO_PIN_13);
```

pin_level为0表示低电平，为1表示高电平。

3.4.1.2 测试验证

1. 用GProgrammer下载*gpio_io_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示IO口的输入/输出状态信息。

3.4.2 GPIO Interrupt

GPIO Interrupt示例实现了GPIO的中断输入功能。GPIO配置为中断输入时，可以配置为上升/下降沿触发、高/低电平触发。

GPIO Interrupt示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\gpio\gpio_interrupt`，其中工程文件在文件夹Keil_5下。

3.4.2.1 代码理解

示例工程流程图如图 3-11所示：

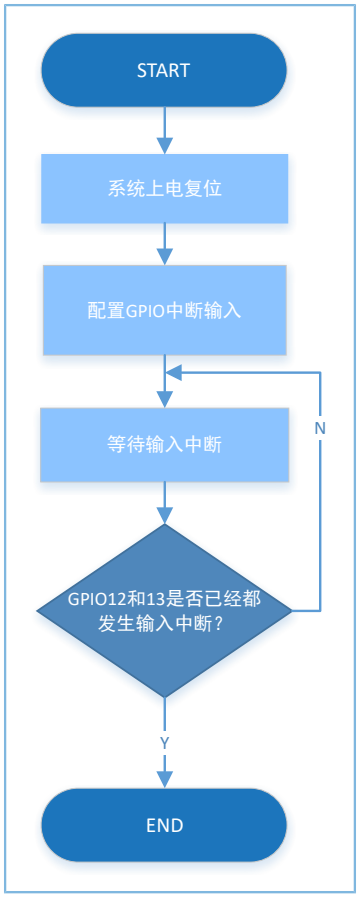

图 3-11 GPIO Interrupt示例工程流程图

1. 配置GPIO模块中断输入。

```
gpio_init_t gpio_config = GPIO_DEFAULT_CONFIG;
gpio_config.mode = GPIO_MODE_IT_FALLING;
gpio_config.pin  = GPIO_KEY0 | GPIO_KEY1;
gpio_config.pull = GPIO_PULLUP;
hal_gpio_init(GPIO_KEY_PORT, &gpio_config);

/* Enable interrupt */
hal_nvic_clear_pending_irq(GPIO_GET_IRQNUM(GPIO_KEY_PORT));
hal_nvic_enable_irq(GPIO_GET_IRQNUM(GPIO_KEY_PORT));
```

GPIO配置参数细节请参考3.4.1 GPIO Input & Output，其中mode选择GPIO_MODE_IT_FALLING，pin选择GPIO_KEY0、GPIO_KEY1即GPIO_PIN_12、GPIO_PIN_13。

2. 调用hal_nvic_clear_pending_irq()和hal_nvic_enable_irq()清除并使能中断。代码如下所示：

```
hal_nvic_clear_pending_irq(GPIO_GET_IRQNUM(GPIO_KEY_PORT));
hal_nvic_enable_irq(GPIO_GET_IRQNUM(GPIO_KEY_PORT));
```

3. 中断状态通过回调函数hal_gpio_exti_callback()返回，用户在此函数内可自定义操作。

3.4.2.2 测试验证

1. 用GProgrammer下载*gpio_interrupt_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示GPIO的中断输入结果。

3.4.3 GPIO LED

GPIO LED示例实现了GPIO的输出功能。GPIO配置为输出驱动LED。

GPIO LED示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\gpio\gpio_led`，其中工程文件在文件夹Keil_5下。

3.4.3.1 代码理解

1. 配置GPIO模块输出。

```
gpio_init_t gpio_config = GPIO_DEFAULT_CONFIG;
gpio_config.mode = GPIO_MODE_OUTPUT;
gpio_config.pin  = LED2_PIN;
hal_gpio_init(LED2_PORT, &gpio_config);
```

   - GPIO配置参数细节请参考3.4.1 GPIO Input & Output。其中pin 配置为LED2_PIN即GPIO_PIN_4。
   - mux为pin_mux配置，参考《GR551x Datasheet》中的pin_mux配置表，输入/输出时需配置为GPIO_MUX_7。

2. 调用宏定义LED2_TOG()即hal_gpio_toggle_pin()实现GPIO口的电平输出翻转。代码如下：

```
#define LED2_TOG()          hal_gpio_toggle_pin(LED2_PORT, LED2_PIN)
```

3.4.3.2 测试验证

1. 用GProgrammer下载*gpio_led_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示GPIO驱动LED的结果以及开发板上LED灯的亮/灭状态。

3.4.4 GPIO Wakeup

GPIO Wakeup示例实现了GPIO的外部输入中断将系统从WFI/WFE中唤醒。

当GPIO配置为外部输入中断模式并使能中断时，通过IO上电平的变化（上升/下降沿，高/低电平）触发中断处理程序。

GPIO Wakeup示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\gpio\gpio_wakeup`，其中工程文件在文件夹Keil_5下。

3.4.4.1 代码理解

示例工程流程图如图 3-12所示：

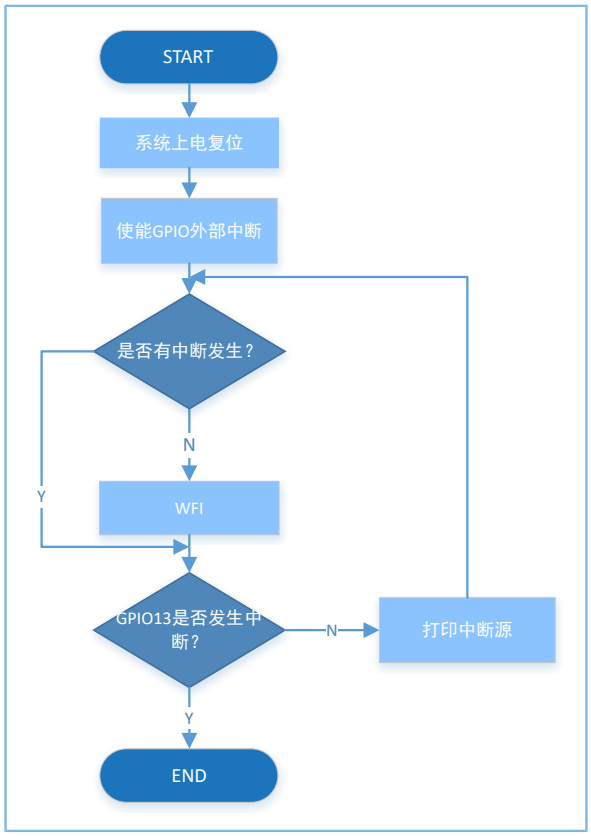

图 3-12 GPIO Wakeup示例工程流程图（WFI/WFE）

1. 配置GPIO模块外部中断。

```
gpio_init_t gpio_config = GPIO_DEFAULT_CONFIG;
gpio_config.mode = GPIO_MODE_IT_FALLING;
gpio_config.pin  = GPIO_KEY0 | GPIO_KEY1;
gpio_config.pull = GPIO_PULLUP;
hal_gpio_init(GPIO_KEY_PORT, &gpio_config);
```

- GPIO配置参数细节请参考3.4.1 GPIO Input & Output。其中mode配置为GPIO_MODE_IT_FALLING，pin配置为GPIO_KEY0和GPIO_KEY1即GPIO_PIN_12和GPIO_PIN_13。
- mux为pin_mux配置，参考《GR551x Datasheet》中的pin_mux配置表，外部中断输入时需配置为GPIO_MUX_7。

2. 调用hal_nvic_clear_pending_irq()和hal_nvic_enable_irq()清除并使能中断。代码如下所示：

```
hal_nvic_clear_pending_irq(GPIO_GET_IRQNUM(GPIO_KEY_PORT));
hal_nvic_enable_irq(GPIO_GET_IRQNUM(GPIO_KEY_PORT));
```

3. 系统进入睡眠模式。代码如下：

```
while (!g_exit_flag)
{
    printf("\r\nEnter sleep at counter = %d\r\n", sleep_count);
```

```
    SCB->SCR |= 0x04;
    __WFI();
    printf("Wakeup from sleep at counter = %d\r\n", sleep_count++);
}
```

4. 按GPIO_KEY0（对应开发板“UP”键）唤醒系统，按GPIO_KEY1（对应开发板“Down”键）唤醒系统并退出示例工程程序。

3.4.4.2 测试验证

1. 用GProgrammer下载*gpio_wakeup_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示GPIO将CPU从WFI/WFE模式中唤醒的调试信息。

3.5 HMAC

哈希运算消息认证码（Hash-based Message Authentication Code，即HMAC）是一种使用单向散列函数构造消息认证码的方法。一般应用于消息摘要或者签名。

3.5.1 HMAC

HMAC示例实现了HMAC的两种生成摘要模式：SHA-256和HMAC-SHA256。

- SHA-256对输入消息产生长度为256 bits的哈希值，称为消息摘要，输入信息需要用户按照相关HMAC驱动API规定进行填充。
- HMAC-SHA256对输入的密钥和消息运用哈希算法生成一段消息摘要，密钥长度为256 bits。

HMAC示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\hmac\hmac`，其中工程文件在文件夹Keil_5下。

以下章节主要介绍HMAC消息认证的Interrupt、Query、DMA三种实现方式。

3.5.1.1 代码理解

示例工程流程图如图 3-13和图 3-14所示：

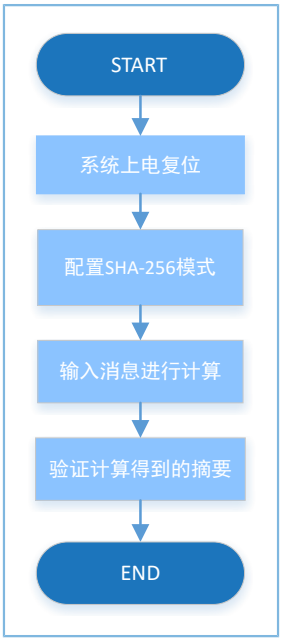

图 3-13 HMAC示例工程流程图（SHA-256）

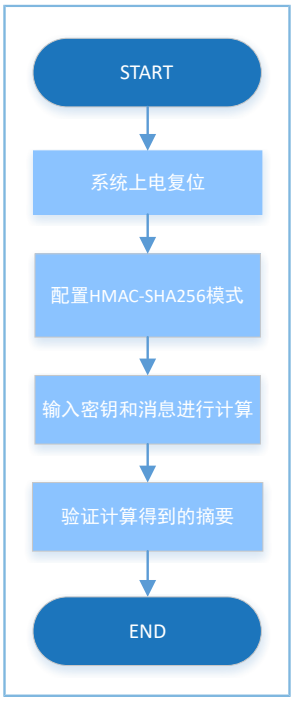

图 3-14 HMAC示例工程流程图（HMAC-SHA256）

1. 配置HMAC模块。

```
g_hmac_handle.p_instance       = HMAC;
g_hmac_handle.init.mode        = HMAC_MODE_HMAC;
g_hmac_handle.init.p_key       = (uint32_t *)hmac_key;
g_hmac_handle.init.p_user_hash = NULL;
g_hmac_handle.init.dpa_mode    = DISABLE;
g_hmac_handle.init.key_fetch_type = HAL_HMAC_KEYTYPE_MCU;
g_hmac_handle.init.enable_irq = HAL_HMAC_DISABLE_IRQ;
g_hmac_handle.init.enable_dma_mode = HAL_HMAC_DISABLE_DMA;
```

- init.mode：HMAC操作模式选择，可选择HMAC_MODE_SHA（SHA-256模式）、HMAC_MODE_HMAC（HMAC-SHA256模式）。
- init.p_key：密钥，只在HMAC-SHA256计算模式下有效。
- init.p_user_hash：用户哈希初始值，可根据需要自定义哈希初始值。
- init.dpa_mode：配置是否使能DPA功能。
- init.key_fetch_type：Key的来源，可选择HAL_HMAC_KEYTYPE_MCU、HAL_HMAC_KEYTYPE_AHB、HAL_HMAC_KEYTYPE_KRAM。
- init.enable_irq：配置是否使能IRQ中断方式，可选择HAL_HMAC_ENABLE_IRQ、HAL_HMAC_DISABLE_IRQ。
- init.enable_dma_mode：配置是否使能DMA模式，可选择HAL_HMAC_ENABLE_DMA、HAL_HMAC_DISABLE_DMA。

2. 用户可按需选择轮询、中断或DMA计算模式。各计算模式的配置方式如下：

   (1) 可根据需要重载自定义哈希初始值p_user_hash和密钥p_key（仅在HMAC-SHA256计算模式生效）。

   (2) 按需配置计算模式相关参数。

   - init.enable_irq：HAL_HMAC_DISABLE_IRQ（轮询模式、DMA模式）、HAL_HMAC_ENABLE_IRQ（中断模式）
   - init.enable_dma_mode：HAL_HMAC_DISABLE_DMA（轮询模式、中断模式）、HAL_HMAC_ENABLE_DMA（DMA模式）
   - init.mode：HMAC_MODE_SHA或者HMAC_MODE_HMAC

   (3) 使用hal_hmac_sha256_digest()计算消息摘要。若数据流太长无法一次性计算完成，则需将数据分段处理。对于非首段数据的处理，需将自定义的哈希初始值p_user_hash重载为上一次的计算结果。

   对于中断及DMA计算模式，当计算完成时hal_hmac_done_callback()会被调用；计算出错时hal_hmac_error_callback()会被调用，用户可重写该接口。

3.5.1.2 测试验证

1. 用GProgrammer下载*hmac_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示HMAC计算结果。

3.6 I2C

I2C是一种简单、双向二线制同步串行总线。它只需要两根线即可在连接于总线上的器件之间传输信息。一般用于获取传感器的数据，例如：温度传感器、加速度传感器等。

以下章节主要介绍使用I2C接口以Interrupt、Query、DMA三种方式实现数据的读写和主从接口的使用方法。

3.6.1 I2C DMA UART

I2C DMA UART示例实现了I2C如何通过DMA方式发送从UART中接收的数据。

I2C DMA UART示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\i2c\i2c_dma_uart`，其中工程文件在文件夹Keil_5下。

3.6.1.1 代码理解

示例工程流程图如图 3-15所示：

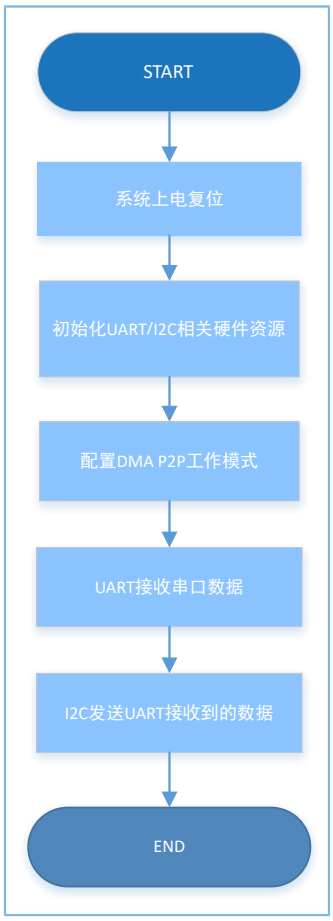

图 3-15 I2C DMA UART示例工程流程图

1. 配置I2C模块。

```
p_i2c_handle.p_instance             = I2C_MODULE;
p_i2c_handle.init.speed             = I2C_SPEED_400K;
p_i2c_handle.init.own_address       = 0x55;
```

```
p_i2c_handle.init.addressing_mode   = I2C_ADDRESSINGMODE_7BIT;
p_i2c_handle.init.general_call_mode = I2C_GENERALCALL_DISABLE;

hal_i2c_deinit(&p_i2c_handle);
hal_i2c_init(&p_i2c_handle);
```

- init.speed：I2C传输速度选择，可选择I2C_SPEED_100K、I2C_SPEED_400K、I2C_SPEED_1000K、I2C_SPEED_2000K。
- init.own_address：I2C本地地址设置，7位或者10位地址，用户可自定义。
- init.addressing_mode：I2C地址模式选择，可选择I2C_ADDRESSINGMODE_7BIT、I2C_ADDRESSINGMODE_10BIT。
- init.general_call_mode：I2C通用呼叫寻址模式选择，可选择I2C_GENERALCALL_DISABLE、I2C_GENERALCALL_ENABLE。

2. I2C硬件初始化。

```
void hal_i2c_msp_init(i2c_handle_t * p_i2c)
{
    gpio_init_t gpio_config = GPIO_DEFAULT_CONFIG;

    gpio_config.mode = GPIO_MODE_MUX;
    gpio_config.pull = GPIO_PULLUP;
    gpio_config.pin  = I2C_SCL_PIN | I2C_SDA_PIN;
    gpio_config.mux  = I2C_GPIO_MUX;
    hal_gpio_init(I2C_GPIO_PORT, &gpio_config);                    (1)

    __HAL_LINKDMA(p_i2c, p_dmatx, s_dma_handle);
    /* Configure the DMA handler for Transmission process */
    hi2c->p_dmatx->channel                 = DMA_Channel0;
    hi2c->p_dmatx->init.direction          = DMA_PERIPH_TO_PERIPH;  (2)
    hi2c->p_dmatx->init.src_request        = DMA_REQUEST_UART0_RX;  (3)
    hi2c->p_dmatx->init.dst_request        = DMA_REQUEST_I2C0_TX;   (4)
    hi2c->p_dmatx->init.src_increment      = DMA_SRC_NO_CHANGE;     (5)
    hi2c->p_dmatx->init.dst_increment      = DMA_DST_NO_CHANGE;     (6)
    hi2c->p_dmatx->init.src_data_alignment = DMA_SDATAALIGN_BYTE;   (7)
    hi2c->p_dmatx->init.dst_data_alignment = DMA_DDATAALIGN_BYTE;   (8)
    hi2c->p_dmatx->init.mode               = DMA_NORMAL;
    hi2c->p_dmatx->init.priority           = DMA_PRIORITY_LOW;

    hal_dma_deinit(p_i2c->p_dmatx);
    hal_dma_init(p_i2c->p_dmatx);
```

```
}
```

相关说明如下：

（1）：硬件IO初始化，将I2C映射的引脚配置为I2C模式。

（2）：配置DMA为P2P模式。

（3）～（4）：配置DMA通道源和目标分别为UART RX和I2C TX。

（5）～（6）：DMA通道源和目标地址都为固定模式。

（7）～（8）：DMA通道传输单元位宽为8 bits。

3. 配置从设备地址。

在传输之前需要将I2C设置为Master，并配置从设备地址。代码如下：

```
/* Enable Master Mode and Set Slave Address */
ll_i2c_disable(p_i2c_handle->p_instance);
ll_i2c_enable_master_mode(p_i2c_handle->p_instance);
ll_i2c_set_slave_address(p_i2c_handle->p_instance, SLAVE_DEV_ADDR);
ll_i2c_enable(p_i2c_handle->p_instance);
```

4. 设置UART接收阈值与DMA Burst长度。代码如下：

```
ll_i2c_set_dma_tx_data_level(p_i2c_handle->p_instance, 4U);
ll_uart_set_rx_fifo_threshold(LOG_UART_GRP, LL_UART_RX_FIFO_TH_CHAR_1);
ll_dma_set_source_burst_length(DMA, p_i2c_handle->p_dmatx->channel,
    LL_DMA_SRC_BURST_LENGTH_1);
ll_dma_set_destination_burst_length(DMA, p_i2c_handle->p_dmatx->channel,
    LL_DMA_DST_BURST_LENGTH_4);
```

5. 等待UART接收到数据。代码如下：

```
/* Wait until receive any data */
while(!ll_uart_is_active_flag_rfne(SERIAL_PORT_GRP));
```

6. 开始DMA传输并等待传输完成。代码如下：

```
hal_dma_start(p_i2c_handle->p_dmatx, (uint32_t)&UART0->RBR_DLL_THR,
    (uint32_t)& p_i2c_handle->p_instance->DATA_CMD, TEST_LENGTH);
/* Enable DMA Request */
ll_i2c_enable_dma_req_tx(p_i2c_handle->p_instance);
hal_dma_poll_for_transfer(p_i2c_handle->p_dmatx, 1000);
```

7. I2C数据传输的最后一个字节需要使能STOP信号，因此在DMA传输完成后需增加STOP流程。代码如下：

```
/* Disable DMA Request */
ll_i2c_disable_dma_req_tx(p_i2c_handle->p_instance);
while(RESET == ll_i2c_is_active_flag_status_tfnf(p_i2c_handle->p_instance));
ll_i2c_transmit_data8(p_i2c_handle->p_instance, 0, LL_I2C_CMD_MST_WRITE |
    LL_I2C_CMD_MST_GEN_STOP);
```

3.6.1.2 测试验证

1. 用GProgrammer下载*i2c_dma_uart_sk_r2_fw .bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示I2C通过DMA方式发送从UART中接收的数据的执行结果。

3.6.2 I2C ADXL345

I2C ADXL345示例实现了I2C驱动与I2C设备通信，通过I2C向ADXL345发送数据，并从ADXL345读取数据。

I2C ADXL345示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\i2c\i2c_master_adxl345`，其中工程文件在文件夹Keil_5下。

3.6.2.1 代码理解

示例工程流程图如图 3-16所示：

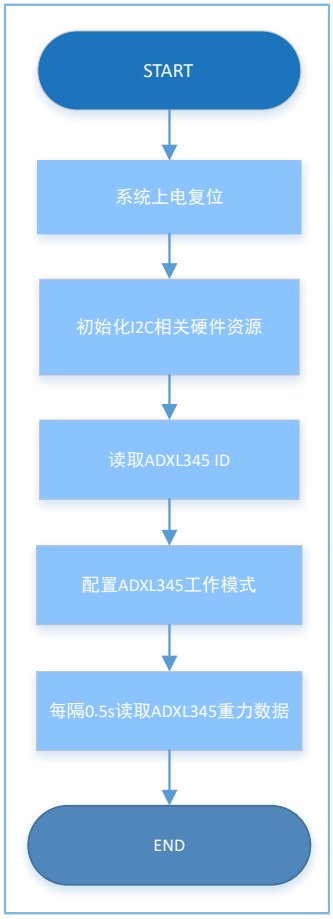

图 3-16 I2C ADXL345示例工程流程图

1. 配置I2C模块。

   I2C详细配置参数请参考3.6.1 I2C DMA UART。此示例工程中，speed选择I2C_SPEED_400K，addressing_mode选择I2C_ADDRESSINGMODE_7BIT，general_call_mode选择I2C_GENERALCALL_DISABLE。

2. I2C硬件初始化。

```
void hal_i2c_msp_init(i2c_handle_t *p_i2c)
{
    gpio_init_t gpio_config = GPIO_DEFAULT_CONFIG;

/*---------------------- Configure the I2C Pins ----------------------*/
    gpio_config.mode = GPIO_MODE_MUX;
    gpio_config.pull = GPIO_PULLUP;
    gpio_config.mux  = I2C_GPIO_MUX;
    gpio_config.pin  = I2C_SCL_PIN | I2C_SDA_PIN;
    hal_gpio_init(I2C_GPIO_PORT, &gpio_config);         (1)

 /*------------------- Configure the DMA for I2C ------------------*/
    /* Configure the DMA handler for Transmission process */
    s_dma_tx_handle.channel = DMA_Channel0;
    s_dma_tx_handle.init.src_request = DMA_REQUEST_MEM;
     s_dma_tx_handle.init.dst_request = p_i2c->p_instance == I2C0) ? DMA_REQUEST_I2C0_TX :
 DMA_REQUEST_I2C1_TX;
    s_dma_tx_handle.init.direction = DMA_MEMORY_TO_PERIPH;
    s_dma_tx_handle.init.src_increment = DMA_SRC_INCREMENT;                  (2)
    s_dma_tx_handle.init.dst_increment = DMA_DST_NO_CHANGE;                  (3)
    s_dma_tx_handle.init.src_data_alignment = DMA_SDATAALIGN_BYTE;                  (4)
    s_dma_tx_handle.init.dst_data_alignment = DMA_DDATAALIGN_BYTE;                  (5)
    s_dma_tx_handle.init.mode = DMA_NORMAL;
    s_dma_tx_handle.init.priority = DMA_PRIORITY_LOW;

    hal_dma_deinit(&s_dma_tx_handle);
    hal_dma_init(&s_dma_tx_handle);

/* Associate the initialized DMA handle to the I2C handle */
    __HAL_LINKDMA(p_i2c, p_dmatx, s_dma_tx_handle);       (6)

   /* Configure the DMA handler for reception process */
    s_dma_rx_handle.channel = DMA_Channel1;
     s_dma_rx_handle.init.src_request = (p_i2c->p_instance == I2C0) ? DMA_REQUEST_I2C0_RX :
 DMA_REQUEST_I2C1_RX;
    s_dma_rx_handle.init.dst_request = DMA_REQUEST_MEM;
    s_dma_rx_handle.init.direction = DMA_PERIPH_TO_MEMORY;
    s_dma_rx_handle.init.src_increment = DMA_SRC_NO_CHANGE;                  (7)
    s_dma_rx_handle.init.dst_increment = DMA_DST_INCREMENT;                  (8)
    s_dma_rx_handle.init.src_data_alignment = DMA_SDATAALIGN_BYTE;                  (9)
```

```
    s_dma_rx_handle.init.dst_data_alignment = DMA_DDATAALIGN_BYTE;                (10)
    s_dma_rx_handle.init.mode = DMA_NORMAL;
    s_dma_rx_handle.init.priority = DMA_PRIORITY_LOW;

    hal_dma_deinit(&s_dma_rx_handle);
    hal_dma_init(&s_dma_rx_handle);

   /* Associate the initialized DMA handle to the the I2C handle */
    __HAL_LINKDMA(p_i2c, p_dmarx, s_dma_rx_handle);                               (11)

    /* NVIC for DMA */
    hal_nvic_set_priority(DMA_IRQn, 0, 1);
    hal_nvic_clear_pending_irq(DMA_IRQn);
    hal_nvic_enable_irq(DMA_IRQn);

    /* NVIC for I2C */
    hal_nvic_set_priority(I2C_GET_IRQNUM(p_i2c->p_instance), 0, 1);
    hal_nvic_clear_pending_irq(I2C_GET_IRQNUM(p_i2c->p_instance));
    hal_nvic_enable_irq(I2C_GET_IRQNUM(p_i2c->p_instance));}
```

具体流程说明如下：

（1）：硬件IO初始化，将I2C映射的引脚配置为I2C模式。

（2）～（5）：配置TX DMA通道源和目标的地址增长模式和传输单元位宽。I2C TX通道源为RAM内数组，地址为自增模式；目标为I2C TX FIFO地址，地址不需要增长。I2C TX通道的传输单元位宽为8 bits。

（6）：将TX DMA通道句柄注册进I2C实例句柄内。

（7）～（10）：配置RX DMA通道源和目标的地址增长模式和传输单元位宽。I2C RX通道源为I2C RX FIFO地址，地址不需要增长；目标为RAM内数组，地址为自增模式。I2C RX通道的传输单元位宽为8 bits。

（11）：将RX DMA通道句柄注册进I2C实例句柄内。

3. 用户需根据不同I2C设备在I2C驱动中实现设备的读写操作。代码如下：

```
void i2c_write_adxl345(uint8_t reg_addr, uint8_t *buf, uint8_t size)
{
    uint8_t wdata[256] = {0};

    wdata[0] = reg_addr;
    memcpy(&wdata[1], buf, size);
    hal_i2c_master_transmit(&g_i2c_handle, SLAVE_DEV_ADDR, wdata, size + 1, 5000);
}

void i2c_read_adxl345(uint8_t reg_addr, uint8_t *buf, uint8_t size)
{
    uint8_t wdata[1] = {0};
```

```
    wdata[0] = reg_addr;
    hal_i2c_master_transmit(&g_i2c_handle, SLAVE_DEV_ADDR, wdata, 1, 5000);
    hal_i2c_master_receive(&g_i2c_handle, SLAVE_DEV_ADDR, buf, size, 5000);
}
```

3.6.2.2 测试验证

1. 用GProgrammer下载*i2c_adxl345_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示通过I2C向ADXL345传感器发送数据，并从ADXL345传感器读取数据并打印ADXL345数据的执行过程。

3.6.3 I2C Master & Slave

I2C Master & Slave示例实现了I2C模块之间的通信，其中一端为Master，另一端为Slave。

I2C Master & Slave示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\i2c\i2c_master_slave`，其中工程文件在文件夹Keil_5下。

3.6.3.1 代码理解

示例工程流程图如图 3-17：

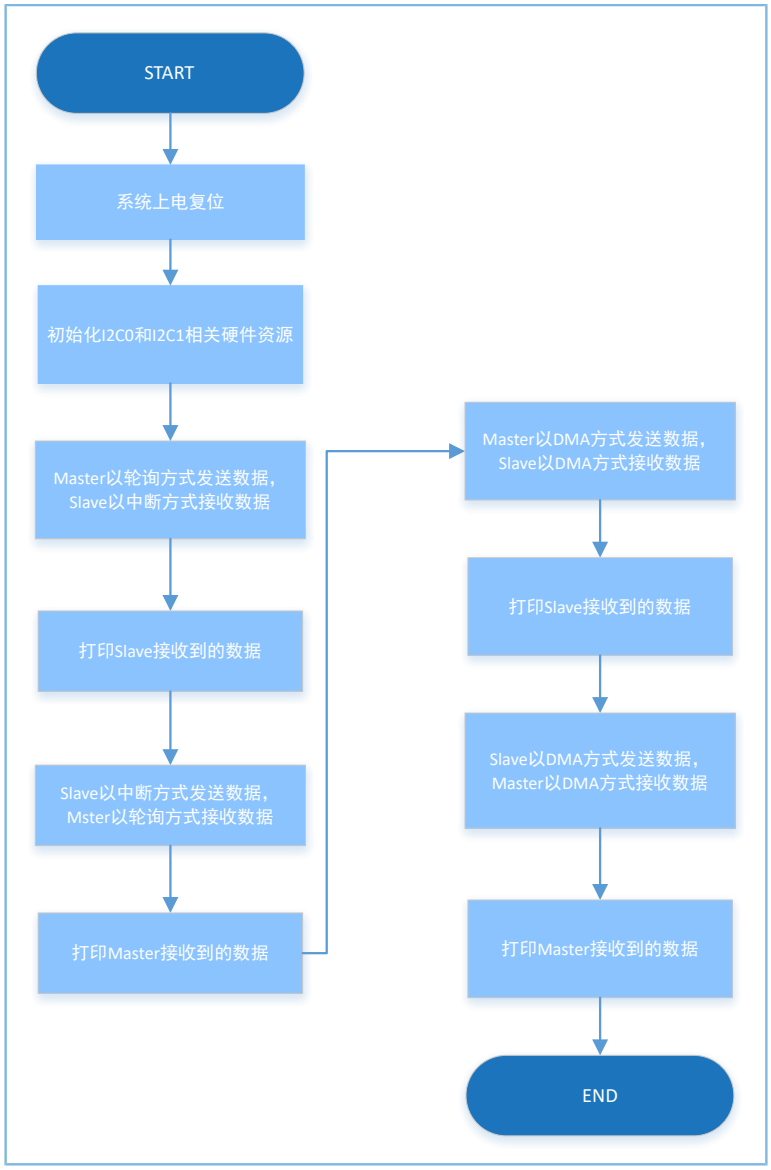

图 3-17 I2C Master & Slave示例工程流程图

1. 配置I2C模块。

   I2C详细配置参数请参考3.6.1 I2C DMA UART。此示例工程中，speed选择I2C_SPEED_400K，addressing_mode选择I2C_ADDRESSINGMODE_7BIT，general_call_mode选择I2C_GENERALCALL_DISABLE。

   Master和Slave需设置各自不同的设备地址：

```
#define MASTER_DEV_ADDR                 0x4D
#define SLAVE_DEV_ADDR                  0x55

g_i2cs_handle.init.own_address  = SLAVE_DEV_ADDR;

g_i2cm_handle.init.own_address = MASTER_DEV_ADDR;
```

2. I2C硬件初始化。

```
void hal_i2c_msp_init(i2c_handle_t *p_i2c)
{
    gpio_init_t gpio_config = GPIO_DEFAULT_CONFIG;
    msio_init_t msio_config = MSIO_DEFAULT_CONFIG;

    if (p_i2c->p_instance == I2C_MASTER_MODULE)
    {
        msio_config.pin  = I2C_MASTER_SCL_PIN | I2C_MASTER_SDA_PIN;
        msio_config.pull = MSIO_PULLUP;
        msio_config.mux  = I2C_MASTER_GPIO_MUX;
        hal_msio_init(&msio_config);
    }
    else if (p_i2c->p_instance == I2C_SLAVE_MODULE)
    {
       gpio_config.mode = GPIO_MODE_MUX;
       gpio_config.pull = GPIO_PULLUP;
       gpio_config.pin  = I2C_SLAVE_SCL_PIN | I2C_SLAVE_SDA_PIN;
       gpio_config.mux  = I2C_SLAVE_GPIO_MUX;
       hal_gpio_init(I2C_SLAVE_GPIO_PORT, &gpio_config);
    }
    NVIC_ClearPendingIRQ(I2C_GET_IRQNUM(p_i2c->p_instance));
    NVIC_EnableIRQ(I2C_GET_IRQNUM(p_i2c->p_instance));

    if (p_i2c->p_instance == I2C_MASTER_MODULE)
    {
        __HAL_LINKDMA(p_i2c, p_dmatx, s_i2cm_dma_tx_handle);
        __HAL_LINKDMA(p_i2c, p_dmarx, s_i2cm_dma_rx_handle);
        s_i2cm_dma_tx_handle.p_parent = p_i2c;
        s_i2cm_dma_rx_handle.p_parent = p_i2c;
        s_i2cm_dma_tx_handle.channel  = DMA_Channel0;
        s_i2cm_dma_rx_handle.channel  = DMA_Channel1;
    }
    else if (p_i2c->p_instance == I2C_SLAVE_MODULE)
    {
        __HAL_LINKDMA(p_i2c, p_dmatx, s_i2cs_dma_tx_handle);
        __HAL_LINKDMA(p_i2c, p_dmarx, s_i2cs_dma_rx_handle);
        s_i2cs_dma_tx_handle.p_parent = p_i2c;
        s_i2cs_dma_rx_handle.p_parent = p_i2c;
        s_i2cs_dma_tx_handle.channel  = DMA_Channel2;
        s_i2cs_dma_rx_handle.channel  = DMA_Channel3;
    }
    p_i2c->p_dmatx->init.src_request = DMA_REQUEST_MEM;
    p_i2c->p_dmarx->init.dst_request = DMA_REQUEST_MEM;
    if (p_i2c->p_instance == I2C0)
    {
        p_i2c->p_dmatx->init.dst_request = DMA_REQUEST_I2C0_TX;
        p_i2c->p_dmarx->init.src_request = DMA_REQUEST_I2C0_RX;
    }
    else if (p_i2c->p_instance == I2C1)
```

```
    {
        p_i2c->p_dmatx->init.dst_request = DMA_REQUEST_I2C1_TX;
        p_i2c->p_dmarx->init.src_request = DMA_REQUEST_I2C1_RX;
    }

    p_i2c->p_dmatx->init.direction           = DMA_MEMORY_TO_PERIPH;
    p_i2c->p_dmatx->init.src_increment       = DMA_SRC_INCREMENT;
    p_i2c->p_dmatx->init.dst_increment       = DMA_DST_NO_CHANGE;
    p_i2c->p_dmatx->init.src_data_alignment  = DMA_SDATAALIGN_BYTE;
    p_i2c->p_dmatx->init.dst_data_alignment  = DMA_DDATAALIGN_BYTE;
    p_i2c->p_dmatx->init.mode  = DMA_NORMAL;
    p_i2c->p_dmatx->init.priority  = DMA_PRIORITY_LOW;

    p_i2c->p_dmarx->init.direction           = DMA_PERIPH_TO_MEMORY;
    p_i2c->p_dmarx->init.src_increment       = DMA_SRC_NO_CHANGE;
    p_i2c->p_dmarx->init.dst_increment       = DMA_DST_INCREMENT;
    p_i2c->p_dmarx->init.src_data_alignment  = DMA_SDATAALIGN_BYTE;
    p_i2c->p_dmarx->init.dst_data_alignment  = DMA_DDATAALIGN_BYTE;
    p_i2c->p_dmarx->init.mode  = DMA_NORMAL;
    p_i2c->p_dmarx->init.priority  = DMA_PRIORITY_LOW;

    hal_dma_init(p_i2c->p_dmatx);
    hal_dma_init(p_i2c->p_dmarx);

    hal_nvic_clear_pending_irq(DMA_IRQn);
    hal_nvic_enable_irq(DMA_IRQn);
}
```

各个函数以及参数的具体意义，请参考3.6.2 I2C ADXL345。

3. Slave以中断方式接收数据，Master以轮询方式发送数据，因slave采用非阻塞方式，故在后续步骤中用while判断Slave是否接收完成。代码如下：

```
hal_i2c_slave_receive_it(&g_i2cs_handle, rdata, 256);
hal_i2c_master_transmit(&g_i2cm_handle, SLAVE_DEV_ADDR, wdata, 256, 5000);
while (hal_i2c_get_state(&g_i2cs_handle) != HAL_I2C_STATE_READY);
```

4. Slave以中断方式发送数据，Master以轮询方式接收数据，因slave采用非阻塞方式，故在后续步骤中用while判断Slave是否发送完成。代码如下：

```
hal_i2c_slave_transmit_it(&g_i2cs_handle, wdata, 256);
hal_i2c_master_receive(&g_i2cm_handle, SLAVE_DEV_ADDR, rdata, 256, 5000);
while (hal_i2c_get_state(&g_i2cs_handle) != HAL_I2C_STATE_READY);
```

5. Master和Slave分别以DMA方式进行数据的收发操作。因采用非阻塞方式，故在后续步骤中用while判断是否发送/接收完成。代码如下：

```
hal_i2c_slave_receive_dma(&g_i2cs_handle, rdata, 256);
hal_i2c_master_transmit_dma(&g_i2cm_handle, SLAVE_DEV_ADDR, wdata, 256);
while (hal_i2c_get_state(&g_i2cs_handle) != HAL_I2C_STATE_READY);

hal_i2c_slave_transmit_dma(&g_i2cs_handle, wdata, 256);
```

```
hal_i2c_master_receive_dma(&g_i2cm_handle, SLAVE_DEV_ADDR, rdata, 256);
while (hal_i2c_get_state(&g_i2cs_handle) != HAL_I2C_STATE_READY);
```

3.6.3.2 测试验证

1. 用GProgrammer下载*i2c_master_slave_sk_r2_fw.bin*至开发板。
2. 连接主从设备开发板上对应的IO引脚。
3. 将开发板串口连接至PC端，打开并配置GRUart。
4. 在GRUart的“Receive Data”窗口中将会显示I2C模块之间的数据交互结果。

3.7 I2S

I2S（Inter-IC Sound Bus）是飞利浦公司为数字音频设备之间的音频数据传输而制定的一种总线标准。一般用于蓝牙耳机、蓝牙音箱进行音频数据传输。

以下章节主要介绍使用I2S接口以Interrupt、Query、DMA三种方式实现数据的读写和主从接口的使用方法。

3.7.1 I2S Master Audio

I2S Master Audio示例中I2S作为Master，输出I2S数据流的示例工程，用户需要配置相应的参数，如时钟源、数据长度、音频频率等信息，然后调用初始化接口完成配置，最后调用中断和DMA的发送接口完成数据的发送。

I2S Master Audio示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\i2s\i2s_master_audio`，其中工程文件在文件夹Keil_5下。

3.7.1.1 代码理解

示例工程流程图如图 3-18所示：

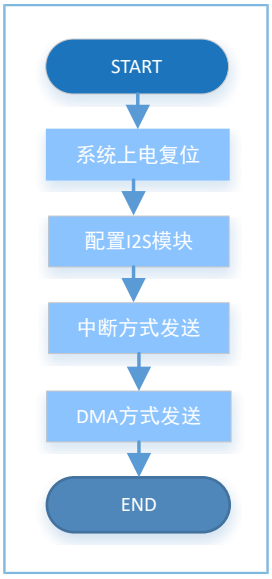

图 3-18 I2S Master Audio示例工程流程图

1. 配置I2S模块。

   选择I2S模块配置代码数据长度为32 bit，时钟源为96 MHz，音频频率为48 kHz，这些参数均可配置为其他值。

```
g_i2sm_handle.p_instance = I2S_M;
g_i2sm_handle.init.data_size = I2S_DATASIZE_32BIT;
g_i2sm_handle.init.clock_source = I2S_CLOCK_SRC_96M;
g_i2sm_handle.init.audio_freq = 48000;

hal_i2s_deinit(&g_i2sm_handle);
hal_i2s_init(&g_i2sm_handle);
```

   - init.data_size：I2S传输数据位宽选择，可选择I2S_DATASIZE_12BIT、I2S_DATASIZE_16BIT、I2S_DATASIZE_20BIT、I2S_DATASIZE_24BIT、I2S_DATASIZE_32BIT。
   - init.clock_source：I2S时钟选择，可选择I2S_CLOCK_SRC_96M、I2S_CLOCK_SRC_32M。
   - init.audio_freq：指定主设备的频率，从设备不需要设置。

   I2S各引脚定义如下，用户可根据参考《GR551x Datasheet》中的pin_mux配置表，设置其他可用引脚：

```
#define I2S_MASTER_WS_PIN               AON_GPIO_PIN_2
#define I2S_MASTER_TX_SDO_PIN           AON_GPIO_PIN_3
#define I2S_MASTER_RX_SDI_PIN           AON_GPIO_PIN_4
#define I2S_MASTER_SCLK_PIN             AON_GPIO_PIN_5
```

2. 调用hal_i2s_transmit()接口以轮询方式发送数据。代码如下：

```
hal_i2s_transmit(&g_i2sm_handle, wdata, sizeof(wdata) >> 2, 1000);
```

3. 调用hal_i2s_transmit_it()接口以中断方式发送数据，因采用非阻塞方式，故在后续步骤中用while判断是否发送完成。代码如下：

```
hal_i2s_transmit_it(&g_i2sm_handle, wdata, sizeof(wdata) >> 2);
while (hal_i2s_get_state(&g_i2sm_handle) != HAL_I2S_STATE_READY);
```

4. 调用hal_i2s_transmit_dma()接口以DMA方式发送，因采用非阻塞方式，故在后续步骤中用while判断是否发送完成。代码如下：

```
hal_i2s_transmit_dma(&g_i2sm_handle, wdata, sizeof(wdata) >> 2);
while (hal_i2s_get_state(&g_i2sm_handle) != HAL_I2S_STATE_READY);
```

3.7.1.2 测试验证

1. 用GProgrammer下载*i2s_audio_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示I2S模块之间的数据交互信息。

3.7.2 I2S Master DMA UART

I2S Master DMA UART示例用于验证DMA的p2p功能，数据通过DMA从UART发送至I2S。

I2S Master DMA UART示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\i2s\i2s_master_dma_uart`，其中工程文件在文件夹Keil_5下。

3.7.2.1 代码理解

示例工程流程图如图 3-19所示：

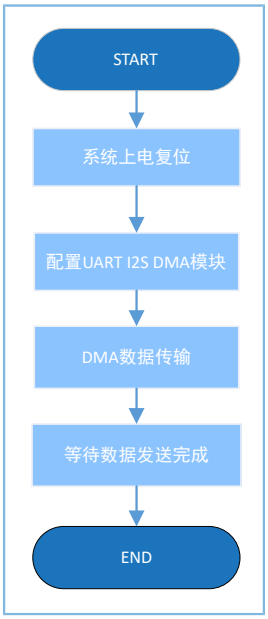

图 3-19 I2S Master DMA UART示例工程流程图

1. 配置I2S模块。

```
hi2s->p_instance        = I2S_M;
hi2s->init.data_size    = I2S_DATASIZE_16BIT;
hi2s->init.clock_source = I2S_CLOCK_SRC_32M;
hi2s->init.audio_freq   = 4000;

hal_i2s_deinit(hi2s);
hal_i2s_init(hi2s);
```

- I2S详细配置参数请参考3.7.1 I2S Master Audio。此示例工程中，init.data_size选择I2S_DATASIZE_16BIT，init.clock_source选择I2S_CLOCK_SRC_32M，init.audio_freq选择4000。
- UART通过调用app_log_init()进行配置，同时可作为日志打印。

2. 使能I2S DMA、刷新TX FIFO，以及使能TX通道、TX block。代码如下：

```
__HAL_I2S_ENABLE_DMA(hi2s);
/* Flush TX FIFO */
ll_i2s_clr_txfifo_all(hi2s->p_instance);
/* Enable channel TX */
ll_i2s_enable_tx(hi2s->p_instance, 0);
/* Enable TX block */
ll_i2s_enable_txblock(hi2s->p_instance);
```

3. 设置UART接收阈值与DMA Burst长度。代码如下：

```
ll_uart_set_rx_fifo_threshold(SERIAL_PORT_GRP, LL_UART_RX_FIFO_TH_QUARTER_FULL);
ll_dma_set_source_burst_length(DMA, hi2s->p_dmatx->channel,
                               LL_DMA_SRC_BURST_LENGTH_8);
ll_dma_set_destination_burst_length(DMA, hi2s->p_dmatx->channel,
                                    LL_DMA_DST_BURST_LENGTH_8);
```

4. 等待UART接收到数据。代码如下：

```
/* Wait until receive any data */
while(!ll_uart_is_active_flag_rfne(SERIAL_PORT_GRP));
```

5. 开始DMA传输、使能TX FIFO空时触发中断、使能I2S时钟，并等待DMA传输完成。代码如下：

```
hal_dma_start(hi2s->p_dmatx, (uint32_t)&UART0->RBR_DLL_THR,
(uint32_t)&hi2s->p_instance->TXDMA, TEST_LENGTH);
__HAL_I2S_ENABLE_IT(hi2s, I2S_IT_TXFE);
/* Enable clock */
ll_i2s_enable_clock(hi2s->p_instance);
hal_dma_poll_for_transfer(hi2s->p_dmatx, 1000);
```

3.7.2.2 测试验证

1. 用GProgrammer下载*i2s_dma_uart_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示数据通过DMA从UART至I2S发送数据交互结果。

3.7.3 I2S Master & Slave

I2S Master & Slave示例用于验证I2S的数据传输功能，通过宏MASTER_BOARD确定为Master或Slave。

该示例需要两块开发板，一块作为Master，一块作为Slave。

I2S Master & Slave示例的源代码和工程文件位于SDK_Folder\projects\peripheral\i2s\i2s_master_salve\，其中工程文件在文件夹Keil_5下。

3.7.3.1 代码理解

示例工程流程图如图 3-20所示：

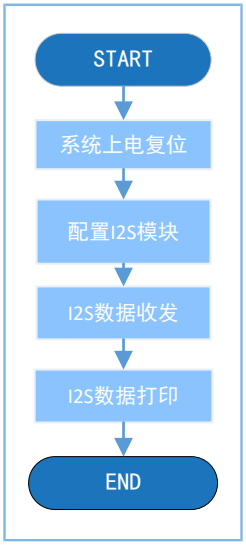

图 3-20 I2S Master & Slave示例工程流程图

1. 配置I2S模块。

   I2S详细配置参数请参考3.7.1 I2S Master Audio。

```
#ifdef MASTER_BOARD
    g_i2s_handle.p_instance  = I2S_M;
    g_i2s_handle.init.audio_freq  = 48000;
#else
    g_i2s_handle.p_instance  = I2S_S;
#endif  /* MASTER_BOARD */
    g_i2s_handle.init.data_size    = I2S_DATASIZE_32BIT;
    g_i2s_handle.init.clock_source = I2S_CLOCK_SRC_96M;
    hal_i2s_init(&g_i2s_handle);
```

2. Master分别使用DMA、中断、轮询方式和Slave进行数据交互。

   - 因I2S目前采用用户控制时钟开关状态，故为了防止接收或者发送不必要的数据，需调用i2s_flush_rx_fifo()或者i2s_flush_tx_fifo()接口，清除RX/TX FIFO中的数据。
   - Master在操作完成后调用__HAL_I2S_DISABLE_CLOCK()关闭时钟或者重新初始化I2S，防止时钟使能设备一直处于收发阶段。
   - 采用非阻塞方式的接口，需在后续步骤中用while判断是否收发完成。代码如下：

```
i2s_flush_rx_fifo(&g_i2s_handle);
hal_i2s_transmit_receive_dma(&g_i2s_handle, wdata, rdata, sizeof(wdata) >> 2);
while (!tx_rx_flag);
while (hal_i2s_get_state(&g_i2s_handle) != HAL_I2S_STATE_READY);
sys_delay_ms(1);
hal_i2s_deinit(&g_i2s_handle);
hal_i2s_init(&g_i2s_handle);
hal_i2s_transmit_receive_it(&g_i2s_handle, wdata, rdata, sizeof(wdata) >> 2);
while (!tx_rx_flag);
while (hal_i2s_get_state(&g_i2s_handle) != HAL_I2S_STATE_READY);
sys_delay_ms(1);
hal_i2s_deinit(&g_i2s_handle);
hal_i2s_init(&g_i2s_handle);

i2s_flush_rx_fifo(&g_i2s_handle);
hal_i2s_transmit_receive(&g_i2s_handle, wdata, rdata, sizeof(wdata) >> 2, 2000);
while (hal_i2s_get_state(&g_i2s_handle) != HAL_I2S_STATE_READY);
sys_delay_ms(1);
hal_i2s_deinit(&g_i2s_handle);
hal_i2s_init(&g_i2s_handle);

i2s_flush_rx_fifo(&g_i2s_handle);
hal_i2s_transmit_it(&g_i2s_handle, wdata, sizeof(wdata) >> 2);
while (hal_i2s_get_state(&g_i2s_handle) != HAL_I2S_STATE_READY);
sys_delay_ms(1);
hal_i2s_deinit(&g_i2s_handle);
hal_i2s_init(&g_i2s_handle);
sys_delay_ms(10);
i2s_flush_rx_fifo(&g_i2s_handle);
hal_i2s_receive_it(&g_i2s_handle, rdata, sizeof(rdata) >> 2);
while (hal_i2s_get_state(&g_i2s_handle) != HAL_I2S_STATE_READY);
sys_delay_ms(1);
hal_i2s_deinit(&g_i2s_handle);
hal_i2s_init(&g_i2s_handle);
```

3. Slave分别使用DMA，中断，轮询方式和Master进行数据交互。

   为了防止接收或者发送不必要的数据，需调用i2s_flush_rx_fifo()或者i2s_flush_tx_fifo()接口，清除RX/TX FIFO中的数据。代码如下：

```
i2s_flush_rx_fifo(&g_i2s_handle);
hal_i2s_transmit_receive_dma(&g_i2s_handle, wdata, rdata, sizeof(wdata) >> 2);
while (!tx_rx_flag);
while (hal_i2s_get_state(&g_i2s_handle) != HAL_I2S_STATE_READY);
hal_i2s_deinit(&g_i2s_handle);
hal_i2s_init(&g_i2s_handle);

i2s_flush_rx_fifo(&g_i2s_handle);
hal_i2s_transmit_receive_it(&g_i2s_handle, wdata, rdata, sizeof(wdata) >> 2);
```

```
while (!tx_rx_flag);
while (hal_i2s_get_state(&g_i2s_handle) != HAL_I2S_STATE_READY);
hal_i2s_deinit(&g_i2s_handle);
hal_i2s_init(&g_i2s_handle);

i2s_flush_rx_fifo(&g_i2s_handle);
hal_i2s_transmit_receive(&g_i2s_handle, wdata, rdata, sizeof(wdata) >> 2, 2000);
while (hal_i2s_get_state(&g_i2s_handle) != HAL_I2S_STATE_READY);
hal_i2s_deinit(&g_i2s_handle);
hal_i2s_init(&g_i2s_handle);

i2s_flush_rx_fifo(&g_i2s_handle);
hal_i2s_receive_it(&g_i2s_handle, rdata, sizeof(rdata) >> 2);
while (hal_i2s_get_state(&g_i2s_handle) != HAL_I2S_STATE_READY);
hal_i2s_deinit(&g_i2s_handle);
hal_i2s_init(&g_i2s_handle);

i2s_flush_rx_fifo(&g_i2s_handle);
hal_i2s_transmit_it(&g_i2s_handle, wdata, sizeof(wdata) >> 2);
while (hal_i2s_get_state(&g_i2s_handle) != HAL_I2S_STATE_READY);
sys_delay_ms(2);
hal_i2s_deinit(&g_i2s_handle);
```

3.7.3.2 测试验证

1. 在工程配置中定义宏MASTER_BOARD，编译并下载至主设备开发板；工程配置中取消宏定义MASTER_BOARD，编译并下载至从设备开发板。
2. 连接主从设备开发板的对应IO引脚。
3. 将主从设备开发板的串口分别连接至PC端，打开并配置GRUart。
4. 在GRUart的“Receive Data”窗口中将会显示I2S的数据传输结果。

3.8 PKC

PKC大数据加、减、移位、取模等运算模块。利用该模块可对超过32位的数据进行加、减、移位、取模等运算。一般用于生成加密运算中所使用的计算因子。

3.8.1 PKC

PKC的示例工程实现了大数模加、大数模减、大数模乘、大数模对比、大数模逆、大数模移位、大数加、大数乘、大数RSA模幂、大数ECC点乘运算。

本章节以大数模加运算为例，进行PKC示例的代码及测试验证说明。其他运算的相关代码请参考示例工程代码。

PKC示例的源代码和工程文件位于SDK_Folder\projects\peripheral\pkc\pkc，其中工程文件在文件夹Keil_5下。

3.8.1.1 代码理解

PKC示例工程（大数模加运算）流程图如图 3-21所示：

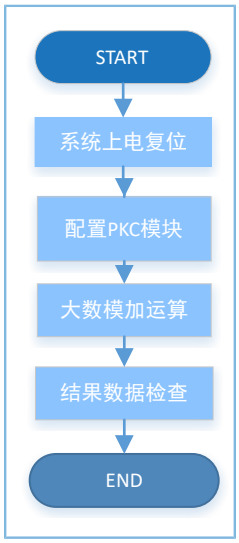

图 3-21 PKC示例工程流程图

1. 配置PKC模块。

```
pkc_modular_add_t PKC_ModularAddStruct = {
    .p_A = In_a,
    .p_B = In_b,
    .p_P = In_n
};

PKCHandle.p_instance          = PKC;
PKCHandle.p_result            = Out_c;
PKCHandle.init.p_ecc_curve    = &ECC_CurveInitStruct;
PKCHandle.init.data_bits    = 256;
PKCHandle.init.secure_mode = PKC_SECURE_MODE_DISABLE;
PKCHandle.init.random_func = NULL;

hal_pkc_deinit(&PKCHandle);
hal_pkc_init(&PKCHandle);
```

- init.p_ecc_curve：椭圆曲线描述指针。
- init.data_bits：数据位数大小。
- init.secure_mode：安全模式设置，可选择PKC_SECURE_MODE_DISABLE、PKC_SECURE_MODE_ENABLE。
- init.random_func：函数指针，指向生成随机数的函数。

2. 调用阻塞接口hal_pkc_modular_add()或中断接口hal_pkc_modular_add_it()，进行In_a和In_b大数模加运算，结果输出到Out_c，并与Exc_c进行比较。代码如下：

```
if (HAL_OK != hal_pkc_modular_add(&PKCHandle, &PKC_ModularAddStruct, 1000))
```

```
{
    printf("\r\n%dth add operation got error\r\n", count++);
    continue;
}
if (check_result((uint8_t*)"pkc_modular_add", Out_c, Exc_c, PKCHandle.init.data_bits))
{
    error++;
}

if (HAL_OK != hal_pkc_modular_add_it(&PKCHandle, &PKC_ModularAddStruct))
{
    printf("\r\n%dth add it operation got error\r\n", count++);
    continue;
}
while(int_done_flag == 0);
if (check_result((uint8_t*)"pkc_modular_add_it", Out_c, Exc_c, PKCHandle.init.data_bits))
{
    error++;
}
```

3.8.1.2 测试验证

1. 用GProgrammer下载*pkc_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将显示PKC的数据执行结果。

3.9 PWM

脉冲宽度调制（Pulse width modulation，即PWM）是一种对模拟信号电平进行数字编码的方法。一般用于输出固定频率的方波或者驱动LED灯以固定频率闪烁。

以下章节主要介绍使用PWM以周期模式或呼吸模式输出频率和占空比可控的信号，最终控制LED灯的亮灭。

3.9.1 PWM Breath

PWM Breath示例工程用于实现PWM的呼吸模式。当示例工程工作时，可以通过逻辑分析仪观察其波形的占空比从0至100%变化，同时通过LED的闪烁快慢变化来呈现。

PWM Breath示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\pwm\pwm_breath`，其中工程文件在文件夹Keil_5下。

3.9.1.1 代码理解

示例工程流程图图 3-22所示：

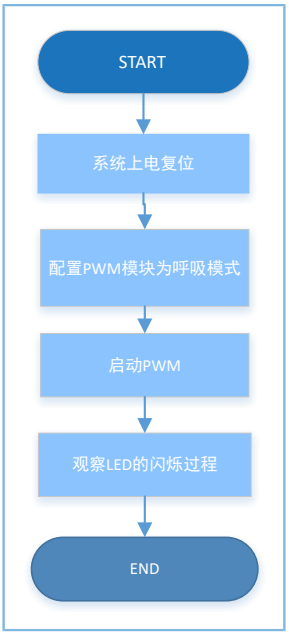

图 3-22 PWM Breath示例工程流程图

1. 配置PWM模块。

```
led1_handle.p_instance = PWM1_MODULE;
led1_handle.active_channel = HAL_PWM_ACTIVE_CHANNEL_B;
led1_handle.init.mode = PWM_MODE_BREATH;
led1_handle.init.align = PWM_ALIGNED_EDGE;
led1_handle.init.freq = 100;   //PWM output freq = 100Hz
led1_handle.init.bperiod = 500;  //breath period = 500ms
led1_handle.init.hperiod = 200;  //hold period = 200ms, max = 262ms
led1_handle.init.channel_b.duty = 50;
led1_handle.init.channel_b.drive_polarity = PWM_DRIVEPOLARITY_POSITIVE;
hal_pwm_init(&led1_handle);

led2_handle.p_instance = PWM0_MODULE;
led2_handle.active_channel = HAL_PWM_ACTIVE_CHANNEL_C;
led2_handle.init.mode = PWM_MODE_BREATH;
led2_handle.init.align = PWM_ALIGNED_EDGE;
led2_handle.init.freq = 100;  //PWM output freq = 100Hz
led2_handle.init.bperiod = 500; //breath period = 500ms
led2_handle.init.hperiod = 200; //hold period = 200ms, max = 262ms
led2_handle.init.channel_c.duty = 50;
led2_handle.init.channel_c.drive_polarity = PWM_DRIVEPOLARITY_POSITIVE;
hal_pwm_init(&led2_handle);
```

- active_channel：PWM通道选择，可选择HAL_PWM_ACTIVE_CHANNEL_A、HAL_PWM_ACTIVE_CHANNEL_B、HAL_PWM_ACTIVE_CHANNEL_C、HAL_PWM_ACTIVE_CHANNEL_ALL、HAL_PWM_ACTIVE_CHANNEL_CLEARED。
- init.mode：PWM模式选择，可选择PWM_MODE_FLICKER、PWM_MODE_BREATH。

- init.align：PWM对齐方式，可选择PWM_ALIGNED_EDGE、PWM_ ALIGNED_CENTER。在breath mode下选择对齐方式无效。
- init.freq：PWM频率，取值范围0 ～ SystemCoreClock/2。
- init.bperiod：呼吸模式下指定PWM呼吸周期，取值范围0 ～ 0xFFFFFFFF/SystemCoreClock * 1000，单位（ms）。
- init.hperiod：呼吸模式下的PWM保持时间，取值范围0 ～ 0xFFFFFFFF/SystemCoreClock * 1000，单位（ms）。
- init.channel_b.duty：PWM B通道输出模式占空比，取值范围0 ～ 100。在breath mode下占空比设置无效。
- init.channel_b.drive_polarity：PWM B 通道输出模式驱动器极性，可选择PWM_DRIVEPOLARITY_NEGATIVE、PWM_DRIVEPOLARITY_POSITIVE。

2. 调用led_breath_on()开启PWM BREATH模式。代码如下：

```
void led_breath_on(uint8_t id)
{
    if (id == LED1)
    {
        hal_pwm_start(&led1_handle);
    }
    else if (id == LED2)
    {
        hal_pwm_start(&led2_handle);
    }
}
```

3.9.1.2 测试验证

1. 用GProgrammer下载*pwm_breath_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示PWM执行过程。
4. 观察逻辑分析仪其波形的占空比从0至100%变化。

3.9.2 PWM Flicker

PWM Flicker的示例工程用于实现PWM的固定占空比模式。当示例工程工作时，可以通过逻辑分析仪观察其波形为固定占空比的波形，或观察LED闪烁频率为一固定频率。

PWM Flicker示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\pwm\pwm_flicker`，其中工程文件在文件夹Keil_5下。

3.9.2.1 代码理解

工程流程图如图 3-23所示：

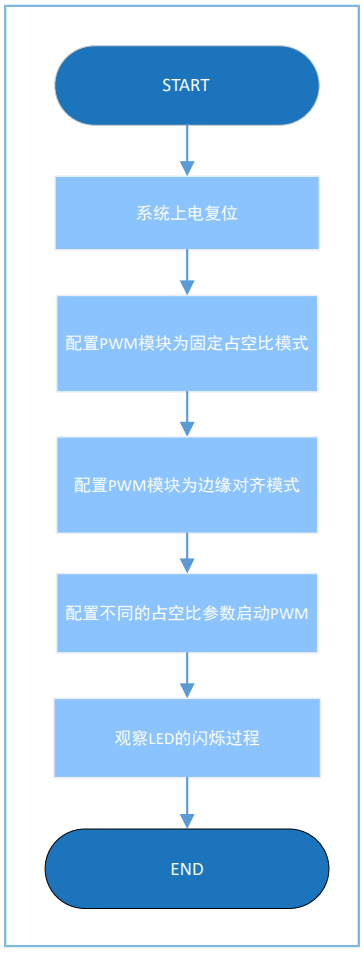

图 3-23 PWM Flicker示例工程流程图

1. 配置PWM模块。

```
led1_handle.p_instance = PWM1_MODULE;
led1_handle.active_channel = HAL_PWM_ACTIVE_CHANNEL_B;
led1_handle.init.mode = PWM_MODE_FLICKER;
led1_handle.init.align = PWM_ALIGNED_EDGE;
led1_handle.init.freq = 1000;           //PWM output freq = 100Hz1000Hz
led1_handle.init.channel_b.duty = 50;
led1_handle.init.channel_b.drive_polarity = PWM_DRIVEPOLARITY_NEGATIVE;
hal_pwm_init(&led1_handle);

led2_handle.p_instance = PWM0_MODULE;
led2_handle.active_channel = HAL_PWM_ACTIVE_CHANNEL_C;
led2_handle.init.mode = PWM_MODE_FLICKER;
led2_handle.init.align = PWM_ALIGNED_EDGE;
led2_handle.init.freq = 1000;           //PWM output freq = 100Hz
led2_handle.init.channel_c.duty = 50;
led2_handle.init.channel_c.drive_polarity = PWM_DRIVEPOLARITY_NEGATIVE;
```

```
hal_pwm_init(&led2_handle);
```

PWM详细配置参数请参考3.9.1 PWM Breath。此示例中init.mode选择PWM_MODE_FLICKER。

2. 调用led_light()开启PWM FLICKER模式。代码如下：

```
void led_light(uint8_t id, uint8_t light)
{
    pwm_channel_init_t pwm_channel;

    pwm_channel.drive_polarity = PWM_DRIVEPOLARITY_NEGATIVE;
    pwm_channel.duty = (uint16_t)light * 100 / 255;
    if (id == LED1)
    {
        hal_pwm_stop(&led1_handle);
        hal_pwm_config_channel(&led1_handle, &pwm_channel, HAL_PWM_ACTIVE_CHANNEL_B);
        hal_pwm_start(&led1_handle);
    }
    else if (id == LED2)
    {
        hal_pwm_stop(&led2_handle);
        hal_pwm_config_channel(&led2_handle, &pwm_channel, HAL_PWM_ACTIVE_CHANNEL_C);
        hal_pwm_start(&led2_handle);
    }
}
```

3.9.2.2 测试验证

1. 用GProgrammer下载*pwm_flicker_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示PWM执行过程。
4. 观察逻辑分析仪其波形为固定占空比的波形，若配置为边缘对齐模式，开启的通道会在占空比周期开始位置对齐；若配置为中心对齐模式，开启的通道会在占空比周期中间以轴对称的方式对齐。观察LED其闪烁频率为一固定频率。

3.10 RNG

随机数生成器（Random Numeral Generator，即RNG）是用于生成随机数的程序或硬件。一般用于生成随机数供其他应用使用。

以下章节主要介绍使用Interrupt和Query两种方式生成RNG数。

3.10.1 RNG Interrupt

RNG Interrupt示例实现了通过中断方式获取RNG随机数。示例工程中采用用户提供的种子以及开关振荡器S0提供的种子两种方式产生随机数。

RNG Interrupt示例的源代码和工程文件位于SDK_Folder\projects\peripheral\rng\rng_interrupt，其中工程文件在文件夹Keil_5下。

3.10.1.1 代码理解

示例工程流程图如图 3-24所示：

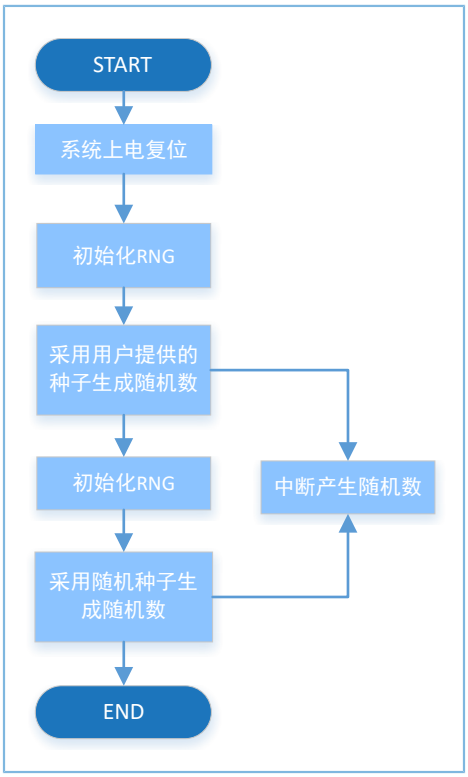

图 3-24 RNG Interrupt示例工程流程图

1. 使用用户提供的种子生成随机数的初始化。

```
uint16_t g_random_seed[8] = {0x1234, 0x5678, 0x90AB, 0xCDEF, 0x1468, 0x2345,
0x5329, 0x2411};

g_rng_handle.init.seed_mode  = RNG_SEED_USER;
g_rng_handle.init.lfsr_mode  = RNG_LFSR_MODE_59BIT;
g_rng_handle.init.out_mode   = RNG_OUTPUT_LFSR;
g_rng_handle.init.post_mode  = RNG_POST_PRO_NOT;

hal_rng_deinit(&g_rng_handle);
hal_rng_init(&g_rng_handle);
```

- seed_mode：种子源的选择，可选择RNG_SEED_FR0_S0、RNG_SEED_USER。
- lfsr_mode：为LFSR选择模式，可选择RNG_LFSR_MODE_59BIT、RNG_LFSR_MODE_128BIT。
- out_mode：输出方式，可选择RNG_OUTPUT_FR0_S0、RNG_OUTPUT_CYCLIC_PARITY、RNG_OUTPUT_CYCLIC、RNG_OUTPUT_LFSR。

说明:

当seed_mode种子源的选择为RNG_SEED_USER时，out_mode输出方式不能选择RNG_OUTPUT_FR0_S0。

- post_mode：后处理模式，可选择RNG_POST_PRO_NOT、RNG_POST_PRO_SKIPPING、RNG_POST_PRO_COUNTING、RNG_POST_PRO_NEUMANN。

2. 使用开关振荡器S0种子生成随机数的初始化。

```
g_rng_handle.init.seed_mode  = RNG_SEED_FR0_S0;
g_rng_handle.init.lfsr_mode  = RNG_LFSR_MODE_128BIT;
g_rng_handle.init.out_mode   = RNG_OUTPUT_FR0_S0;

hal_rng_deinit(&g_rng_handle);
hal_rng_init(&g_rng_handle);
```

- seed_mode：种子源的选择，此处选择RNG_SEED_FR0_S0，开关振荡器s0提供的种子。
- lfsr_mode：LFSR选择，此处选择RNG_LFSR_MODE_128BIT。
- out_mode：输出方式，此处选择RNG_OUTPUT_FR0_S0。

3. 中断方式接口生成随机数。代码如下：

```
hal_rng_generate_random_number_it(&g_rng_handle, g_random_seed);
```

3.10.1.2 测试验证

1. 用GProgrammer下载*rng_interrupt_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示获取到的随机数。

3.10.2 RNG Query

RNG Query示例实现了通过阻塞方式获取RNG随机数。示例工程中采用用户提供的种子以及开关振荡器S0提供的种子两种方式产生随机数。

RNG Query示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\rng\rng_query`，其中工程文件在文件夹Keil_5下。

3.10.2.1 代码理解

示例工程流程图如图 3-25所示：

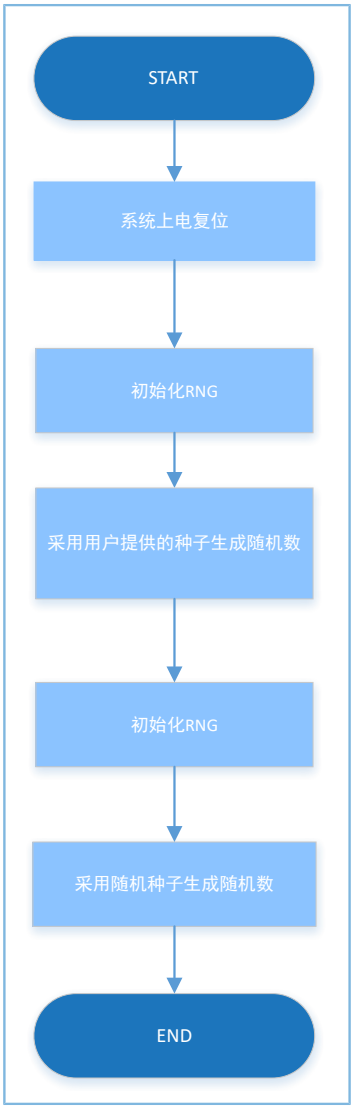

图 3-25 RNG Query示例流程图

1. 使用用户提供的种子生成随机数的初始化。

```
uint16_t g_random_seed[8] = {0x1234, 0x5678, 0x90AB, 0xCDEF, 0x1468,
  0x2345, 0x5329, 0x2411};

g_rng_handle.init.seed_mode  = RNG_SEED_USER;
g_rng_handle.init.lfsr_mode  = RNG_LFSR_MODE_59BIT;
g_rng_handle.init.out_mode   = RNG_OUTPUT_LFSR;
g_rng_handle.init.post_mode  = RNG_POST_PRO_NOT;

hal_rng_deinit(&g_rng_handle);
hal_rng_init(&g_rng_handle);
```

RNG详细配置参数请参考3.10.1 RNG Interrupt。

2. 开关振荡器S0的种子生成随机数的初始化。

```
g_rng_handle.init.seed_mode  = RNG_SEED_FR0_S0;
g_rng_handle.init.lfsr_mode  = RNG_LFSR_MODE_128BIT;
```

```
g_rng_handle.init.out_mode   = RNG_OUTPUT_FR0_S0;

hal_rng_deinit(&g_rng_handle);
hal_rng_init(&g_rng_handle);
```

RNG详细配置参数请参考3.10.1 RNG Interrupt。

3. 阻塞方式接口生成随机数。

```
hal_rng_generate_random_number(&g_rng_handle, g_random_seed, &data);
```

3.10.2.2 测试验证

1. 用GProgrammer下载*rng_query_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示获取到的随机数。

3.11 RTC

RTC（Real-Time Clock）即实时时钟，提供精确的实时时间，或者为电子系统提供精确的时间基准。一般用于计时或者做闹钟应用。

以下章节主要介绍利用RTC实现Calendar日历功能。

3.11.1 Calendar

Calendar示例实现了日历以及闹钟功能。程序中会每秒打印一次当前的时间信息。

Calendar示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\rtc\calendar`，其中工程文件在文件夹Keil_5下。

3.11.1.1 代码理解

示例工程流程图如图 3-26所示：

图 3-26 Calendar示例流程图

1. Calendar的初始化。

```
calendar_time_t time;
calendar_alarm_t alarm;
g_systick_flag = 0;

hal_calendar_init(&g_calendar_handle);
```

- alarm_sel：生成闹钟的方式，可选择CALENDAR_ALARM_SEL_DATE、CALENDAR_ALARM_SEL_WEEKDAY。
- alarm_date_week_mask：闹钟生成的掩码，当设置为CALENDAR_ALARM_SEL_WEEKDAY时，可选择CALENDAR_ALARM_WEEKDAY_SUN - CALENDAR_ALARM_WEEKDAY_SAT其一或任何组合；当设置为CALENDAR_ALARM_SEL_DATE时，可选择1 ～ 31。
- hour：闹钟生成的小时时间。
- min：闹钟生成的分钟时间。

2. 将初始日期设置为19年5月20日8点0分0秒。代码如下：

```
time.year = 19;
time.mon  = 5;
time.date = 20;
time.hour = 8;
time.min  = 0;
time.sec  = 0;
hal_calendar_init_time(&g_calendar_handle, &time)
```

3. 设置闹钟为每个工作日的上午8点01分。此函数必须在每次设置系统时间后重新调用。代码如下：

```
alarm.alarm_sel = CALENDAR_ALARM_SEL_WEEKDAY;
alarm.alarm_date_week_mask = 0x3E;
alarm.hour = 8;
alarm.min  = 1;
hal_calendar_set_alarm(&g_calendar_handle, &alarm);
```

4. 获取当前的时间。代码如下：

```
hal_calendar_get_time(&g_calendar_handle, &time);
```

3.11.1.2 测试验证

1. 用GProgrammer下载*calendar_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中每秒打印一次当前的时间信息。

3.11.2 Alarm

Alarm示例实现了多闹钟功能。示例程序中会每秒打印一次当前的时间信息。

Alarm示例的源代码和工程文件位于SDK_Folder\projects\peripheral\rtc\alarm，其中工程文件在文件夹Keil_5下。

3.11.2.1 代码理解

示例工程流程图如图 3-27所示：

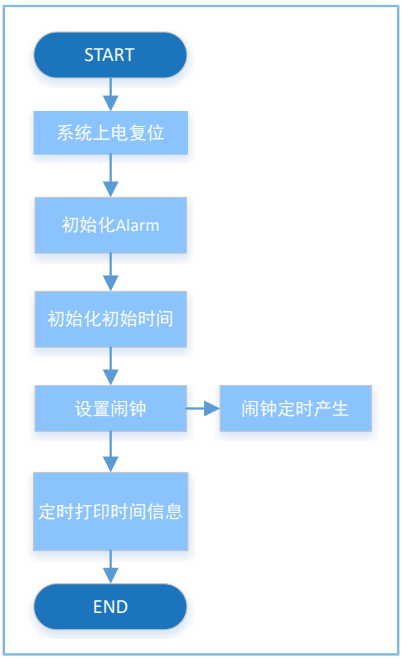

图 3-27 Alarm示例流程图

1. Alarm的初始化。

```
app_time_t s_time = {0, 8, 9, 28, 10, 19 , 0, 0};
```

```
app_alarm_t s_alarm;
app_drv_err_t app_err_code;

app_err_code = app_alarm_init(alarm_data_tag, app_alarm_fun);
if (app_err_code != APP_DRV_SUCCESS)
{
   printf(" Initializing the alarm failed!\r\n");
   return;
}
app_err_code = app_alarm_set_time(&s_time);
if (app_err_code != APP_DRV_SUCCESS)
{
   printf(" Initializing the alarm time failed!\r\n");
   return;
}

app_alarm_reload();
app_alarm_del_all();

s_alarm.hour = 9;
s_alarm.min  = 10;
s_alarm.alarm_sel = CALENDAR_ALARM_SEL_WEEKDAY;
s_alarm.alarm_date_week_mask = 0x7F;
app_alarm_add(&s_alarm, alarm_0);
```

ALARM闹钟详细参数设置，请参考3.11.1 Calendar。

2. 初始化闹钟并设置闹钟用户回调函数。代码如下：

```
app_alarm_init(alarm_data_tag, app_alarm_fun);
```

3. 设置设备系统当前的时间。代码如下：

```
app_alarm_set_time(&s_time);
```

4. 重载系统闹钟，此函数必须在每次设置系统时间后调用。代码如下：

```
app_alarm_reload();
```

5. 删除当前所有闹钟。代码如下：

```
app_alarm_del_all();
```

6. 添加系统闹钟。代码如下：

```
app_alarm_add(&s_alarm, alarm_0);
```

3.11.2.2 测试验证

1. 用GProgrammer下载*alarm_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。

3. 在GRUart的“Receive Data”窗口中每秒打印一次当前的时间信息。

3.12 SPI

串行设备接口（Serial Peripheral Interface，即SPI）是一种高速的，全双工，同步的通信总线。一般用于外接传感器或者FLASH存储器。

以下章节主要介绍使用SPI接口以Interrupt、Query、DMA三种方式实现数据的读写和主从接口的使用方法。

3.12.1 SPIM DMA

SPIM DMA示例实现了SPI采用主模式用DMA方式传输数据。

SPIM DMA示例的源代码和工程文件位于SDK_Folder\projects\peripheral\spi\spi_master_dma，其中工程文件在文件夹Keil_5下。

3.12.1.1 代码理解

示例工程流程图如图 3-28所示：

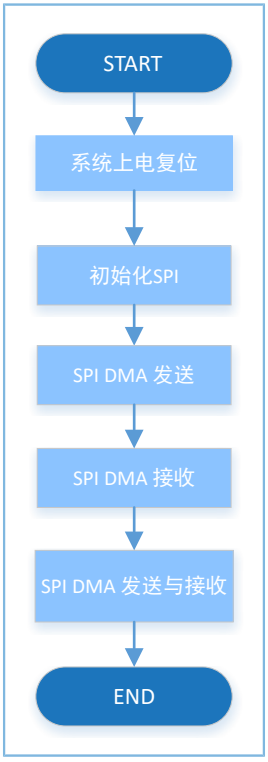

图 3-28 SPIM DMA示例流程图

1. SPIM DMA的初始化。

```
g_spim_handle.init.data_size     = SPI_DATASIZE_8BIT;
g_spim_handle.init.clock_polarity   = SPI_POLARITY_LOW;
g_spim_handle.init.clock_phase    = SPI_PHASE_1EDGE;
g_spim_handle.init.baudrate_prescaler = SystemCoreClock / 2000000;
g_spim_handle.init.ti_mode    = SPI_TIMODE_DISABLE;
```

```
g_spim_handle.init.slave_select   = SPI_SLAVE_SELECT_0;
hal_spi_init(&g_spim_handle);
```

- data_size：SPI数据发送宽度，可选择SPI_DATASIZE_4BIT ～ SPI_DATASIZE_32BIT。
- clock_polarity：空闲状态时的时钟信号，可选择SPI_POLARITY_LOW、SPI_POLARITY_HIGH。
- clock_phase：时钟切换的时间，可选择SPI_PHASE_1EDGE、SPI_PHASE_2EDGE。
- baudrate_prescaler：SPI的时钟，此处为SystemCoreClock / 2000000即20 MHz。
- ti_mode：TI模式使能/禁能，可选择SPI_TIMODE_DISABLE、SPI_TIMODE_ENABLE。
- slave_select：从设备的选择，可选择SPI_SLAVE_SELECT_0、SPI_SLAVE_SELECT_1、SPI_SLAVE_SELECT_ALL。

2. SPI DMA发送接口。代码如下：

```
hal_spi_transmit_dma(&g_spim_handle, tx_buffer, sizeof(tx_buffer));
```

3. SPI DMA接收接口。代码如下：

```
hal_spi_receive_dma(&g_spim_handle, rx_buffer, sizeof(rx_buffer));
```

4. SPI DMA发送接收接口。代码如下：

```
hal_spi_transmit_receive_dma(&g_spim_handle, tx_buffer, rx_buffer, sizeof(rx_buffer));
```

3.12.1.2 测试验证

1. 用GProgrammer下载*spim_dma_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示SPI的数据传输结果。

3.12.2 SPIM DMA UART

SPIM DMA UART示例实现了将UART中的数据通过DMA方式发送至SPI中并用SPI DMA方式发出数据，此示例请使用逻辑分析仪观察数据的正确性。

SPIM DMA UART示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\spi\spim_dma_uart`，其中工程文件在文件夹Keil_5下。

3.12.2.1 代码理解

示例工程流程图如图 3-29所示：

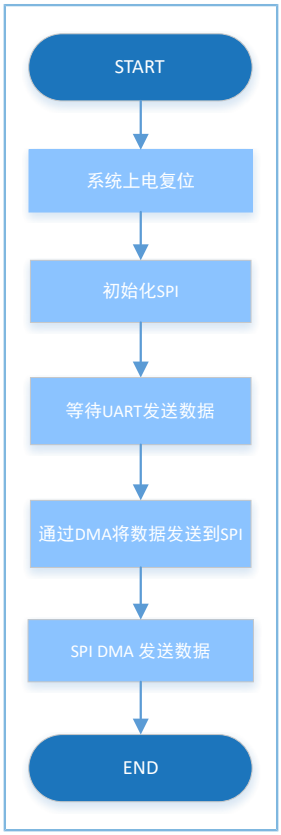

图 3-29 SPIM DMA UART示例流程图

1. SPIM DMA UART的初始化。

```
g_spim_handle.init.data_size    = SPI_DATASIZE_32BIT;
g_spim_handle.init.clock_polarity    = SPI_POLARITY_LOW;
g_spim_handle.init.clock_phase    = SPI_PHASE_1EDGE;
g_spim_handle.init.baudrate_prescaler = SystemCoreClock/2000000;
g_spim_handle.init.ti_mode      = SPI_TIMODE_DISABLE;
g_spim_handle.init.slave_select     = SPI_SLAVE_SELECT_0;
hal_spi_init(&g_spim_handle);
```

SPI详细配置参数请参考3.12.1 SPIM DMA。

2. 在写模式下配置SPI。代码如下：

```
__HAL_SPI_DISABLE(hspi);
ll_spi_set_transfer_direction(hspi->p_instance, LL_SSI_SIMPLEX_TX);
__HAL_SPI_ENABLE(hspi);
```

3. 设置UART接收阈值与DMA Burst长度。代码如下：

```
ll_uart_set_rx_fifo_threshold(SERIAL_PORT_GRP, LL_UART_RX_FIFO_TH_CHAR_1);
ll_dma_set_source_burst_length(DMA, hspi->p_dmatx->channel, LL_DMA_SRC_BURST_LENGTH_1);
ll_dma_set_destination_burst_length(DMA,
                                    hspi->p_dmatx->channel, LL_DMA_DST_BURST_LENGTH_4);
```

4. 等待UART接收到数据。代码如下：

```
while(!ll_uart_is_active_flag_rfne(SERIAL_PORT_GRP));
```

5. 开始DMA传输并等待传输完成。代码如下：

```
hal_dma_start(hspi->p_dmatx, (uint32_t)&UART0->RBR_DLL_THR,
              (uint32_t)&hspi->p_instance->DATA, TEST_LENGTH);
__HAL_SPI_ENABLE_DMATX(hspi);
hal_dma_poll_for_transfer(hspi->p_dmatx, 1000);
```

3.12.2.2 测试验证

1. 用GProgrammer下载*spim_dma_uart_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示SPI的数据传输结果。
4. 使用逻辑分析仪观察数据的正确性。

3.12.3 SPIM ADXL345

SPIM ADXL345示例实现了SPI读取ADXL345的加速度数据并打印到UART终端。

SPIM ADXL345示例的源代码和工程文件位于：`SDK_Folder\projects\peripheral\spi\spi_master_adxl345`，其中工程文件在文件夹Keil_5下。

3.12.3.1 代码理解

示例工程流程图如图 3-30所示：

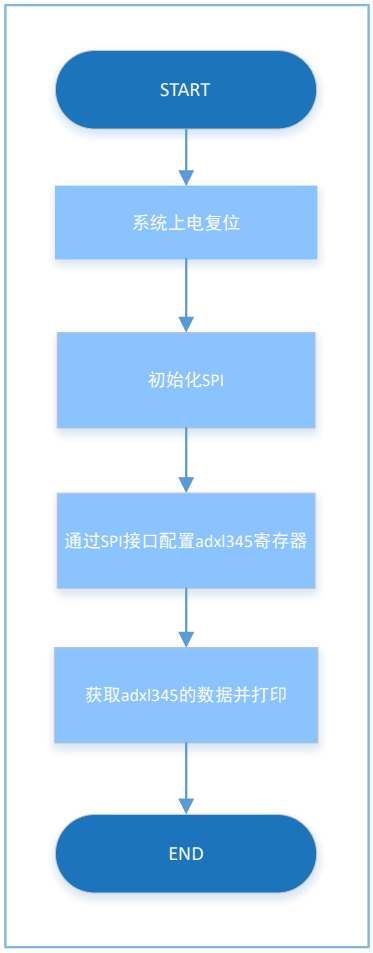

图 3-30 SPIM ADXL345示例流程图

1. 初始化SPIM ADXL345。

```
g_spim_handle.init.data_size  = SPI_DATASIZE_8BIT;
g_spim_handle.init.clock_polarity = SPI_POLARITY_HIGH;
g_spim_handle.init.clock_phase  = SPI_PHASE_2EDGE;
g_spim_handle.init.baudrate_prescaler = SystemCoreClock / 2000000;
g_spim_handle.init.ti_mode   = SPI_TIMODE_DISABLE;
g_spim_handle.init.slave_select = SPI_SLAVE_SELECT_0;

hal_spi_init(&g_spim_handle);
```

SPI详细配置参数设置请参考3.12.1 SPIM DMA。

2. 通过hal_spi_read_eeprom()接口从ADXL345读取数。代码如下：

```
wdata[0] = read_cmd + 0x0;
hal_spi_read_eeprom(&g_spim_handle, wdata, rdata, 1, 1, 5000);
```

3. 通过hal_spi_transmit()接口向ADXL345发送数据。代码如下：

```
/* normal, 100Hz */
wdata[0] = write_cmd + 0x2C;
wdata[1] = 0x0A;
wdata[2] = 0x08;
```

```
wdata[3] = 0x00;
hal_spi_transmit(&g_spim_handle, wdata, 4, 5000);

/* full-bits, 4mg/LSB */
wdata[0] = write_cmd + 0x31;
wdata[1] = 0x09;
hal_spi_transmit(&g_spim_handle, wdata, 2, 5000);

/* FIFO flow */
wdata[0] = write_cmd + 0x38;
wdata[1] = 0x80;
hal_spi_transmit(&g_spim_handle, wdata, 2, 5000);
```

4. 通过hal_spi_read_eeprom()接口从ADXL345读取数据。

```
wdata[0] = read_cmd + 0x1D;
hal_spi_read_eeprom(&g_spim_handle, wdata, rdata, 1, 29, 5000);

wdata[0] = read_cmd + 0x32;
hal_spi_read_eeprom(&g_spim_handle, wdata, rdata, 1, 6, 5000);
```

3.12.3.2 测试验证

1. 用GProgrammer下载*spim_adxl345_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示从ADXL345传感器获取的数据。

3.12.4 SPI Master & Slave

SPI Master & Slave示例实现了SPI主从设备之间的数据交互。该示例中同时实现了主从设备的代码，使用时请用宏定义MASTER_BOARD区分主从设备。

SPI Master & Slave示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\spi\spi_master_slave`，其中工程文件在文件夹Keil_5下。

3.12.4.1 代码理解

示例工程流程图如图 3-31所示：

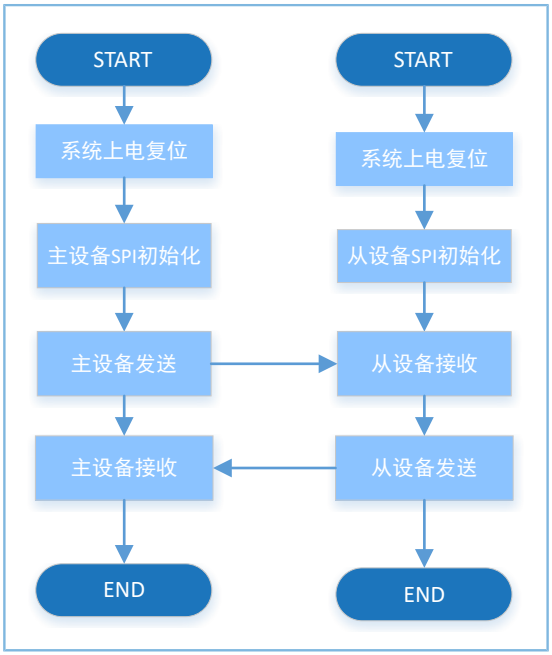

图 3-31 SPI Master & Slave示例流程图

1. SPI Master & Slave的主设备初始化。

```
g_spi_handle.p_instance  = SPIM;
g_spi_handle.init.data_size  = SPI_DATASIZE_32BIT;
g_spi_handle.init.clock_polarity  = SPI_POLARITY_LOW;
g_spi_handle.init.clock_phase = SPI_PHASE_1EDGE;
g_spi_handle.init.baudrate_prescaler = SystemCoreClock / 1000000;
g_spi_handle.init.ti_mode   = SPI_TIMODE_DISABLE;
g_spi_handle.init.slave_select  = SPI_SLAVE_SELECT_0;
hal_spi_init(&g_spim_handle);
```

SPI详细配置参数请参考3.12.1 SPIM DMA。

2. SPI Master & Slave的从设备初始化。

```
g_spi_handle.p_instance  = SPIS;
g_spi_handle.init.data_size  = SPI_DATASIZE_32BIT;
g_spi_handle.init.clock_polarity = SPI_POLARITY_LOW;
g_spi_handle.init.clock_phase = SPI_PHASE_1EDGE;
g_spi_handle.init.ti_mode  = SPI_TIMODE_DISABLE;
hal_spi_init(&g_spim_handle);
```

SPI详细配置参数请参考3.12.1 SPIM DMA。

3. SPI主设备以中断方式发送或者接收数据。因采用非阻塞方式的接口，需在后续步骤中用while判断是否收发完成。代码如下：

```
g_tx_done = 0;
hal_spi_transmit_it(&g_spi_handle, wdata, sizeof(wdata));
/* Wait for send done */
while(!g_tx_done);
```

```
g_rx_done = 0;
hal_spi_receive_it(&g_spi_handle, rdata, sizeof(rdata));
/* Wait for receive done */
while(!g_rx_done);
```

4. SPI从设备以中断方式接收或发送收数据。因采用非阻塞方式的接口，需在后续步骤中用while判断是否收发完成。代码如下：

```
hal_spi_receive_it(&g_spi_handle, rdata, sizeof(rdata))
/* Wait for receive done */
 while(!g_rx_done);

g_tx_done = 0;
hal_spi_transmit_it(&g_spi_handle, wdata, sizeof(wdata))
while(!g_tx_done);
```

5. 主从设备采用非阻塞方式的接口进行收发数据，返回结果或者状态通过hal_spi_rx_cplt_callback()、hal_spi_tx_cplt_callback()、hal_spi_tx_rx_cplt_callback()、hal_spi_abort_cplt_callback()接口返回，用户可在这些函数自定义可执行操作。代码如下：

```
void hal_spi_rx_cplt_callback(spi_handle_t *hspi)
{
    g_rx_done = 1;
}

void hal_spi_tx_cplt_callback(spi_handle_t *hspi)
{
    g_tx_done = 1;
}

void hal_spi_tx_rx_cplt_callback(spi_handle_t *p_spi)
{
    g_rx_done = 1;
    g_tx_done = 1;
}

void hal_spi_abort_cplt_callback(spi_handle_t *hspi)
{
    if (hspi->p_instance == SPIM)
        printf("This is Abort complete Callback in SPIM.\r\n");
    else if (hspi->p_instance == SPIS)
        printf("This is Abort complete Callback in SPIS.\r\n");
}
```

3.12.4.2 测试验证

1. 在工程配置中定义宏MASTER_BOARD，编译并下载至主设备开发板；在工程配置中取消MASTER_BOARD宏定义，编译并下载至从设备开发板。
2. 连接主从设备开发板的对应IO引脚。
3. 将主从设备开发板的串口分别连接至PC端，打开并配置GRUart。
4. 在GRUart的“Receive Data”窗口中将会显示SPI主从设备开发板的数据交互信息。

3.13 UART

UART即通用异步收发传输器（Universal Asynchronous Receiver/Transmitter），可以实现全双工传输和接收。一般用于调试程序或者与其他外围设备交换数据。

以下章节将介绍DMA、Interrupt、阻塞三种方式的UART数据传输。

3.13.1 UART DMA

UART DMA的示例工程用于实现uart的DMA方式收发数据。

UART DMA示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\uart\uart_dma`，其中工程文件在文件夹Keil_5下。

3.13.1.1 代码理解

示例工程流程图如图 3-32所示：

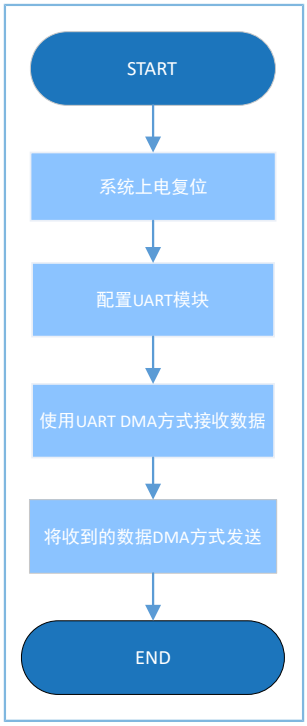

图 3-32 UART DMA工程流程图

1. 配置UART模块。

```
g_uart_handle.p_instance        = SERIAL_PORT_GRP;
g_uart_handle.init.baud_rate    = 115200;
g_uart_handle.init.data_bits    = UART_DATABITS_8;
g_uart_handle.init.stop_bits    = UART_STOPBITS_1;
g_uart_handle.init.parity       = UART_PARITY_NONE;
g_uart_handle.init.hw_flow_ctrl = UART_HWCONTROL_NONE;
g_uart_handle.init.rx_timeout_mode = UART_RECEIVER_TIMEOUT_ENABLE;
hal_uart_deinit(&g_uart_handle);
hal_uart_init(&g_uart_handle);
```

- init.baud_rate：UART波特率。
- init.data_bits：UART数据位。
- init.stop_bits：UART停止位。
- init.parity：UART奇偶校验位。
- init.hw_flow_ctrl：硬件流控使能位。
- init.rx_timeout_mode：接收超时使能位。

2. 调用hal_uart_transmit_dma()接口发送数据，接收结果与状态通过回调函数hal_uart_rx_cplt_callback()返回，用户可在此函数自定义可执行操作。因采用非阻塞方式的接口，需在后续步骤中用while判断是否收发完成。代码如下：

```
void hal_uart_rx_cplt_callback(uart_handle_t *p_uart)
{
    rdone = 1;
    rlen = p_uart->rx_xfer_size - p_uart->rx_xfer_count;
    memcpy(g_tdata, g_rdata, rlen);
    memset(g_rdata, 0, sizeof(g_rdata));
}

void hal_uart_tx_cplt_callback(uart_handle_t *p_uart)
{
    tdone = 1;
}

void hal_uart_error_callback(uart_handle_t *p_uart)
{
    tdone = 1;
    rdone = 1;
}
hal_uart_transmit_dma(&g_uart_handle, g_print_str1, 55);
while (hal_uart_get_state(&g_uart_handle) != HAL_UART_STATE_READY);
hal_uart_transmit_dma(&g_uart_handle, g_print_str2, 8);
while (hal_uart_get_state(&g_uart_handle) != HAL_UART_STATE_READY);
```

3. 调用hal_uart_transmit_dma()发送接收到的数据，发送结果与状态通过回调函数hal_uart_tx_cplt_callback()返回，直到收到的数据为‘0’时，测试结束。代码如下：

```
do {
    rdone = 0;
    hal_uart_receive_dma(&g_uart_handle, g_rdata, sizeof(g_rdata));
    while (0 == rdone);
    tdone = 0;
    hal_uart_transmit_dma(&g_uart_handle, g_tdata, rlen);
    while (0 == tdone);
} while (g_tdata[0] != '0');
```

3.13.1.2 测试验证

1. 用GProgrammer下载*uart_dma_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示通过UART接收与发送的数据信息。

3.13.2 UART Interrupt

UART Interrupt的示例工程实现了UART的中断方式收发数据。

UART Interrupt示例的源代码和工程文件位于SDK_Folder\projects\peripheral\uart\uart_interrupt，其中工程文件在文件夹Keil_5下。

3.13.2.1 代码理解

示例工程流程图如图 3-33所示：

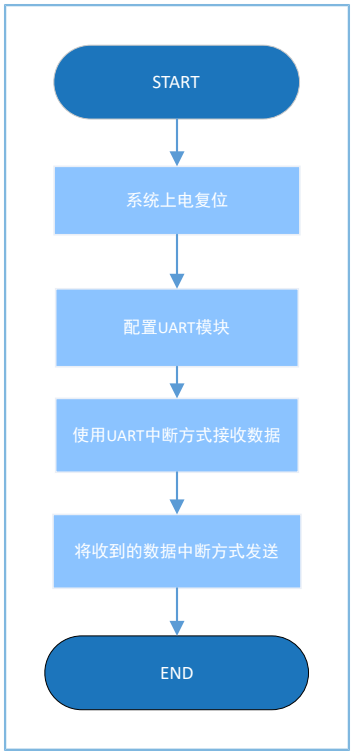

图 3-33 UART Interrupt工程流程图

1. 配置UART模块。

```
g_uart_handle.p_instance = SERIAL_PORT_GRP;
g_uart_handle.init.baud_rate = 115200;
g_uart_handle.init.data_bits = UART_DATABITS_8;
g_uart_handle.init.stop_bits  = UART_STOPBITS_1;
g_uart_handle.init.parity = UART_PARITY_NONE;
g_uart_handle.init.hw_flow_ctrl = UART_HWCONTROL_NONE;
g_uart_handle.init.rx_timeout_mode = UART_RECEIVER_TIMEOUT_ENABLE;
hal_uart_deinit(&g_uart_handle);
hal_uart_init(&g_uart_handle);
```

UART详细配置参数请参考 3.13.1 UART DMA 章节。

2. 调用hal_uart_transmit_it()接口发送数据，接收结果与状态通过回调函数hal_uart_rx_cplt_callback()返回，用户可在此函数自定义可执行操作。因采用非阻塞方式的接口，需在后续步骤中用while判断是否收发完成。

```
hal_uart_transmit_it(&g_uart_handle, (uint8_t *)"\r\nPlease input characters(<126)
                     and end with newline.\r\n", 55);
while (!g_tx_done);
g_tx_done = 0;
hal_uart_transmit_it(&g_uart_handle, (uint8_t *)"Input:\r\n", 8);
while (!g_tx_done);
```

3. 调用hal_uart_receive_it()接口接收数据，接收结果与状态通过回调函数hal_uart_rx_cplt_callback()返回，在接收完成的回调里调用hal_uart_receive_it()继续接收，同时通过hal_uart_transmit_it()发送

接收到的数据，发送结果与状态通过回调函数hal_uart_tx_cplt_callback()返回，直到收到的数据为‘0’时，测试结束。代码如下：

```
void hal_uart_tx_cplt_callback(uart_handle_t *huart)
{
    g_tx_done = 1;
}

void hal_uart_rx_cplt_callback(uart_handle_t *huart)
{
    g_rx_done = 1;
    g_rx_len  = g_uart_handle.rx_xfer_size - g_uart_handle.rx_xfer_count;
    hal_uart_receive_it(&g_uart_handle, rx_data, 128);
}

do {
    while (!g_rx_done);
    g_rx_done = 0;
    g_tx_done = 0;
    hal_uart_transmit_it(&g_uart_handle, rx_data, g_rx_len);
    while (!g_tx_done);
} while (rx_data[0] != '0');
```

3.13.2.2 测试验证

1. 用GProgrammer下载*uart_interrupt_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示通过UART接收与发送的数据信息。

3.13.3 UART TX & RX

UART TX & RX的示例工程实现了以UART的轮询方式收发数据。

UART TX & RX示例的源代码和工程文件位于SDK_Folder\projects\peripheral\uart\uart_tx_rx，其中工程文件在文件夹Keil_5下。

3.13.3.1 代码理解

示例工程流程图如图 3-34所示：

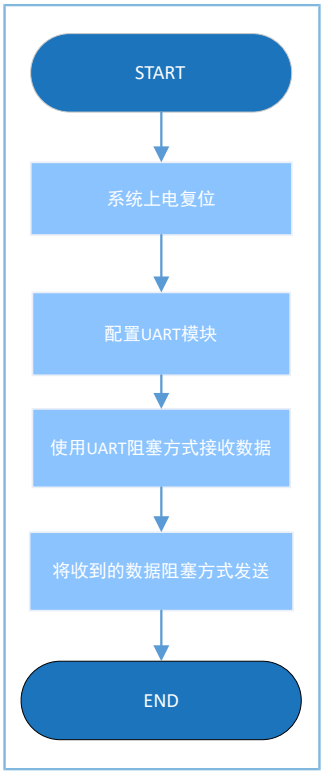

图 3-34 UART TX & RX工程流程图

1. 配置UART模块。

```
g_uart_handle.p_instance  = SERIAL_PORT_GRP;
g_uart_handle.init.baud_rate = 115200;
g_uart_handle.init.data_bits  = UART_DATABITS_8;
g_uart_handle.init.stop_bits  = UART_STOPBITS_1;
g_uart_handle.init.parity  = UART_PARITY_NONE;
g_uart_handle.init.hw_flow_ctrl = UART_HWCONTROL_NONE;
g_uart_handle.init.rx_timeout_mode = UART_RECEIVER_TIMEOUT_ENABLE;
hal_uart_deinit(&g_uart_handle);
hal_uart_init(&g_uart_handle);
```

UART详细配置参数请参考3.13.1 UART DMA。

2. 调用hal_uart_receive()接口用polling方式接收数据，在接收完成后通过hal_uart_transmit()发送接收到的数据，直到收到的数据为‘0’时，测试结束。代码如下：

```
do {
    hal_uart_receive(&g_uart_handle, rdata, 128, 20);
    rlen = g_uart_handle.rx_xfer_size - g_uart_handle.rx_xfer_count;
    hal_uart_transmit(&g_uart_handle, rdata, rlen, 1000);
} while (rdata[0] != '0');
```

3.13.3.2 测试验证

1. 用GProgrammer下载*uart_tx_rx_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。

3. 在GRUart的“Receive Data”窗口中将会显示通过UART接收与发送的数据信息。

3.14 DMA

DMA（Direct Memory Access），即直接存储器存取，是一种快速传送数据的机制，允许在外部设备和存储器之间直接读写数据，CPU除了在数据传输开始和结束时做一点处理外（开始和结束时候要做中断处理），在传输过程中CPU可以进行其他的工作。

3.14.1 DMA Memory to Memory

DMA Memory to Memory的示例工程用于实现DMA方式将数据从一块内存搬运到另一块内存。

DMA Memory to Memory示例的源代码和工程文件位于SDK_Folder\projects\peripheral\dma\dma_memory_to_memory，其中工程文件在文件夹Keil_5下。

3.14.1.1 代码理解

示例工程流程图如图 3-35所示：

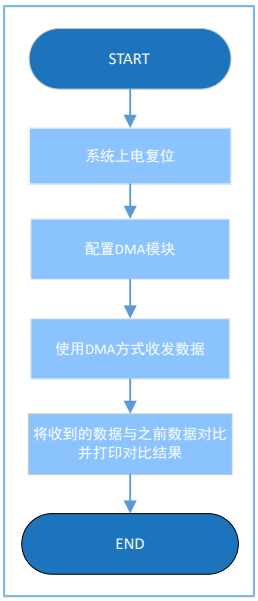

图 3-35 DMA Memory to Memory工程流程图

1. 配置DMA模块。

```
g_dma_handle.channel = DMA_Channel0;
g_dma_handle.init.direction = DMA_MEMORY_TO_MEMORY;
g_dma_handle.init.src_increment = DMA_SRC_INCREMENT;
g_dma_handle.init.dst_increment = DMA_DST_INCREMENT;
g_dma_handle.init.src_data_alignment = DMA_SDATAALIGN_BYTE;
g_dma_handle.init.dst_data_alignment = DMA_DDATAALIGN_BYTE;
g_dma_handle.init.mode = DMA_NORMAL;
g_dma_handle.init.priority = DMA_PRIORITY_LOW;

hal_dma_init(&g_dma_handle);
```

- Channel：DMA通道选择，可选择DMA_Channel0 ～ DMA_Channel7。

- direction：DMA数据传输方向，可选择DMA_MEMORY_TO_MEMORY、DMA_MEMORY_TO_PERIPH、DMA_PERIPH_TO_MEMORY、DMA_PERIPH_TO_PERIPH。
- src_increment：源地址寄存器更新方式，可选择DMA_SRC_INCREMENT、DMA_SRC_DECREMENT、DMA_SRC_NO_CHANGE。
- dst_increment：目的地址寄存器更新方式，可选择DMA_DST_INCREMENT、DMA_DST_DECREMENT、DMA_DST_NO_CHANGE。
- src_data_alignment：源数据宽度，可选择DMA_SDATAALIGN_BYTE、DMA_SDATAALIGN_HALFWORD、DMA_SDATAALIGN_WORD。
- dst_data_alignment：目标数据宽度，可选择DMA_DDATAALIGN_BYTE、DMA_DDATAALIGN_HALFWORD、DMA_DDATAALIGN_WORD。
- mode：DMA的操作模式，可选择DMA_NORMAL、DMA_CIRCULAR。
- priority：DMA通道的软件优先级，可选择DMA_PRIORITY_LOW、DMA_PRIORITY_MEDIUM、DMA_PRIORITY_HIGH。

2. 调用阻塞接口hal_dma_start()开始传输数据，调用阻塞接口hal_dma_poll_for_transfer()完成数据的收发。代码如下：

```
hal_dma_start(&g_dma_handle, (uint32_t)&g_src_data, (uint32_t)&g_dst_data, DMA_DATA_LEN);
hal_dma_poll_for_transfer(&g_dma_handle, 10);
```

3. 调用hal_dma_start_it()以中断方式开始传输数据。传输结果通过dma_tfr_callback()，dma_blk_callback()，dma_err_callback()，dma_abort_callback()用户接口返回，用户可在上述函数内自定义可执行操作。代码如下：

```
void dma_tfr_callback(struct _dma_handle * hdma)
{
    g_tfr_flag = 1;
   //    printf("tfr interrupt\r\n");
}

void dma_blk_callback(struct _dma_handle * hdma)
{
    g_blk_flag = 1;
    //    printf("blk interrupt\r\n");
}

void dma_err_callback(struct _dma_handle * hdma)
{
    g_err_flag = 1;
    //    printf("err interrupt\r\n");
}

void dma_abort_callback(struct _dma_handle * hdma)
```

```
{
    g_abt_flag = 1;
    //    printf("abort interrupt\r\n");
}

hal_dma_start(&g_dma_handle, (uint32_t)&g_src_data, (uint32_t)&g_dst_data, DMA_DATA_LEN);
```

3.14.1.2 测试验证

1. 用GProgrammer下载*dma_m2m_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示DMA搬运数据后数据对比结果。

3.15 QSPI

QSPI（Queued SPI），SPI接口的扩展，增加了队列传输机制，是一种专用的通信接口，连接单、双或四（条数据线）SPI Flash存储介质。一般用于外接Flash存储器。

3.15.1 QSPI Flash

QSPI Flash的示例工程用于实现QSPI方式连接外部Flash，并读写数据。

QSPI Flash示例的源代码和工程文件位于SDK_Folder\projects\peripheral\qspi\qspi_flash，其中工程文件在文件夹Keil_5下。

3.15.1.1 代码理解

示例工程流程图如图 3-36所示：

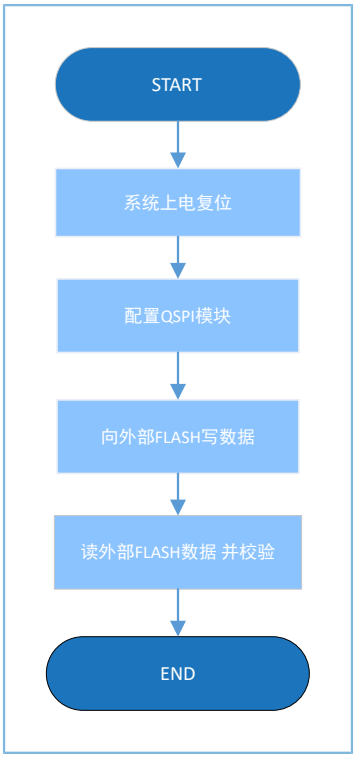

图 3-36 QSPI Flash工程流程图

1. 配置QSPI模块。

```
g_qspi_handle.init.clock_prescaler = SystemCoreClock / 1000000;
g_qspi_handle.init.clock_mode = QSPI_CLOCK_MODE_0;
g_qspi_handle.init.rx_sample_delay = 0;

hal_qspi_deinit(&g_qspi_handle);
if (HAL_OK != hal_qspi_init(&g_qspi_handle))
{
    printf("\r\nFlash initial failed.\r\n");
    return 0;
}
```

- init.clock_prescaler：QSPI时钟频率。
- init.clock_mode：QSPI时钟模式选择，可选择QSPI_CLOCK_MODE_0、QSPI_CLOCK_MODE_1、QSPI_CLOCK_MODE_2、QSPI_CLOCK_MODE_3。

2. 调用hal_qspi_init()初始化QSPI模块。代码如下：

```
hal_qspi_init(&g_qspi_handle)
```

3. 调用SPI_FLASH_Read_Device_ID()获取Flash的设备ID。代码如下：

```
device_id = SPI_FLASH_Read_Device_ID();
printf("Read_Device_ID = 0x%06X\r\n", device_id);
```

4. 调用SPI_FLASH_Page_Program()向Flash写数据。代码如下：

```
printf("Page_Program...\r\n");
```

```
SPI_FLASH_Page_Program(FLASH_PROGRAM_START_ADDR, write_buffer);
printf("Page Program Success.\r\n");
```

5. 调用SPI_FLASH_Read()读取Flash中的数据。代码如下：

```
SPI_FLASH_Read(FLASH_PROGRAM_START_ADDR, read_buffer, FLASH_OPERATION_LENGTH);
```

6. 调用SPI_FLASH_Enable_Quad()使能QSPI的四线模式。代码如下：

```
SPI_FLASH_Enable_Quad();
```

7. 调用如下接口实现QSPI的四线读操作，其源代码均在*spi_flash.c*源文件中，用户可查看其实现细节。

```
SPI_FLASH_Dual_Output_Fast_Read(FLASH_PROGRAM_START_ADDR, read_buffer,
                                FLASH_OPERATION_LENGTH);
printf("Dual_Output_Fast_Read.");
qspi_flash_memcmp(FLASH_PROGRAM_START_ADDR, write_buffer, read_buffer,
                  FLASH_OPERATION_LENGTH);
printf("\r\n");

SPI_FLASH_Dual_IO_Fast_Read(FLASH_PROGRAM_START_ADDR, read_buffer,
                            FLASH_OPERATION_LENGTH);
printf("Dual_IO_Fast_Read.");
qspi_flash_memcmp(FLASH_PROGRAM_START_ADDR, write_buffer, read_buffer,
                  FLASH_OPERATION_LENGTH);
printf("\r\n");

SPI_FLASH_Quad_Output_Fast_Read(FLASH_PROGRAM_START_ADDR, read_buffer,
                                FLASH_OPERATION_LENGTH);
printf("Quad_Output_Fast_Read.");
qspi_flash_memcmp(FLASH_PROGRAM_START_ADDR, write_buffer, read_buffer,
                  FLASH_OPERATION_LENGTH);
printf("\r\n");

SPI_FLASH_Quad_IO_Fast_Read(FLASH_PROGRAM_START_ADDR, read_buffer,
                            FLASH_OPERATION_LENGTH);
printf("Quad_IO_Fast_Read.");
qspi_flash_memcmp(FLASH_PROGRAM_START_ADDR, write_buffer, read_buffer,
                  FLASH_OPERATION_LENGTH);
printf("\r\n");
```

3.15.1.2 测试验证

1. 用GProgrammer下载*qspi_flash_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示QSPI操作Flash的数据信息。

3.15.2 QSPI DMA SPI

QSPI DMA SPI示例实现了采用QSPI接口的外部Flash，读取Flash中的数据并通过SPI Master DMA方式传输数据。

QSPI DMA SPI示例的源代码和工程文件位于SDK_Folder\projects\peripheral\qspi\qspi_flash_dma_spim，其中工程文件在文件夹Keil_5下。

3.15.2.1 代码理解

示例工程流程图如图 3-37所示：

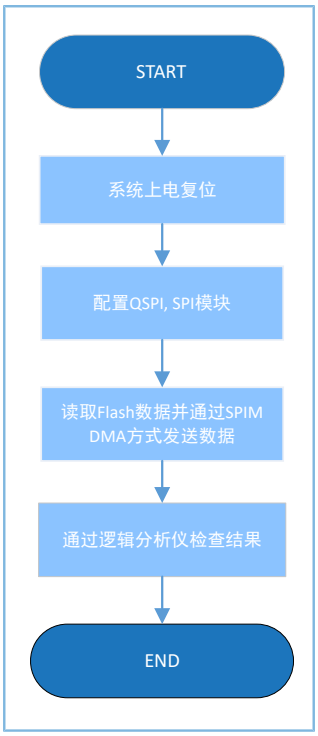

图 3-37 QSPI DMA SPI示例流程图

1. 配置QSPI DMA SPI。

```
g_qspi_handle.init.clock_prescaler = SystemCoreClock / 1000000;
g_qspi_handle.init.clock_mode = QSPI_CLOCK_MODE_3;
g_qspi_handle.init.rx_sample_delay = 0;

hal_qspi_deinit(&g_qspi_handle);
if (HAL_OK != hal_qspi_init(&g_qspi_handle))
{
    printf("\r\nFlash initial failed.\r\n");
    return 0;
}

p_spi_handle->init.data_size          = SPI_DATASIZE_32BIT;
p_spi_handle->init.clock_polarity     = SPI_POLARITY_LOW;
p_spi_handle->init.clock_phase        = SPI_PHASE_1EDGE;
p_spi_handle->init.baudrate_prescaler = SystemCoreClock / 1000000;
```

```
p_spi_handle->init.ti_mode            = SPI_TIMODE_DISABLE;
p_spi_handle->init.slave_select       = SPI_SLAVE_SELECT_0;
hal_spi_deinit(p_spi_handle);
hal_spi_init(p_spi_handle);
```

(1) 配置QSPI模块。

配置QSPI的时钟为1 MHz，配置空闲状态时时钟信号为高电平。

(2) 配置SPI模块。

- init.data_size：SPI数据发送宽度，可选择：SPI_DATASIZE_4BIT ～ SPI_DATASIZE_32BIT。
- init.clock_polarity：SPI空闲状态时的时钟信号，可选择：SPI_POLARITY_LOW、SPI_POLARITY_HIGH。
- init.clock_phase：SPI时钟切换的时间，可选择SPI_PHASE_1EDGE、SPI_PHASE_2EDGE。
- baudrate_prescaler：SPI的时钟选择，此处为SystemCoreClock / 1000000即1 MHz；
- init.ti_mode：SPI TI模式使能，可选择：SPI_TIMODE_DISABLE、SPI_TIMODE_ENABLE。
- init.slave_select：SPI从设备的选择，可选择：SPI_SLAVE_SELECT_0、SPI_SLAVE_SELECT_1、SPI_SLAVE_SELECT_ALL。

2. 调用hal_qspi_init()初始化QSPI模块。代码如下：

```
hal_qspi_init(&g_qspi_handle);
```

3. 调用SPI_FLASH_Read_Device_ID()获取Flash的设备ID。代码如下：

```
device_id = SPI_FLASH_Read_Device_ID();
printf("Read_Device_ID = 0x%06X\r\n", device_id);
```

4. 调用SPI_FLASH_Page_Program()向Flash写数据。代码如下：

```
printf("Page_Program...\r\n");
SPI_FLASH_Page_Program(FLASH_PROGRAM_START_ADDR, flash_buffer);
printf("Page Program Success.\r\n");
```

5. 配置QSPI模式为SPI模式，配置数据位宽为32位，传输目的为EEPROM，输出大小为(FLASH_PAGE_SIZE >> 2) - 1。代码如下：

```
/* Config QSPI in SPI mode, data size is 32bits */
__HAL_QSPI_DISABLE(p_qspi_handle);
ll_spi_set_frame_format(p_qspi_handle->p_instance, LL_SSI_FRF_SPI);
ll_spi_set_data_size(p_qspi_handle->p_instance, LL_SSI_DATASIZE_32BIT);
ll_spi_set_transfer_direction(p_qspi_handle->p_instance, LL_SSI_READ_EEPROM);
ll_spi_set_receive_size(p_qspi_handle->p_instance, (FLASH_PAGE_SIZE >> 2) - 1);
__HAL_QSPI_ENABLE(p_qspi_handle);
```

6. 配置SPI为写模式。代码如下：

```
/* Config SPI in write mode */
```

```
__HAL_SPI_DISABLE(p_spi_handle);
ll_spi_set_transfer_direction(p_spi_handle->p_instance, LL_SSI_SIMPLEX_TX);
__HAL_SPI_ENABLE(p_spi_handle);
```

7. 配置DMA burst length。代码如下：

```
/* Config burst length of DMA */
ll_dma_set_source_burst_length(DMA, p_qspi_handle->p_dma->channel,
                               LL_DMA_SRC_BURST_LENGTH_1);
ll_dma_set_destination_burst_length(DMA, p_qspi_handle->p_dma->channel,
                                    LL_DMA_DST_BURST_LENGTH_4);
```

8. 开始DMA传输并等待传输完成。代码如下：

```
/* Start read SPI flash */
uint32_t cmd_addr = ((uint32_t)SPI_FLASH_CMD_READ << 24) | FLASH_PROGRAM_START_ADDR;
hal_dma_start(p_qspi_handle->p_dma, (uint32_t)& p_qspi_handle->p_instance->DATA, (uint32_t)&
              p_spi_handle->p_instance->DATA, FLASH_PAGE_SIZE >> 2);
p_qspi_handle->p_instance->DATA = cmd_addr;
__HAL_QSPI_ENABLE_DMARX(p_qspi_handle);
__HAL_SPI_ENABLE_DMATX(p_spi_handle);
hal_status_t status = hal_dma_poll_for_transfer(p_qspi_handle->p_dma, 1000);
if (status == HAL_OK)
{
    printf("Transfer finished.\r\n");
}
else
{
    printf("Transfer failed, hal_status = %d\r\n", status);
}
```

3.15.2.2 测试验证

1. 用GProgrammer下载*qspi_flash_dma_spim_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示QSPI FLASH SPI数据的交互数据信息。

3.15.3 QSPI DMA UART

QSPI DMA UART示例实现了采用QSPI接口连接外部Falsh，读取Flash中的数据并通过UART DMA方式传输数据。

QSPI DMA UART示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\qspi\qspi_flash_dma_uart`，其中工程文件在文件夹Keil_5下。

3.15.3.1 代码理解

示例工程流程图如图 3-38所示：

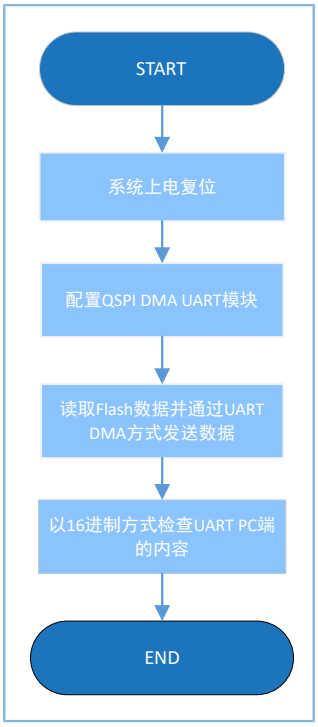

图 3-38 QSPI DMA UART示例流程图

1. 配置QSPI DMA UART。

```
g_qspi_handle.init.clock_prescaler = SystemCoreClock / 1000000;
g_qspi_handle.init.clock_mode = QSPI_CLOCK_MODE_3;
g_qspi_handle.init.rx_sample_delay = 0;

hal_qspi_deinit(&g_qspi_handle);
if (HAL_OK != hal_qspi_init(&g_qspi_handle))
{
    printf("\r\nFlash initial failed.\r\n");
    return 0;
}

uart_handle.p_instance         = UART1;
uart_handle.init.baud_rate     = 115200;
uart_handle.init.data_bits     = UART_DATABITS_8;
uart_handle.init.stop_bits     = UART_STOPBITS_1;
uart_handle.init.parity        = UART_PARITY_NONE;
uart_handle.init.hw_flow_ctrl  = UART_HWCONTROL_NONE;
uart_handle.init.rx_timeout_mode = UART_RECEIVER_TIMEOUT_DISABLE;
hal_uart_deinit(&uart_handle);
hal_uart_init(&uart_handle);
```

(1) 配置QSPI模块。

配置QSPI的时钟为1 MHz，配置空闲状态时的时钟信号为高电平。

(2) 配置UART模块。

UART详细配置信息请参考3.13.1 UART DMA。其中UART配置为UART1。

2. 调用hal_qspi_init()初始化QSPI模块。代码如下：

```
hal_qspi_init(&g_qspi_handle)
```

3. 调用SPI_FLASH_Read_Device_ID()获取Flash的设备ID。代码如下：

```
device_id = SPI_FLASH_Read_Device_ID();
printf("Read_Device_ID = 0x%06X\r\n", device_id);
```

4. 调用SPI_FLASH_Read()读取Flash中的数据。代码如下：

```
SPI_FLASH_Read(FLASH_PROGRAM_START_ADDR, flash_buffer, FLASH_PAGE_SIZE);
for (i = 0; i < FLASH_PAGE_SIZE; i++) {
    if (flash_buffer[i] != 0xFF)
        break;
}

if (i < FLASH_PAGE_SIZE)
{
    printf("Erase Sector...\r\n");
    SPI_FLASH_Sector_Erase(FLASH_PROGRAM_START_ADDR);
    printf("Erase Sector Success.\r\n");
}
```

5. 调用SPI_FLASH_Page_Program()向Flash写数据。代码如下：

```
printf("Page_Program...\r\n");
SPI_FLASH_Page_Program(FLASH_PROGRAM_START_ADDR, flash_buffer);
printf("Page Program Success.\r\n");
```

6. 配置QSPI模式为QSPI模式，配置数据位宽为32位，传输目的为EEPROM，输出大小为(FLASH_PAGE_SIZE >> 2) - 1。代码如下：

```
/* Config QSPI in SPI mode, data size is 32bits */
__HAL_QSPI_DISABLE(p_qspi_handle);
ll_spi_set_frame_format(p_qspi_handle->p_instance, LL_SSI_FRF_SPI);
ll_spi_set_data_size(p_qspi_handle->p_instance, LL_SSI_DATASIZE_8BIT);
ll_spi_set_transfer_direction(p_qspi_handle->p_instance, LL_SSI_READ_EEPROM);
ll_spi_set_receive_size(p_qspi_handle->p_instance, FLASH_PAGE_SIZE - 1);
__HAL_QSPI_ENABLE(p_qspi_handle);
```

7. 配置UART1 FIFO threshold 和burst length。代码如下：

```
/* Config FIFO threshold of UART and burst length of DMA */
ll_uart_set_tx_fifo_threshold(uart_handle.p_instance, LL_UART_TX_FIFO_TH_QUARTER_FULL);
ll_uart_enable_it(uart_handle.p_instance, UART_IT_THRE);

ll_dma_set_source_burst_length(DMA, p_qspi_handle->p_dma->channel,
                               LL_DMA_SRC_BURST_LENGTH_1);
```

```
ll_dma_set_destination_burst_length(DMA, p_qspi_handle->p_dma->channel,
                                    LL_DMA_DST_BURST_LENGTH_8);
```

8. 开始DMA传输并等待传输完成。代码如下：

```
uint32_t cmd_addr = ((uint32_t)SPI_FLASH_CMD_READ << 24) | FLASH_PROGRAM_START_ADDR;
hal_dma_start(p_qspi_handle->p_dma, (uint32_t)& p_qspi_handle->p_instance->DATA,
              (uint32_t)&UART1->RBR_DLL_THR, FLASH_PAGE_SIZE);
p_qspi_handle->p_instance->DATA = (cmd_addr >> 24 ) & 0xff;
p_qspi_handle->p_instance->DATA = (cmd_addr >> 16 ) & 0xff;
p_qspi_handle->p_instance->DATA = (cmd_addr >>  8 ) & 0xff;
p_qspi_handle->p_instance->DATA = (cmd_addr >>  0 ) & 0xff;

__HAL_QSPI_ENABLE_DMARX(p_qspi_handle);
ll_spi_enable_ss(p_qspi_handle->p_instance, LL_SSI_SLAVE0);
hal_status_t status = hal_dma_poll_for_transfer(p_qspi_handle->p_dma, 1000);
if (status == HAL_OK)
{
    printf("Transfer finished.\r\n");
}
else
{
    printf("Transfer failed, hal_status = %d\r\n", status);
}
```

3.15.3.2 测试验证

1. 用GProgrammer下载*qspi_flash_dma_uart_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示QSPI FLASH UART的交互数据信息。

3.16 Timer

Timer，定时器，在指定的时间间隔内反复触发指定事件或者任务。一般用于处理一些定时任务。

3.16.1 Dual Timer

Dual Timer的示例工程实现了以Dual Timer的中断方式定时打印程序中相关信息。

Dual Timer示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\timer\dual_timer`，其中工程文件在文件夹Keil_5下。

3.16.1.1 代码理解

示例工程流程图如图 3-39所示：

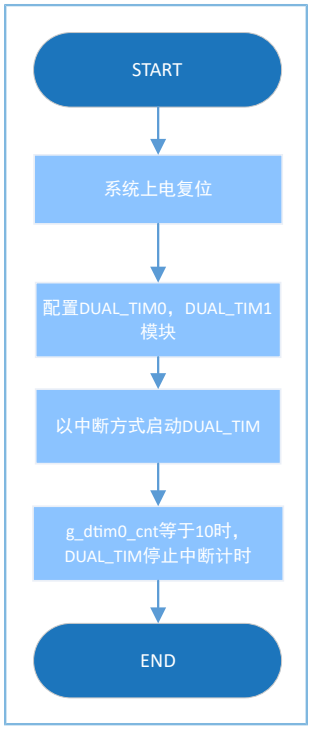

图 3-39 Dual Timer工程流程图

1. 配置Dual Timer模块。

```
g_dual_tim0_handle.p_instance = DUAL_TIMER0;
g_dual_tim0_handle.init.prescaler   = DUAL_TIM_PRESCALER_DIV0;
g_dual_tim0_handle.init.counter_mode = DUAL_TIM_COUNTERMODE_LOOP;
g_dual_tim0_handle.init.auto_reload  = SystemCoreClock - 1;
hal_dual_timer_base_init(&g_dual_tim0_handle);

g_dual_tim1_handle.p_instance = DUAL_TIMER1;
g_dual_tim1_handle.init.prescaler   = DUAL_TIM_PRESCALER_DIV0;
g_dual_tim1_handle.init.counter_mode = DUAL_TIM_COUNTERMODE_LOOP;
g_dual_tim1_handle.init.auto_reload = SystemCoreClock / 100 - 1;
```

- prescaler：预分频器选择，可选择DUAL_TIM_PRESCALER_DIV0，DUAL_TIM_PRESCALER_DIV16，DUAL_TIM_PRESCALER_DIV256。
- counter_mode：计数器模式，可选择DUAL_TIM_COUNTERMODE_LOOP，DUAL_TIM_COUNTERMODE_ONESHOT。
- auto_reload：自动重载值，DUAL_TIMER0选择1s加载一次，DUAL_TIMER1选择10 ms加载一次。

2. 调用hal_dual_timer_base_start_it()启用中断方式计时，auto_reload定时时间通过接口hal_dual_timer_period_elapsed_callback()返回。代码如下：

```
void hal_dual_timer_period_elapsed_callback(dual_timer_handle_t *hdtim)
{
    if (hdtim->p_instance == DUAL_TIMER0)
    {
        printf("\r\nThis is %dth call DUALTIM0.\r\n", g_dtim0_cnt++);
```

```
    }
    else
    {
        g_dtim1_cnt++;
    }
}

hal_dual_timer_base_start_it(&g_dual_tim1_handle);
hal_dual_timer_base_start_it(&g_dual_tim0_handle);
```

3. 调用hal_dual_timer_base_stop_it()停止中断方式计时。代码如下：

```
hal_dual_timer_base_stop_it(&g_dual_tim1_handle);
hal_dual_timer_base_stop_it(&g_dual_tim0_handle);
```

3.16.1.2 测试验证

1. 用GProgrammer下载*dual_timer_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中程序每秒打印一次调试信息，打印10次后程序运行结束。

3.16.2 Timer

Timer的示例工程实现了以Timer的中断方式定时打印程序中相关信息。

Timer示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\timer\timer`，其中工程文件在文件夹Keil_5下。

3.16.2.1 代码理解

示例工程流程图如图 3-40所示：

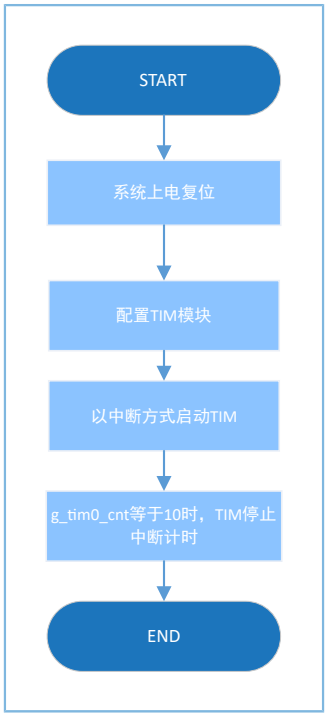

图 3-40 Timer工程流程图

1. 配置Timer模块。

```
g_tim0_handle.p_instance = TIMER0;
g_tim0_handle.init.auto_reload = SystemCoreClock - 1;
hal_timer_base_init(&g_tim0_handle);

g_tim1_handle.p_instance = TIMER1;
g_tim1_handle.init.auto_reload = SystemCoreClock / 100 - 1;
hal_timer_base_init(&g_tim1_handle);
```

auto_reload：自动重载值，TIMER0选择1s加载一次，TIMER1选择10 ms加载一次。

2. 调用hal_timer_base_start_it()启用中断方式计时，auto_reload定时时间到时通过接口hal_timer_period_elapsed_callback()返回。代码如下：

```
void hal_timer_period_elapsed_callback(timer_handle_t *htim)
{
    if (htim->p_instance == TIMER0)
    {
        printf("\r\nThis is %dth call TIMER0.\r\n", g_tim0_cnt++);
    }
    else
    {
        g_tim1_cnt++;
    }
}

hal_timer_base_start_it(&g_tim1_handle);
hal_timer_base_start_it(&g_tim0_handle);
```

3. 调用hal_timer_base_stop_it()结束中断方式计时。代码如下：

```
hal_timer_base_stop_it(&g_tim1_handle);
hal_timer_base_stop_it(&g_tim0_handle);
```

3.16.2.2 测试验证

1. 用GProgrammer下载*timer_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart窗口中程序每秒打印一次调试信息，打印10次后程序运行结束。

3.16.3 Timer Wakeup

Timer Wakeup的示例工程实现了以Dual Timer的中断方式定时唤醒系统。

Timer Wakeup示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\timer\timer_wakeup`，其中工程文件在文件夹Keil_5下。

3.16.3.1 代码理解

示例工程流程图如图 3-41所示：

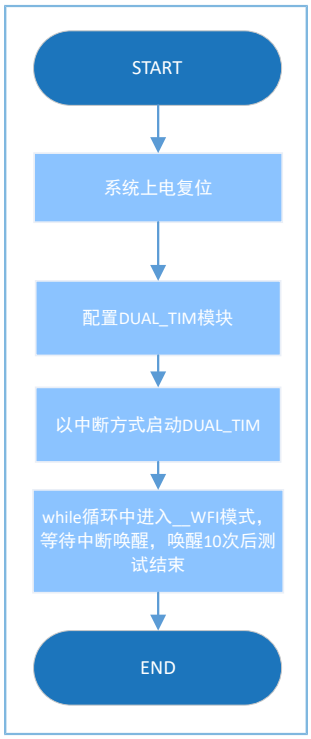

图 3-41 Timer Wakeup工程流程图

1. 配置Dual Timer模块。

```
g_dual_tim0_handle.p_instance = DUAL_TIMER0;
g_dual_tim0_handle.init.prescaler    = DUAL_TIM_PRESCALER_DIV0;
g_dual_tim0_handle.init.counter_mode = DUAL_TIM_COUNTERMODE_LOOP;
g_dual_tim0_handle.init.auto_reload  = SystemCoreClock - 1;
```

```
hal_dual_timer_base_init(&g_dual_tim0_handle);
```

Dual Timer详细配置参数请参考3.16.1 Dual Timer。

2. 调用hal_dual_timer_base_start_it()启用中断方式唤醒系统，auto_reload定时时间到时唤醒系统并通过接口hal_dual_timer_period_elapsed_callback()返回结果。代码如下：

```
void hal_dual_timer_period_elapsed_callback(dual_timer_handle_t *hdtim)
{
    if (hdtim->p_instance == DUAL_TIMER0)
    {
        g_dtim0_cnt++;
    }
}

hal_dual_timer_base_start_it(&g_dual_tim0_handle);
```

3. 系统进入睡眠状态。代码如下：

```
while (g_dtim0_cnt < 10)
{
    printf("\r\nEnter sleep at timer counter = %d\r\n", g_dtim0_cnt);
    SCB->SCR |= 0x04;
    __WFI();
    printf("Wakeup from sleep at timer counter = %d\r\n", g_dtim0_cnt);
}
```

4. 调用hal_dual_timer_base_stop_it()结束中断方式计时。代码如下：

```
hal_dual_timer_base_stop_it(&g_dual_tim0_handle);
```

3.16.3.2 测试验证

1. 用GProgrammer下载*timer_wakeup_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart窗口中程序每秒打印一次调试信息，打印10次后程序运行结束。

3.17 WDT

WDT（Watchdog Timer）一种硬件式计时设备。当系统的主程序发生某些错误事件时，如假死机或未定时喂狗，WDT就会对系统发出重置、重启或关闭的信号，使系统从悬停状态恢复到正常运作状态。一般用于程序异常时重启系统。

3.17.1 WDT Reset

WDT Reset的示例工程实现了以WDT方式定时重启系统。

WDT Reset示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\wdt\wdt_reset`，其中工程文件在文件夹Keil_5下。

3.17.1.1 代码理解

示例工程流程图如图 3-42所示：

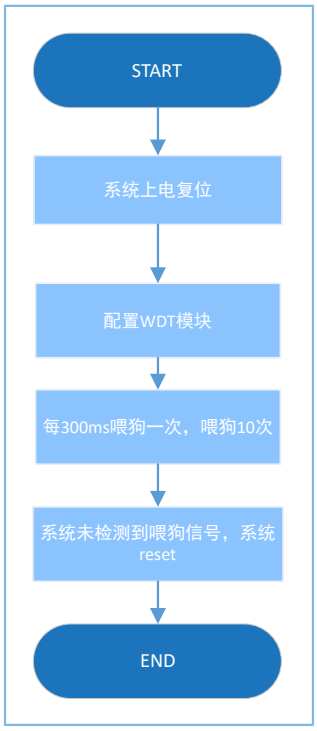

图 3-42 WDT Reset工程流程图

1. 配置WDT模块。

```
g_wdt_handle.init.counter    = SystemCoreClock;
g_wdt_handle.init.reset_mode = WDT_RESET_ENABLE;
hal_wdt_init(&g_wdt_handle);
```

- counter：WDT计数时间，此处设置为1s。
- reset_mode：系统reset使能，此处设置为WDT_RESET_ENABLE。

2. 调用hal_wdt_refresh()进行WDT喂狗。代码如下：

```
hal_wdt_refresh(&g_wdt_handle);
```

3.17.1.2 验证测试

1. 用GProgrammer下载*wdt_reset_sk_r2_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart窗口中程序每300 ms打印一次调试信息，打印10次，系统重启。

3.18 COMP

COMP，比较器，比较器是通过比较两个输入端的电流或电压的大小，在输出端输出不同电压结果的电子元件，当电压高于某个门限时触发中断，常被用于模数转换电路中。

3.18.1 COMP MSIO

COMP MSIO的示例工程实现了两个MSIO引脚外接电压值的比较。

COMP MSIO示例的源代码和工程文件位于SDK_Folder\projects\peripheral\comp\comp_msio_msio，其中工程文件在文件夹Keil_5下。

3.18.1.1 代码理解

示例工程流程图如图 3-43所示：

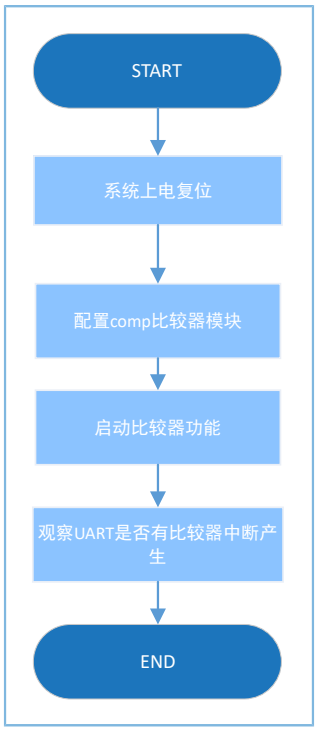

图 3-43 COMP MSIO示例工程流程图

1. 配置COMP模块。

```
g_comp_handle.init.input_source = COMP_INPUT_SRC_IO0;
g_comp_handle.init.ref_source   = COMP_REF_SRC_IO1;
g_comp_handle.init.ref_value    = 0;
hal_comp_deinit(&g_comp_handle);
hal_comp_init(&g_comp_handle);

msio_config.pin = COMP_INPUT_PIN | COMP_VREF_PIN;
msio_config.direction = MSIO_DIRECTION_INPUT;
msio_config.pull = MSIO_PULLUP;
msio_config.mode = MSIO_MODE_ANALOG;
hal_msio_init(&msio_config);
```

- input_source：比较器输入引脚选择，可选择：COMP_INPUT_SRC_IO0 ～ COMP_INPUT_SRC_IO4，对应引脚为MSIO0 – MSIO4。
- ref_source：参考源选择，可选择：COMP_REF_SRC_IO0 ～ COMP_REF_SRC_VREF，对应引脚为MSIO0 – MSIO4（外部参考），VBAT（电池），VREF（内部参考）。

- ref_value：比较器参考输入电压值，此示例工程使用外部的物理COMP_REF_SRC_IO1作为输入，无法确定具体值，此处无需设置或者设置为默认值0。
- 配置COMP_INPUT_PIN（MSIO 0），COMP_VREF_PIN（MSIO 1）为模拟输入引脚。

2. 调用hal_comp_start()启用中断方式比较器，比较结果通过回调函数hal_comp_trigger_callback()返回，用户可在此函数中自定义可执行操作。代码如下：

```
void hal_comp_trigger_callback(comp_handle_t *p_comp)
{
    printf("Comp is triggered.\r\n");
}

hal_comp_start(&g_comp_handle);
```

3.18.1.2 测试验证

1. 用GProgrammer下载*comp_msio_msio_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示比较器的比较结果。

3.18.2 COMP VBAT

COMP VBAT的示例工程实现了电池电压与MSIO引脚外接电压值的比较。

COMP VBAT示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\comp\comp_msio_vbat`，其中工程文件在文件夹Keil_5下。

3.18.2.1 代码理解

示例工程流程图如图 3-44所示：

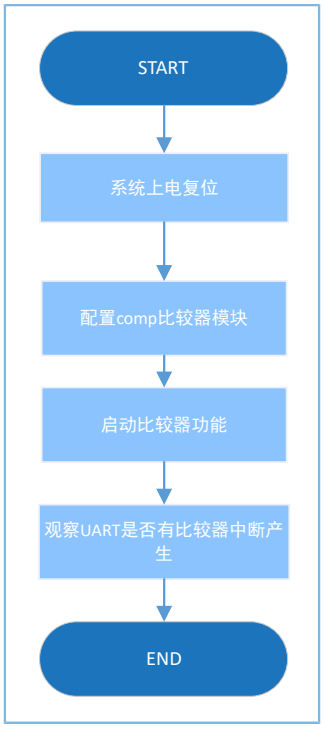

图 3-44 COMP VBAT工程流程图

1. 配置COMP模块。

```
g_comp_handle.init.input_source = COMP_INPUT_SRC_IO0;
g_comp_handle.init.ref_source   = COMP_REF_SRC_VBAT;
g_comp_handle.init.ref_value    = 5;  // Reference = ((ref_value + 1) / 10) * VBATT
hal_comp_deinit(&g_comp_handle);
hal_comp_init(&g_comp_handle);

msio_config.pin = COMP_INPUT_PIN;
msio_config.direction = MSIO_DIRECTION_INPUT;
msio_config.pull = MSIO_PULLUP;
msio_config.mode = MSIO_MODE_ANALOG;
hal_msio_init(&msio_config);
hal_timer_base_init(&g_tim1_handle);
```

- input_source，ref_source：请参考 COMP MSIO > 代码理解 章节。
- ref_value：比较器参考电压，此示例中取值范围在0 ～ 7。值为0时，不符合比较器使用规范。
- 配置COMP_INPUT_PIN（MSIO 0）为模拟输入引脚。

2. 调用hal_comp_start()启用中断方式比较器，比较结果通过回调函数hal_comp_trigger_callback()返回，用户可在此函数自定义操作。代码如下：

```
void hal_comp_trigger_callback(comp_handle_t *p_comp)
{
    printf("Comp is triggered.\r\n");
}
```

```
hal_comp_start(&g_comp_handle);
```

3.18.2.2 测试验证

1. 用GProgrammer下载*comp_msio_msio_fw .bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示比较器的比较结果。

3.18.3 COMP Vref

COMP Vref的示例工程实现了内部参考电压与MSIO引脚外接电压值的比较。

COMP Vref示例的源代码和工程文件位于`SDK_Folder\projects\peripheral\comp\comp_msio_vref`，其中工程文件在文件夹Keil_5下。

3.18.3.1 代码理解

示例工程流程图如图 3-45所示：

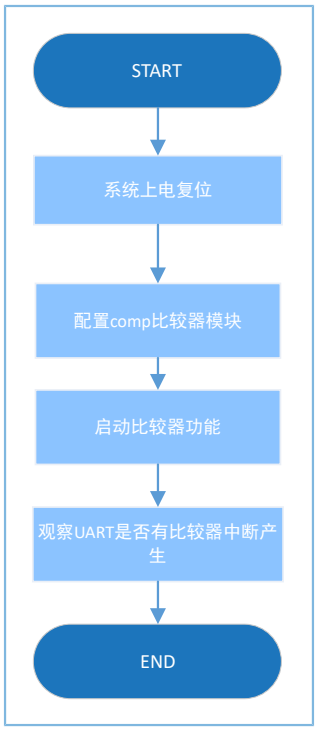

图 3-45 Comp Vref工程流程图

1. 配置COMP模块。

```
g_comp_handle.init.input_source = COMP_INPUT_SRC_IO0;
g_comp_handle.init.ref_source   = COMP_REF_SRC_VREF;
g_comp_handle.init.ref_value    = 30;  // Reference = 30mv * ref_value
hal_comp_deinit(&g_comp_handle);
hal_comp_init(&g_comp_handle);
```

- input_source，ref_source：请参考COMP MSIO > 代码理解。

- ref_value：比较器参考电压，此示例中取值范围0 ～ 63。值为0时，不符合比较器使用规范。

2. 调用hal_comp_start()启用中断方式比较器，比较结果通过回调函数hal_comp_trigger_callback()返回，用户可在此函数自定义操作。代码如下：

```
void hal_comp_trigger_callback(comp_handle_t *p_comp)
{
    printf("Comp is triggered.\r\n");
}

hal_comp_start(&g_comp_handle);
```

3.18.3.2 测试验证

1. 用GProgrammer下载*comp_msio_vref_fw.bin*至开发板。
2. 将开发板串口连接至PC端，打开并配置GRUart。
3. 在GRUart的“Receive Data”窗口中将会显示比较器的比较结果。

4 常见问题

本章描述了在验证及应用外设示例时，可能出现的问题、原因及处理方法。

4.1 SPI常见问题

- 问题描述
  1. CS引脚配置硬件控制时，DMA数据发送异常。
  2. HAL层SPI速率不能达到32 MHz。
- 问题分析
  1. 配置为硬件控制时，一旦TX FIFO为空，硬件会认为数据已发送完毕，将自动拉高CS信号。该问题可能是SPI数据发送期间，被高优先级任务中断并去处理高优先级任务，没有及时填数据到TX FIFO导致。
  2. 在HAL层中TX FIFO为空时CS脚会被自动拉高，导致无法满足需求中的32 MHz时钟。目前不同模式下的速率关系为：DMA > Polling > Interrupt。在 Mirror模式下发送8 bit的数据，采用DMA模式可达16 MHz，Polling模式可达4 ～ 8 MHz，Interrupt模式下则更慢。
- 处理方法
  1. 使用PIN_MUX，将CS引脚配置为普通GPIO进行使能操作。
  2. 采用8 MHz以下的速率进行SPI操作或者使用LL层接口操作寄存器。

4.2 QSPI常见问题

- 问题描述
  1. CS引脚配置硬件控制时，数据发送异常。
- 问题分析
  1. 配置为硬件控制时，一旦TX FIFO为空，硬件会认为数据已发送完毕，将自动拉高CS信号。该问题可能是QSPI数据发送期间，被高优先级任务中断并去处理高优先级任务，没有及时填数据到TX FIFO导致。
  2. 传输采用hal_qspi_transmit()接口，该接口不能支持8 MHz及以上的QSPI传输速率
- 处理方法
  1. 使用PIN_MUX，将CS引脚配置为普通GPIO进行使能操作。
  2. 采用DMA进行QSPI操作。

4.3 ADC常见问题

- 问题描述

1. ADC采用单端输入方式，并利用公式计算采样值，计算结果出现异常。
2. 参考源采用选择错误，导致测试结果错误。
3. 初始化ADC之后，立刻使能ADC获取采样数值，前几个数据会存在错误。

- 问题分析

1. 校验公式使用错误。
2. 参考源错误，现阶段不能将MSIO4、VBAT作为参考源输入。
3. 初始化ADC之后，ADC未能立马进入到最佳的工作状态，如果此时通过ADC采集数据，会导致采集异常。

- 处理方法

1. 使用正常的校验公式：
   - 单端内部： `V = (ADC-offset)/((-1)*slope)`

     offset和slope是保存在芯片内部的对应内部参考电压下的偏移修正值和斜率值。
   - 单端外部： `V = (ADC-offset)/(((-1)*slope*1.0f)/vref)`

     offset和slope是保存在芯片内部的对应外部参考电压1.0 V下的偏移修正值和斜率值。
   - 差分内部： `V = 2*(offset-ADC)/((-1)*slope)`

     offset为在P = N时校准的code值；slope是保存在芯片内部的对应内部参考电压下的斜率值。
   - 差分外部： `V = 2*(offset–2*ADC)/(((-1)*slope*1.0f)/vref)`

     offset在P = N时校准的code值；slope是保存在芯片内部的对应外部参考电压1.0 V下的斜率值。
2. 使用头文件中指定的参考源做为输入。
3. 调用hal_adc_init()接口后，使用hal_gr551x_adc_voltage_intern()或hal_adc_start_dma()，需要抛弃前几个数据（具体个数由ADC采样速度来决定）。

4.4 PWM常见问题

- 问题描述

  PWM 3路通道有独立的占空比调节，但是3路通道设置不同的频率时，无效或者异常。

- 问题分析

  PWM频率共用相同配置参数，不支持不同的频率。

- 处理方法

  将3路通道设置成相同的频率。