

# **RTL87x2D Peripheral Sample Project User Manual**

**v 1.0**

**2021/4/25 2021/3/23**

## 修订历史 (Revision History)

日期	版本	修改	作者	Reviewer
2021/03/23	V1.0	初稿	Echo_Gao	

Realtek Confidential

# 目 录

修订历史 (Revision History) .....	2
目 录 .....	3
表目录 .....	9
图目录 .....	10
1 外设固件概述 .....	11
2 测试环境 .....	12
2.1 RTL87x2D 评估板 .....	12
2.2 Keil .....	12
2.3 下载 (JLINK 仿真器或 MPTool) .....	13
2.4 LogTool .....	14
3 通用输入/输出(GPIO) .....	16
3.1 输出控制闪灯 .....	16
3.1.1 硬件设计 .....	16
3.1.2 软件流程 .....	16
3.2 输入检测-轮询方式 .....	17
3.2.1 硬件设计 .....	17
3.2.2 软件流程 .....	17
3.3 按键检测-中断方式 .....	19
3.3.1 硬件设计 .....	19
3.3.2 软件流程 .....	19
4 通用异步接收发送端(UART) .....	23
4.1 轮询收发 .....	23
4.1.1 硬件设计 .....	23
4.1.2 软件流程 .....	23
4.2 中断接收 .....	25
4.2.1 硬件设计 .....	25
4.2.2 软件流程 .....	25
4.3 UART 唤醒深度低功耗状态 (DLPS) .....	29

4.3.1 硬件设计 .....	29
4.3.2 软件流程 .....	29
4.4 UART 通过 GDMA 接收和发送定长数据 .....	33
4.4.1 硬件设计 .....	34
4.4.2 软件流程 .....	34
4.5 UART 通过 GDMA 接收不定长数据 .....	38
4.5.1 硬件设计 .....	38
4.5.2 软件流程 .....	38
5 定时器和脉冲宽度调制(TIM&PWM).....	44
5.1 脉冲宽度调制输出（PWM） .....	44
5.1.1 硬件设计 .....	44
5.1.2 软件流程 .....	44
5.2 定时器中断 .....	45
5.2.1 硬件设计 .....	46
5.2.2 软件流程 .....	46
5.3 数字信号任意波形输出 .....	47
5.3.1 硬件设计 .....	47
5.3.2 软件流程 .....	47
6 直接内存存取控制器(GDMA) .....	50
6.1 单 Block 传输（Single Block） .....	50
6.1.1 软件流程 .....	50
6.2 多 Block 传输（Multi Block） .....	52
6.2.1 软件流程 .....	52
6.3 分散和收集（scatter&gather） .....	56
6.3.1 软件流程 .....	56
7 模数转换器(ADC) .....	59
7.1 单次采样模式-轮询方式 .....	59
7.1.1 硬件设计 .....	59
7.1.2 软件流程 .....	59
7.2 单次采样模式-中断方式 .....	62

7.2.1 硬件设计 .....	62
7.2.2 软件流程 .....	62
7.3 单次、差分采样模式 .....	65
7.3.1 硬件设计 .....	65
7.3.2 软件流程 .....	65
7.4 单次采样模式-GDMA 搬运采样数据 .....	67
7.4.1 硬件设计 .....	67
7.4.2 软件流程 .....	67
7.5 单次采样模式-唤醒深度低功耗状态 (DLPS) .....	71
7.5.1 硬件设计 .....	71
7.5.2 软件流程 .....	71
7.6 连续采样模式 .....	76
7.6.1 硬件设计 .....	76
7.6.2 软件流程 .....	76
7.7 连续采样模式-GDMA 搬运采样数据 .....	79
7.7.1 硬件设计 .....	79
7.7.2 软件流程 .....	79
8 串型外设接口(SPI) .....	83
8.1 EEPROM 模式读取 ID .....	83
8.1.1 硬件设计 .....	83
8.1.2 软件流程 .....	83
8.2 FullDuplex 模式读取 ID .....	85
8.2.1 硬件设计 .....	86
8.2.2 软件流程 .....	86
8.3 中断方式读取 ID .....	88
8.3.1 硬件设计 .....	88
8.3.2 软件流程 .....	88
8.4 闪存 (Flash) .....	90
8.4.1 硬件设计 .....	90
8.4.2 软件流程 .....	91

8.5 GDMA 方式搬运数据 .....	92
8.5.1 硬件设计 .....	92
8.5.2 软件流程 .....	92
8.6 伪静态随机存储（PSRAM） .....	96
8.6.1 硬件设计 .....	96
8.6.2 软件流程 .....	96
9 内部集成电路(I2C) .....	102
9.1 I2C 通信 .....	102
9.1.1 硬件设计 .....	102
9.1.2 软件流程 .....	102
10 按键扫描(KEYSCAN).....	105
10.1 按键扫描 .....	105
10.1.1 硬件设计 .....	105
10.1.2 软件流程 .....	105
11 红外(IR) .....	110
11.1 红外发送 .....	110
11.1.1 硬件设计 .....	110
11.1.2 软件流程 .....	110
11.2 红外接收 .....	113
11.2.1 硬件设计 .....	113
11.2.2 软件流程 .....	114
11.3 红外编码协议发送 .....	116
11.3.1 硬件设计 .....	117
11.3.2 软件流程 .....	117
11.4 红外学习 .....	120
11.4.1 硬件设计 .....	121
11.4.2 软件流程 .....	121
11.5 红外 GDMA 发送 .....	124
11.5.1 硬件设计 .....	124
11.5.2 软件流程 .....	125

11.6 红外 GDMA 接收 .....	127
11.6.1 硬件设计 .....	127
11.6.2 软件流程 .....	128
12 集成电路内置音频总线(I2S) .....	131
12.1 音频数据传输 .....	131
12.1.1 硬件设计 .....	131
12.1.2 软件流程 .....	131
13 编解码器(CODEC) .....	134
13.1 模拟麦克 .....	134
13.1.1 硬件设计 .....	134
13.1.2 软件流程 .....	134
13.2 数字麦克 .....	136
13.2.1 硬件设计 .....	136
13.2.2 软件流程 .....	136
13.3 脉冲密度调制接口（PDM） .....	138
13.3.1 硬件设计 .....	139
13.3.2 软件流程 .....	139
14 三线 SPI(SPI-3WIRE) .....	142
14.1 三线 SPI 通信 .....	142
14.1.1 硬件设计 .....	142
14.1.2 软件流程 .....	142
15 正交解调(QDEC) .....	145
15.1 相位检测 .....	145
15.1.1 硬件设计 .....	145
15.1.2 软件流程 .....	145
16 8080 并口(IF8080) .....	148
16.1 自动 GDMA 刷屏模式 .....	148
16.1.1 硬件设计 .....	148
16.1.2 软件流程 .....	148
17 实时时钟(RTC) .....	155

---

17.1 滴答定时器 .....	155
17.1.1 软件流程 .....	155
17.2 计数溢出 .....	156
17.2.1 软件流程 .....	156
17.3 定时比较器（闹钟） .....	157
17.3.1 软件流程 .....	157
17.4 分频比较器 .....	158
17.4.1 软件流程 .....	158
18 低功耗比较器(LPC) .....	161
18.1 电压检测 .....	161
18.1.1 硬件设计 .....	161
18.1.2 软件流程 .....	161
18.2 电压比较计数器 .....	163
18.2.1 硬件设计 .....	163
18.2.2 软件流程 .....	163
18.3 电压检测唤醒低功耗状态 (DLPS) .....	164
18.3.1 硬件设计 .....	164
18.3.2 软件流程 .....	165
19 看门狗(WDG) .....	169
19.1 看门狗按键 .....	169
19.1.1 软件流程 .....	169
19.2 看门狗定时器 .....	170
19.2.1 软件流程 .....	170
参考文献 .....	172

## 表目录

表 1-1 外设所有缩写定义.....	<u>11</u> <u>44</u>
表 2-1 公用硬件环境.....	<u>12</u> <u>42</u>
表 2-2 公用测试软件.....	<u>12</u> <u>42</u>

## 图目录

图 2-1 Keil 工具链版本信息 .....	<u>12</u> <u>12</u>
图 2-2 Keil 设置流程示意图 .....	<u>13</u> <u>13</u>
图 2-3 UART 跳线 .....	<u>14</u> <u>14</u>
图 2-4 MPTool 操作流程示意图 .....	<u>14</u> <u>14</u>
图 2-5 LOG 跳线 .....	<u>15</u> <u>15</u>
图 2-6 DebugAnalyser 操作流程示意图 .....	<u>15</u> <u>15</u>
图 3-1 LED 驱动电路 .....	<u>16</u> <u>16</u>
图 3-2 按键驱动电路 .....	<u>19</u> <u>19</u>
图 5-1 GPIO 输出 PWM 波形 .....	49
图 5-2 GPIO 输出 PWM 波形 .....	49
图 10-1 外接矩阵键盘 .....	105
图 11-1 IR 发送数据编码 .....	113
图 11-2 IR 发送数据编码 .....	120
图 11-3 IR-GDMA 发送数据编码 .....	127
图 12-1 I2S 输出波形 .....	133
图 15-1 QDEC 连接图 .....	<u>145</u> <u>145</u>
图 15-2 QDEC 相位转移图 .....	<u>145</u> <u>145</u>

# 1 外设固件概述

表 1-1 外设所有缩写定义

缩写	外设/单元
GPIO	通用输入输出
UART	通用异步接收发送端
TIM(PWM)	定时器(脉冲宽度调制)
GDMA	直接内存存取控制器
ADC	模数转换器
SPI	串型外设接口
I2C	内部集成电路
KEYSCAN	按键扫描
IR	红外
I2S	集成电路内置音频总线
CODEC	编解码器
SPI-3WIRE	三线 SPI
QDEC	正交解调
IF8080	8080 并行接口
RTC	实时时钟
LPC	低功耗比较器
WDG	看门狗

## 2 测试环境

Peripheral Sample Project 的工程目录: \HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ ...。

在演示、测试 Peripheral Sample Project 过程中公用的硬件设备、测试软件如表所示:

表 2-1 公用硬件环境

硬件环境	
1	RTL87x2D 评估板
2	JLINK 仿真器
3	MPTool

表 2-2 公用测试软件

软件环境	
1	Keil MDK-ARM v5
2	LogTool

某些工程使用到专用的硬件设备、测试软件，将在具体章节介绍。

### 2.1 RTL87x2D 评估板

Peripheral Sample Project 是在 RTL87x2D 评估板上进行演示、测试的，RTL87x2D 评估板由母板和子板组成，本测试使用的母版：EVB MotherBoard 2V0，子板 RTL8762DW QFN56 EVB V1.0。

RTL87x2D 评估板的使用请参考：RTL8762D Evaluation Board User Manual CN\_V1.0<sup>[1]</sup>。

### 2.2 Keil

Peripheral Sample Project 中所有工程能够通过 Keil Microcontroller Development Kit(MDK)编译、使用。本测试使用的工具链版本信息如图所示，建议使用以下版本或更高版本。



图 2-1 Keil 工具链版本信息

使用 Keil 前，需要进行设置，设置流程如图所示：

- 点击工具栏上的 ，或在菜单中进入 Project > Options，点击 Debug 页面。选择 J-LINK/J-TRACE Cortex，然后点击“Settings”：
- J-Link Port 选择 SW。如果硬件连接是正确的，CPU 就能在 SW Device 列表中识别出来。
- 选择“Flash Download”页面，修改“RAM for Algorithm”的起始地址为 0x00200000，大小为 0x4000。如果已经有下载算法就先删除。点击“Add”添加 RTL8763Bx\_FLASH\_8MB.FLM 和 RTL876x\_LOG\_TRACE\_16MB（在 sdk\tool\flash 目录下，需将这 2 个文件拷贝到 Keil 安装目录）。

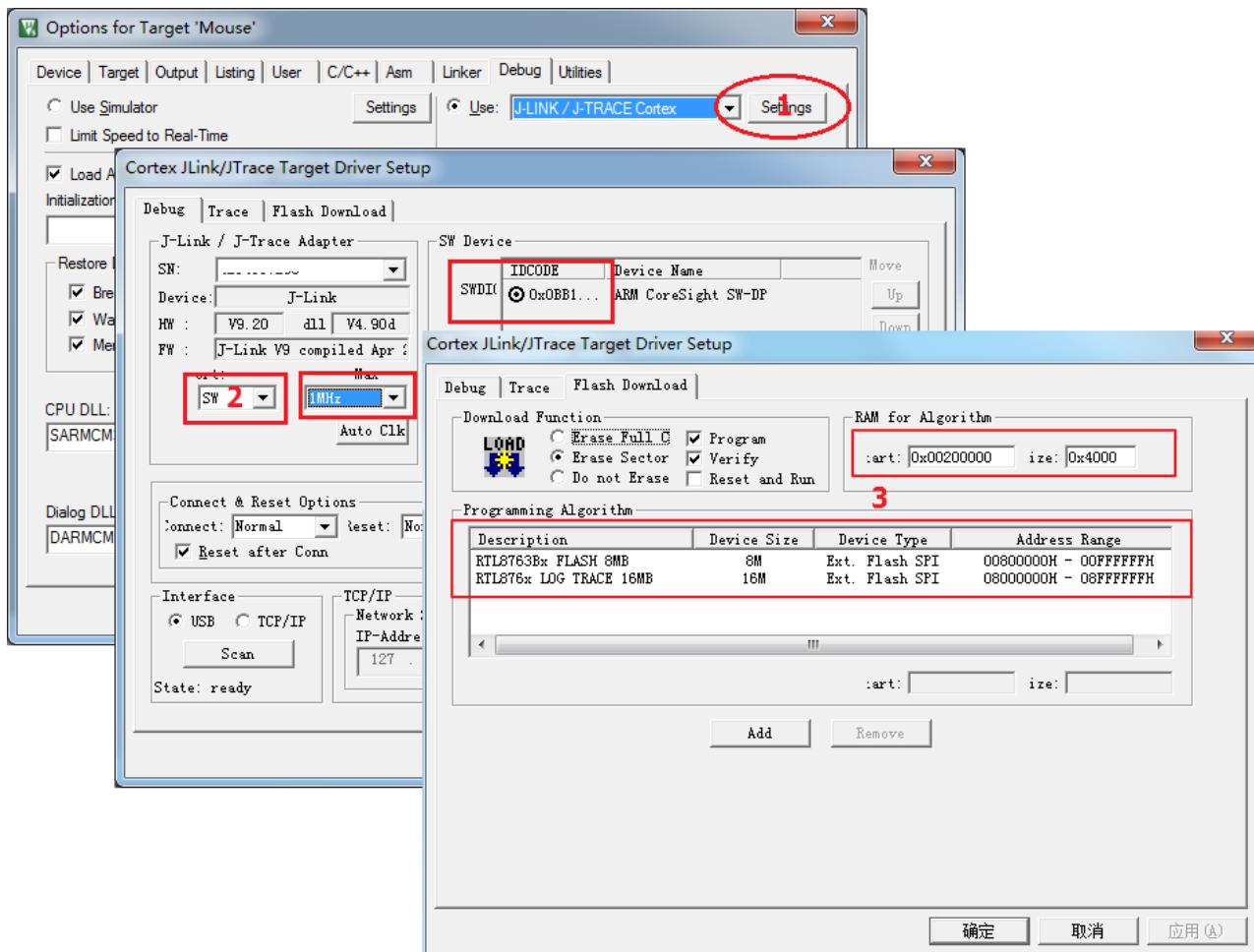


图 2-2 Keil 设置流程示意图

Keil 的使用请参考：RTL8762D SDK User Guide CN<sup>[2]</sup>。

## 2.3 下载 (JLINK 仿真器或 MPTool)

可以使用 J-Link 仿真器搭配 Keil MDK 进行下载、调试，详情请参考：RTL8762D SDK User Guide CN<sup>[2]</sup>。

除了 J-Link 仿真器，可以使用 MPTool 进行下载：

- 连接 EVB 的 UART 到 PC 端，跳线连接方式如图所示：

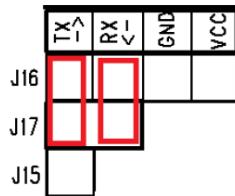


图 2-3 UART 跳线

- 导入 flash map.ini 文件，设置 Config 参数、选择 Secure boot、OTA Header、Patch 等文件；
- 导入编译生成的 APP 镜像文件；
- 探测并打开 UART 端口；
- 点击“Download”按钮开始下载。MP Tool 操作流程如图所示：

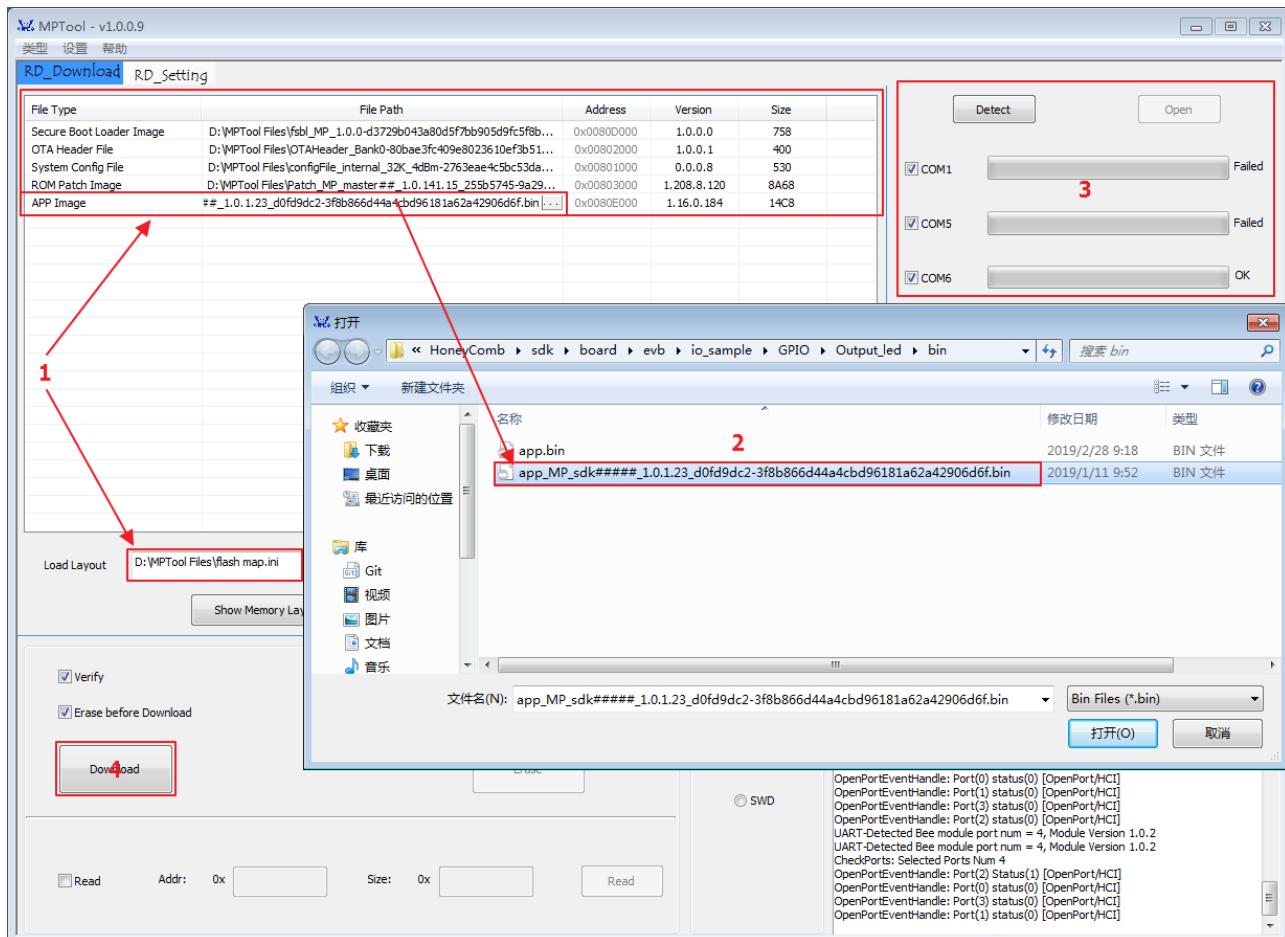


图 2-4 MPTool 操作流程示意图

MPTool 的使用请参考： RTL8762D MP Tool User Guide CN<sup>[3]</sup>。

## 2.4 LogTool

代码下载到 EVB 上，可以使用 LogTool: DebugAnalyser 软件打印 LOG 信息：

- 连接 EVB 的 LOG 引脚到 PC 端，跳线连接方式如图所示：

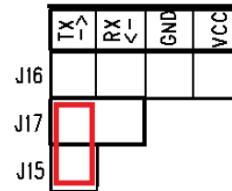


图 2-5 LOG 跳线

- 设置解析输入、解析输出、log 等；
- 配置 UART 参数、加载并导入编译生成的 Trace 文件（加载错误的 Trace 文件，log 信息会出错）；
- Start（开始解析），在 log 窗口会打印 log 信息。DebugAnalyser 的操作流程如图所示。

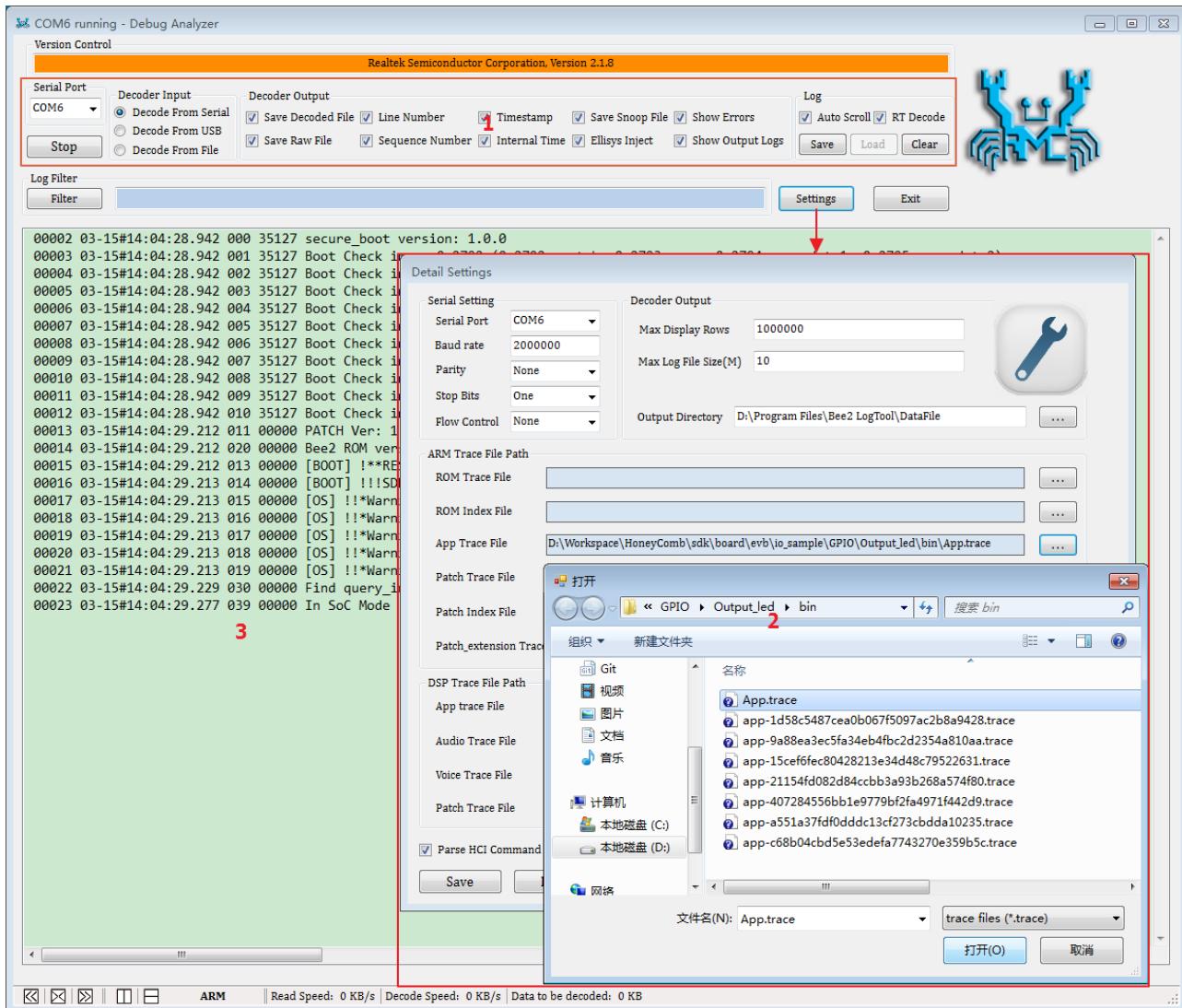


图 2-6 DebugAnalyser 操作流程示意图

LogTool 的使用请参考：DebugAnalyzer User Guide<sup>[4]</sup>。

## 3 通用输入/输出(GPIO)

### 3.1 输出控制闪灯

通过控制 GPIO 输出高低电平，实现 LED 闪烁功能。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ GPIO\ Output\_led。

#### 3.1.1 硬件设计

硬件连接：P0\_1 -> LED0。

在 EVB 上，使用跳帽短接 J24，连接 P0\_1 和 LED0，LED 驱动电路如图 3-1 所示。

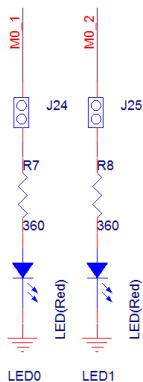


图 3-1 LED 驱动电路

#### 3.1.2 软件流程

引脚定义，GPIO\_GetPin 用于得到 PAD 对应的 GPIO。

```
#define GPIO_OUTPUT_PIN_0      P0_1
#define GPIO_PIN_OUTPUT        GPIO_GetPin(GPIO_OUTPUT_PIN_0)
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、无内部上拉、输出使能、输出高；

配置 PINMUX：分配引脚为 GPIO 功能。

```
board_gpio_init(void)
{
    Pad_Config(GPIO_OUTPUT_PIN_0, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_NONE,
    PAD_OUT_ENABLE, PAD_OUT_HIGH);
    Pinmux_Config(GPIO_OUTPUT_PIN_0, DWGPIO);
}
```

使能 GOIO 时钟；

初始化 GPIO 外设：

- GPIO\_Pin 设置为 GPIO\_PIN\_OUTPUT; GPIO\_Mode 设置为 GPIO\_Mode\_OUT;
- GPIO\_ITCmd 设置为 DISABLE, 即不配置 GPIO 中断模式。

```
void driver_gpio_init(void)
{
    RCC_PeriphClockCmd(APBPeriph_GPIO, APBPeriph_GPIO_CLOCK, ENABLE);
    GPIO_InitTypeDef GPIO_InitStruct;
    GPIO_StructInit(&GPIO_InitStruct);
    GPIO_InitStruct.GPIO_Pin      = GPIO_PIN_OUTPUT;
    GPIO_InitStruct.GPIO_Mode    = GPIO_Mode_OUT;
    GPIO_InitStruct.GPIO_ITCmd  = DISABLE;
    GPIO_Init(&GPIO_InitStruct);
}
```

循环控制 GPIO 输出高低电平。

```
while (1)
{
    /* Light up LED0 */
    GPIO_WriteBit(GPIO_PIN_OUTPUT, (BitAction)(1));
    for (uint32_t i = 0; i < 100000; i++);
    /* Light off LED0 */
    GPIO_WriteBit(GPIO_PIN_OUTPUT, (BitAction)(0));
    for (uint32_t i = 0; i < 100000; i++);
}
```

实验现象：LED0 闪烁。

## 3.2 输入检测-轮询方式

实现 GPIO 输入功能，检测 GPIO 输入的高低电平信号。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ GPIO\ Input\_polling。

### 3.2.1 硬件设计

硬件连接：P2\_2 -> 输入引脚。

在 EVB 上，使用杜邦线短接 P2\_2 和 VCC 检测输入高电平；短接 P2\_2 和 GND 检测输入低电平。

### 3.2.2 软件流程

引脚定义：

```
#define GPIO_INPUT_PIN_0          P2_2  
#define GPIO_PIN_INPUT           GPIO_GetPin(GPIO_INPUT_PIN_0)
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出失能、输出高；

配置 PINMUX：分配引脚为 GPIO 功能。

```
void board_gpio_init(void)  
{  
    Pad_Config(GPIO_INPUT_PIN_0, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_UP,  
    PAD_OUT_DISABLE, PAD_OUT_HIGH);  
    Pinmux_Config(GPIO_INPUT_PIN_0, DWGPIO);  
}
```

使能 GPIO 时钟；

初始化 GPIO 外设：

- GPIO\_Pin 设置为 GPIO\_PIN\_INPUT；GPIO\_Mode 设置为 GPIO\_Mode\_IN；
- GPIO\_ITCmd 设置为 DISABLE，即不配置 GPIO 中断模式。

```
void driver_gpio_init(void)  
{  
    RCC_PeriphClockCmd(APBPeriph_GPIO, APBPeriph_GPIO_CLOCK, ENABLE);  
    GPIO_InitTypeDef GPIO_InitStruct;  
    GPIO_InitStructInit(&GPIO_InitStruct);  
    GPIO_InitStruct.GPIO_Pin = GPIO_PIN_INPUT;  
    GPIO_InitStruct.GPIO_Mode = GPIO_Mode_IN;  
    GPIO_InitStruct.GPIO_ITCmd = DISABLE;  
    GPIO_Init(&GPIO_InitStruct);  
}
```

循环检测 P2\_2 电平状态。

```
while (1)  
{  
    /* Read GPIO input value: gpio_input_data */  
    uint8_t gpio_input_data = GPIO_ReadInputDataBit(GPIO_PIN_INPUT);  
    for (uint32_t i = 0; i < 100000; i++)  
    {  
        /** User code here.  
         * for example: print the value of gpio_input_data;  
         */  
    }  
}
```

### 3.3 按键检测-中断方式

实现检测 GPIO 输入，通过 GPIO 中断向 app\_task 发送 GPIO 消息事件，app\_task 检测到消息事件，在 app 层解析 GPIO 消息，执行用户程序：在 DebugAnalyser 工具上，打印按键信息。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\GPIO\Input\_key。

#### 3.3.1 硬件设计

硬件连接：P4\_0 -> KEY0。

在 EVB 上，按键驱动电路如图 3-2 所示。

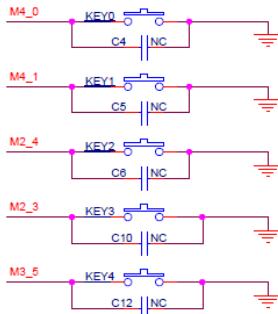


图 3-2 按键驱动电路

#### 3.3.2 软件流程

引脚定义：P4\_0 对应 GPIO28，定义 GPIO28\_IRQn 中断通道和 GPIO28\_Handler 中断处理函数。

```
#define KEY0 P4_0
#define GPIO_INPUT_PIN_0 KEY0
#define GPIO_PIN_INPUT GPIO_GetPin(GPIO_INPUT_PIN_0)
#define GPIO_PIN_INPUT_IRQN GPIO28_IRQn
#define GPIO_Input_Handler GPIO28_Handler
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出失能、输出高；

配置 PINMUX：分配引脚为 GPIO 功能。

```
void board_gpio_init(void)
{
    Pad_Config(GPIO_INPUT_PIN_0, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_UP,
    PAD_OUT_DISABLE, PAD_OUT_HIGH);
    Pinmux_Config(GPIO_INPUT_PIN_0, DWGPIO);
}
```

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,  
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，对 GPIO 外设进行初始化：

- GPIO\_ITCmd 设置为 ENABLE，即配置 GPIO 中断模式；
- GPIO\_ITTrigger 设置为 GPIO\_INT\_Trigger\_EDGE，即边沿触发中断；
- GPIO\_ITPolarity 设置为 GPIO\_INT\_POLARITY\_ACTIVE\_LOW，即下降沿触发；
- ITDebounce 设置为 GPIO\_INT\_DEBOUNCE\_ENABLE，即使能 GPIO 中断去抖动；
- GPIO\_DebounceTime 设置为 10，即去抖时间设置为 10ms；

初始化中断：设置中断优先级，并使能 GPIO\_PIN\_INPUT\_IRQN 通道；

取消屏蔽 GPIO 外部中断；使能 GPIO 中断。

```
void driver_gpio_init(void)  
{  
    .....  
  
    GPIO_InitTypeDef GPIO_InitStruct;  
    GPIO_StructInit(&GPIO_InitStruct);  
  
    GPIO_InitStruct.GPIO_Pin      = GPIO_PIN_INPUT;  
    GPIO_InitStruct.GPIO_Mode     = GPIO_Mode_IN;  
    GPIO_InitStruct.GPIO_ITTrigger = GPIO_INT_Trigger_EDGE;  
    GPIO_InitStruct.GPIO_ITPolarity = GPIO_INT_POLARITY_ACTIVE_LOW;  
    GPIO_InitStruct.GPIO_ITDebounce = GPIO_INT_DEBOUNCE_ENABLE;  
    GPIO_InitStruct.GPIO_DebounceTime = 10; /* unit:ms , can be 1~64 ms */  
    GPIO_Init(&GPIO_InitStruct);  
  
    NVIC_InitTypeDef NVIC_InitStruct;  
    NVIC_InitStruct.NVIC IRQChannel = GPIO_PIN_INPUT_IRQN;  
    NVIC_InitStruct.NVIC IRQChannelPriority = 3;  
    NVIC_InitStruct.NVIC IRQChannelCmd = ENABLE;  
    NVIC_Init(&NVIC_InitStruct);  
  
    GPIO_MaskINTConfig(GPIO_PIN_INPUT, DISABLE);  
    GPIO_INTConfig(GPIO_PIN_INPUT, ENABLE);  
}
```

开始任务调度。

```
os_sched_start();
```

当按键按下（P4\_0 由高电平变为低电平）时，检测到 GPIO 下降沿，触发 GPIO 中断，进入中断处理

函数 GPIO\_Input\_Handler:

- 关 GPIO 中断，屏蔽 GPIO 中断；
- 定义消息类型 IO\_MSG\_TYPE\_GPIO，发送 msg 给 task；
- 清除 GPIO 中断挂起位，取消屏蔽 GPIO 中断，使能 GPIO 中断。

```
void GPIO_Input_Handler(void)
{
    GPIO_INTConfig(GPIO_PIN_INPUT, DISABLE);
    GPIO_MaskINTConfig(GPIO_PIN_INPUT, ENABLE);

    T_IO_MSG int_gpio_msg;
    int_gpio_msg.type = IO_MSG_TYPE_GPIO;
    int_gpio_msg.subtype = 0;
    if (false == app_send_msg_to_apptask(&int_gpio_msg))
        .....
    GPIO_ClearINTPendingBit(GPIO_PIN_INPUT);
    GPIO_MaskINTConfig(GPIO_PIN_INPUT, DISABLE);
    GPIO_INTConfig(GPIO_PIN_INPUT, ENABLE);
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

```
void app_main_task(void *p_param)
{
    .....
    while (true)
    {
        .....
        if (os_msg_recv(io_queue_handle, &io_msg, 0) == true)
        {
            app_handle_io_msg(io_msg);
        }
    }
}
```

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_GPIO，执行用户程序。

```
void app_handle_io_msg(T_IO_MSG io_msg)
{
    uint16_t msg_type = io_msg.type;
```

```
switch (msg_type)
{
    .....
    case IO_MSG_TYPE_GPIO:
        {
            APP_PRINT_INFO0("[app] app_handle_io_msg: GPIO input msg.");
        }
        break;
    default:
        break;
}
```

按下 KEY0，在 DebugAnalyser 工具上，打印按键信息。

## 4 通用异步接收发送端(UART)

专用硬件设备：FT232 usb 转串口

专用测试软件：串口调试工具

### 4.1 轮询收发

通过 UART 实现 S-Bee2 与 PC 端的数据通信。PC 端发送数据，S-Bee2 收到数据并回相同数据给 PC 端，PC 端收到 S-Bee2 回的数据。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\UART\Polling。

#### 4.1.1 硬件设计

硬件连接：P3\_0 -> RX， P3\_1 -> TX。

EVB 外接 FT232 模块，连接 P3\_0 和 FT232 的 RX， P3\_1 和 FT232 的 TX。

#### 4.1.2 软件流程

引脚定义：

```
#define UART_TX_PIN          P3_0  
#define UART_RX_PIN          P3_1
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出失能、输出高；

配置 PINMUX：分配引脚分别为 UART0\_TX、UART0\_RX 功能。

```
void board_uart_init(void)  
{  
    Pad_Config(UART_TX_PIN,      PAD_PINMUX_MODE,      PAD_IS_PWRON,      PAD_PULL_UP,  
    PAD_OUT_DISABLE, PAD_OUT_HIGH);  
    Pad_Config(UART_RX_PIN,      PAD_PINMUX_MODE,      PAD_IS_PWRON,      PAD_PULL_UP,  
    PAD_OUT_DISABLE, PAD_OUT_HIGH);  
    Pinmux_Config(UART_TX_PIN, UART0_TX);  
    Pinmux_Config(UART_RX_PIN, UART0_RX);  
}
```

使能 UART0 时钟；

初始化 UART0 外设：

- 设置 div、ovsr、ovsr\_adj，波特率 115200；
- parity 设置为 UART\_PARITY\_NO\_PARTY，无奇偶校验；

- stopBits 设置为 UART\_STOP\_BITS\_1, 停止位 1 位;
- wordLen 设置为 UART\_WROD\_LENGTH\_8BIT, 数据长度 8bits;
- rxTriggerLevel 设置为 16, FIFO 阈值为 16;
- idle\_time 设置为 UART\_RX\_IDLE\_2BYTE, 即当前波特率下接收 2 个字节的时间。

```
void driver_uart_init(void)
{
    RCC_PeriphClockCmd(APBPeriph_UART0, APBPeriph_UART0_CLOCK, ENABLE);
    UART_InitTypeDef UART_InitStruct;
    UART_StructInit(&UART_InitStruct);

    UART_InitStruct.div          = BaudRate_Table[BAUD_RATE_115200].div;
    UART_InitStruct.ovsr         = BaudRate_Table[BAUD_RATE_115200].ovsr;
    UART_InitStruct.ovsr_adj    = BaudRate_Table[BAUD_RATE_115200].ovsr_adj;
    UART_InitStruct.parity      = UART_PARITY_NO_PARTY;
    UART_InitStruct.stopBits    = UART_STOP_BITS_1;
    UART_InitStruct.wordLen     = UART_WROD_LENGTH_8BIT;
    UART_InitStruct.rxTriggerLevel = 16;           //1~29
    UART_InitStruct.idle_time   = UART_RX_IDLE_2BYTE;    //idle interrupt wait time
    UART_Init(UART, &UART_InitStruct);
}
```

发送字符串: "## Uart demo polling read uart data ##\r\n", 实现多字节连续发送。

```
void uart_senddata_continuous(UART_TypeDef *UARTx, const uint8_t *pSend_Buf, uint16_t vCount)
{
    uint8_t count;
    while (vCount / UART_TX_FIFO_SIZE > 0)
    {
        while (UART_GetFlagState(UARTx, UART_FLAG_THR_EMPTY) == 0);
        for (count = UART_TX_FIFO_SIZE; count > 0; count--)
        {
            UARTx->RB_THR = *pSend_Buf++;
        }
        vCount -= UART_TX_FIFO_SIZE;
    }
    while (UART_GetFlagState(UARTx, UART_FLAG_THR_EMPTY) == 0);
    while (vCount--)
    {

```

```
    UARTx->RB_THR = *pSend_Buf++;  
}  
}
```

检测接收到数据状态，状态为 SET 时，接收数据，并发送接收的数据。

```
while (1)  
{  
    if (UART_GetFlagState(UART, UART_FLAG_RX_DATA_RDY) == SET)  
    {  
        rx_byte = UART_ReceiveByte(UART);  
        UART_SendByte(UART, rx_byte);  
    }  
}
```

使用串口调试助手，发送数据，S-Bee2 收到数据并回相同数据，串口调试助手收到 S-Bee2 回的数据。

## 4.2 中断接收

通过 UART 实现 S-Bee2 与 PC 端的数据通信。PC 端发送数据，S-Bee2 收到数据触发中断，在中断处理函数中接收数据，并传输相同数据给 PC 端，PC 端收到 S-Bee2 回的数据。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ UART\ Interrupt

### 4.2.1 硬件设计

硬件连接：P3\_0 -> RX， P3\_1 -> TX。

EVB 外接 FT232 模块，连接 P3\_0 和 FT232 的 RX， P3\_1 和 FT232 的 TX。

### 4.2.2 软件流程

引脚定义：

```
#define UART_TX_PIN          P3_0  
#define UART_RX_PIN          P3_1
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出失能、输出高；

配置 PINMUX：分配引脚分别为 UART0\_TX、UART0\_RX 功能。

```
void board_uart_init(void)  
{  
    Pad_Config(UART_TX_PIN,      PAD_PINMUX_MODE,      PAD_IS_PWRON,      PAD_PULL_UP,  
    PAD_OUT_DISABLE, PAD_OUT_HIGH);  
    Pad_Config(UART_RX_PIN,      PAD_PINMUX_MODE,      PAD_IS_PWRON,      PAD_PULL_UP,
```

```
PAD_OUT_DISABLE, PAD_OUT_HIGH);  
Pinmux_Config(UART_TX_PIN, UART0_TX);  
Pinmux_Config(UART_RX_PIN, UART0_RX);  
}
```

使能 UART0 时钟；

初始化 UART0 外设：

➤ 设置 div、ovsr、ovsr\_adj，波特率 115200；

➤ 使能 UART 接收中断 (UART\_INT\_RD\_AVA) 和 UART 接收空闲中断 (UART\_INT\_RX\_IDLE)；

初始化中断：设置中断优先级，并使能 UART0\_IRQn 通道。

```
void driver_uart_init(void)  
{  
    RCC_PeriphClockCmd(APBPeriph_UART0, APBPeriph_UART0_CLOCK, ENABLE);  
    UART_InitTypeDef UART_InitStruct;  
    UART_StructInit(&UART_InitStruct);  
    .....  
    UART_InitStruct.UART_Div          = BaudRate_Table[BAUD_RATE_115200].div;  
    UART_InitStruct.UART_Ovsr         = BaudRate_Table[BAUD_RATE_115200].ovsr;  
    UART_InitStruct.UART_OvsrAdj     = BaudRate_Table[BAUD_RATE_115200].ovsr_adj;  
    UART_Init(UART0, &UART_InitStruct);  
  
    UART_INTConfig(UART, UART_INT_RD_AVA, ENABLE);  
    UART_INTConfig(UART, UART_INT_RX_IDLE);  
    .....  
    NVIC_InitTypeDef NVIC_InitStruct;  
    NVIC_InitStruct.NVIC IRQChannel      = UART0_IRQn;  
    NVIC_InitStruct.NVIC IRQChannelCmd   = (FunctionalState)ENABLE;  
    NVIC_InitStruct.NVIC IRQChannelPriority = 3;  
    NVIC_Init(&NVIC_InitStruct);  
}
```

发送字符串："### Uart demo--Auto Hardware Flow Contrl ###\r\n"，实现多字节连续发送(详情参考：UART->Polling)。

```
void uart_demo(void)  
{  
    .....  
    char *demoStr = "### Uart demo--Auto Hardware Flow Contrl ###\r\n";  
    demoStrLen = strlen(demoStr);
```

```
    memcpy(String_Buf, demoStr, demoStrLen);
    uart_senddata_continuous(UART0, String_Buf, demoStrLen);
    .....
}
```

当 UART 收到数据时触发 UART\_INT\_RD\_AVA 中断，或当 UART 空闲时触发 UART\_INT\_RX\_IDLE 中断，进入中断处理函数 UART0\_Handler：

失能 UART\_INT\_RD\_AVA 中断；

判断 UART\_FLAG\_RX\_IDLE 状态为 SET 时，即 UART 空闲（UART 接收数据结束，进入空闲状态）：

- 失能 UART\_INT\_RX\_IDLE 中断；
- 清除 UART0 接收 FIFO；
- 重新使能 UART\_INT\_RX\_IDLE 中断；
- 接收判断标志 receive\_flag 置位；

判断 Interrupt Identifier：

- 当 id 为 UART\_INT\_ID\_RX\_LEVEL\_REACH 时（数据长度达到 rxTriggerLevel，产生该类型中断），接收 FIFO 数据，保存到 UART\_Recv\_Buf 中；
- 当 id 为 UART\_INT\_ID\_RX\_DATA\_TIMEOUT 时（数据长度未达到 rxTriggerLevel，产生该类型中断），接收 FIFO 数据，保存到 UART\_Recv\_Buf 中；

使能 UART\_INT\_RD\_AVA 中断。

```
void UART0_Handler()
{
    .....
    UART_INTConfig(UART, UART_INT_RD_AVA, DISABLE);
    if (UART_GetFlagState(UART, UART_FLAG_RX_IDLE) == SET)
    {
        UART_INTConfig(UART0, UART_INT_RX_IDLE, DISABLE);
        UART_ClearRxFIFO(UART0);
        UART_INTConfig(UART0, UART_INT_RX_IDLE, ENABLE);
        receive_flag = true;
    }
    switch (int_status & 0x0E)
    {
        case UART_INT_ID_RX_DATA_TIMEOUT:
        {
            length = UART_GetRxFIFODataLen(UART0);
            UART_ReceiveData(UART0, UART_Recv_Buf, length);
        }
    }
}
```

```
for (uint8_t i = 0; i < length; i++)  
{  
    UART_Send_Buf[UART_Recv_Buf_Lenth + i] = UART_Recv_Buf[i];  
}  
UART_Recv_Buf_Lenth += length;  
break;  
}  
  
case UART_INT_ID_RX_LEVEL_REACH:  
{  
    length = UART_GetRxFIFODataLen(UART0);  
    UART_ReceiveData(UART0, UART_Recv_Buf, length);  
    for (uint8_t i = 0; i < length; i++)  
    {  
        UART_Send_Buf[UART_Recv_Buf_Lenth + i] = UART_Recv_Buf[i];  
    }  
    UART_Recv_Buf_Lenth += length;  
    break;  
}  
.....  
}  
UART_INTConfig(UART, UART_INT_RD_AVA, ENABLE);  
}
```

循环检测接收数据标志 receive\_flag，若其值为 true，将接收数据标志复位，发送接收的数据到 PC 端。

```
while (1)  
{  
    if (receive_flag == true)  
    {  
        receive_flag = false;  
        uart_senddata_continuous(UART0, UART_Send_Buf, UART_Recv_Buf_Lenth);  
        .....  
    }  
}
```

使用串口调试助手，发送数据，S-Bee2 收到数据触发中断，在中断处理函数中接收数据；S-Bee2 回相同数据到 PC 端，串口调试助手收到 S-Bee2 回的数据。在 DebugAnalyser 工具上，打印 UART 接收数据以及中断信息。

## 4.3 UART 唤醒深度低功耗状态 (DLPS)

通过 UART 实现 S-Bee2 与 PC 端的数据通信。PC 端发送数据给 S-Bee2 时，S-Bee2 收到数据并通过中断向 app\_task 发送消息事件，app\_task 检测到消息事件，在 app 层解析消息，并回相同数据给 PC 端，PC 端收到 S-Bee2 回的数据；

同时，系统处于 IDLE 状态时，会自动进入 DLPS；当通过串口调试助手发送数据给 S-Bee2 时，UART\_RX\_PIN 出现低电平，唤醒系统。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\UART\DLPS。

### 4.3.1 硬件设计

硬件连接：P3\_0 -> RX， P3\_1 -> TX。

EVB 外接 FT232 模块，连接 P3\_0 和 FT232 的 RX， P3\_1 和 FT232 的 TX。

### 4.3.2 软件流程

开启 DLPS 功能。

```
#define DLPS_EN          1  
#define USE_UART_DLPS     1  
#define USE_USER_DEFINE_DLPS_EXIT_CB    1  
#define USE_USER_DEFINE_DLPS_ENTER_CB    1
```

初始化 UART 全局数据。

```
void global_data_uart_init(void)  
{  
    IO_UART_DLPS_Enter_Allowed = true;  
    UART_RX_Count = 0;  
    memset(UART_RX_Buffer, 0, sizeof(UART_RX_Buffer));  
}
```

配置 PAD、PINMUX：设置 P3\_0、P3\_1 为 UART0 功能。

执行 pwr\_mgr\_init 函数，初始化电源管理：

- 注册用户进入 DLPS 回调函数 app\_enter\_dlps\_config，注册用户退出 DLPS 回调函数 app\_exit\_dlps\_config；
- 注册硬件控制回调函数 DLPS\_IO\_EnterDlpsCb 和 DLPS\_IO\_ExitDlpsCb，进入 DLPS 会保存 CPU、PINMUX、Peripheral 等，退出 DLPS 会恢复 CPU、PINMUX、Peripheral 等；
- 设置 DLPS 模式；
- 设置唤醒 DLPS 方式为 UART\_RX\_PIN 低电平唤醒。

```
void pwr_mgr_init(void)
{
#if DLPS_EN
    if (false == dlps_check_cb_reg(app_dlps_check_cb))
        .....
    DLPS_IORRegUserDlpsEnterCb(app_enter_dlps_config);
    DLPS_IORRegUserDlpsExitCb(app_exit_dlps_config);
    DLPS_IORRegister();
    lps_mode_set(LPM_DLPS_MODE);
    System_WakeUpPinEnable(UART_RX_PIN, PAD_WAKEUP_POL_LOW, 0);
    .....
#endif
}
```

在 app\_enter\_dlps\_config 中，执行 io\_uart\_dlps\_enter 函数，设置 UART\_TX\_PIN 和 UART\_RX\_PIN 为 SW 模式，设置唤醒 DLPS 方式为 UART\_RX\_PIN 低电平唤醒。

```
void io_uart_dlps_enter(void)
{
    Pad_ControlSelectValue(UART_TX_PIN, PAD_SW_MODE);
    Pad_ControlSelectValue(UART_RX_PIN, PAD_SW_MODE);
    System_WakeUpPinEnable(UART_RX_PIN, PAD_WAKEUP_POL_LOW, 0);
}
```

在 app\_exit\_dlps\_config 中，执行 io\_uart\_dlps\_exit 函数，设置 UART\_TX\_PIN 和 UART\_RX\_PIN 为 PINMUX 模式。

```
void io_uart_dlps_exit(void)
{
    Pad_ControlSelectValue(UART_TX_PIN, PAD_PINMUX_MODE);
    Pad_ControlSelectValue(UART_RX_PIN, PAD_PINMUX_MODE);
}
```

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_uart\_init 函数，对 UART 外设进行初始化：

- 默认波特率 115200，无奇偶校验，停止位 1 位，数据长度 8bits，FIFO 阈值为 16 等；
- 使能 UART 接收中断（UART\_INT\_RD\_AVA）和 UART 空闲中断（UART\_INT\_IDLE）。

```
void driver_uart_init(void)
```

```
{  
    .....  
    UART_InitStruct.parity      = UART_PARITY_NO_PARTY;  
    UART_InitStruct.stopBits     = UART_STOP_BITS_1;  
    UART_InitStruct.wordLen      = UART_WROD_LENGTH_8BIT;  
    UART_InitStruct.rxTriggerLevel = 16;  
    UART_InitStruct.idle_time     = UART_RX_IDLE_2BYTE;  
    .....  
    UART_INTConfig(UART, UART_INT_RD_AVA, ENABLE);  
    UART_INTConfig(UART, UART_INT_IDLE, ENABLE);  
}
```

开始任务调度。

```
os_sched_start();
```

当 UART 收到数据时触发 UART\_INT\_RD\_AVA 中断，或当 UART 空闲时触发 UART\_INT\_RX\_IDLE 中断，进入中断处理函数 UART0\_Handler:

失能 UART\_INT\_RD\_AVA 中断；

判断 UART\_FLAG\_RX\_IDLE 状态为 SET 时，即 UART 空闲（UART 接收数据结束，进入空闲状态）：

- 失能 UART\_INT\_RX\_IDLE 中断；
- 定义消息类型 IO\_MSG\_TYPE\_UART，子类型 IO\_MSG\_UART\_RX，保存 UART\_RX\_Buffer 数据，发送 msg 给 task；
- 清除 UART0 的 RxFIFO；重新使能 UART\_INT\_RX\_IDLE 中断；

判断 Interrupt Identifier：

- 当 id 为 UART\_INT\_ID\_RX\_LEVEL\_REACH 时（数据长度达到 rxTriggerLevel，产生该类型中断），接收 FIFO 数据，保存到 UART\_RX\_Buffer 中；
- 当 id 为 UART\_INT\_ID\_RX\_DATA\_TIMEOUT 时（数据长度未达到 rxTriggerLevel，产生该类型中断），接收 FIFO 数据，保存到 UART\_RX\_Buffer 中；

使能 UART\_INT\_RD\_AVA 中断。

```
void UART0_Handler()  
{  
    .....  
    UART_INTConfig(UART0, UART_INT_RD_AVA | UART_INT_RX_LINE_STS, DISABLE);  
    if (UART_GetFlagState(UART, UART_FLAG_RX_IDLE) == SET)  
    {  
        UART_INTConfig(UART0, UART_INT_RX_IDLE, DISABLE);  
        T_IO_MSG int_uart_msg;
```

```
int_uart_msg.type = IO_MSG_TYPE_UART;
int_uart_msg.subtype = IO_MSG_UART_RX;
UART_RX_Buffer[UART_RX_Count] = UART_RX_Count;
int_uart_msg.u.buf = (void *)(&UART_RX_Buffer);
if(false == app_send_msg_to_apptask(&int_uart_msg))
.....
UART_ClearRxFIFO(UART0);
UART_INTConfig(UART0, UART_INT_RX_IDLE, ENABLE);
}

switch (int_status & 0x0E)
{
    case UART_INT_ID_RX_DATA_TIMEOUT:
        rx_len = UART_GetRxFIFOLen(UART0);
        UART_ReceiveData(UART0, &UART_RX_Buffer[UART_RX_Count], rx_len);
        UART_RX_Count += rx_len;
        break;
    case UART_INT_ID_RX_LEVEL_REACH:
        rx_len = UART_GetRxFIFODataLen (UART0);
        UART_ReceiveData(UART0, &UART_RX_Buffer[UART_RX_Count], rx_len);
        UART_RX_Count += rx_len;
        break;
.....
}
UART_INTConfig(UART, UART_INT_RD_AVA, ENABLE);
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_UART，执行 io\_handle\_uart\_msg 函数，执行 io\_uart\_handle\_msg：

判断子类型为 IO\_MSG\_UART\_RX 时：

- 将 UART\_RX\_Buffer 中的数据通过 UART 发送到 PC 端；
- 初始化 UART 全局数据；
- 判断 UART 发送缓冲区为空的状态为 0 时，即发送完成，IO\_UART\_DLPS\_Enter\_Allowed 置位。

```
void io_uart_handle_msg(T_IO_MSG *io_uart_msg)
{
.....
```

```
if (IO_MSG_UART_RX == subtype)
{
    uart_senddata_continuous(UART0, UART_RX_Buffer, UART_RX_Count);
    global_data_uart_init();
    while (UART_GetFlagState(UART0, UART_FLAG_THR_EMPTY) == 0)
    {
        IO_UART_DLPS_Enter_Allowed = true;
    }
}
```

系统处于 IDLE 状态时，会自动进入 DLPS；

通过串口调试助手发送数据给 S-Bee2 时，UART\_RX\_PIN 出现低电平，唤醒 DLPS，执行 System\_Handler：

- 清除 UART\_RX\_PIN 的唤醒中断挂起位；
- 失能 UART\_RX\_PIN 的唤醒功能；
- IO\_UART\_DLPS\_Enter\_Allowed 置 0。

```
void System_Handler(void)
{
    APP_PRINT_INFO0("[main] System_Handler");
    if (System_WakeUpInterruptValue(UART_RX_PIN) == SET)
    {
        Pad_ClearWakeupINTPendingBit(UART_RX_PIN);
        System_WakeUpPinDisable(UART_RX_PIN);
        IO_UART_DLPS_Enter_Allowed = false;
    }
}
```

系统处于 IDLE 状态时，会自动进入 DLPS。

在 DLPS 状态下，使用串口调试助手发送的第一笔数据唤醒 S-Bee2，再发送数据，S-Bee2 收到数据并回相同数据，串口调试助手收到 S-Bee2 回的数据。

## 4.4 UART 通过 GDMA 接收和发送定长数据

通过 UART 实现 S-Bee2 与 PC 端的数据通信。S-Bee2 发送数据到 PC 端时：S-Bee2 发送数据并通过 GDMA 外设搬运数据到 PC 端；PC 端发送数据给 S-Bee2 时：S-Bee2 收到数据并通过 GDMA 搬运数据，并回相同数据给 PC 端，PC 端收到 S-Bee2 回的数据。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ UART\ GDMA。

## 4.4.1 硬件设计

硬件连接: P3\_0 -> RX, P3\_1 -> TX。

EVB 外接 FT232 模块, 连接 P3\_0 和 FT232 的 RX, P3\_1 和 FT232 的 TX。

## 4.4.2 软件流程

引脚定义:

```
#define UART_TX_PIN          P3_0  
#define UART_RX_PIN          P3_1
```

开启 UART\_GDMA 功能:

```
#define UART_GDMA_TX_ENABLE    EVB_ENABLE  
#define UART_GDMA_RX_ENABLE    EVB_ENABLE
```

配置 PAD: 设置引脚、PINMUX 模式、PowerOn、内部上拉、输出失能、输出高;

配置 PINMUX: 分配引脚 P3\_0、P3\_1 分别为 UART0\_TX、UART0\_RX 功能。

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,  
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数, 执行 driver\_uart\_init 函数, 对 UART 外设进行初始化:

- 设置 div、ovsr、ovsr\_adj, 波特率 115200;
- rxThdLevel 设置为 16, FIFO 阈值为 16;
- idle\_time 设置为 UART\_RX\_IDLE\_2BYTE, 即当前波特率下接收 2 个字节的时间;
- dmaEn 设置为 UART\_DMA\_ENABLE, 即使能 UART\_DMA 传输;
- txWaterLevel 设置为 15, 其值一般设置为 (TX\_FIFO\_SIZE- GDMA\_MSize);
- rxWaterLevel 设置为 1, 其值一般设置为 GDMA\_MSize;
- 使能 UART\_TxDmaEn 和 UART\_RxDmaEn;

```
void driver_uart_init(void)  
{  
    .....  
    UART_InitStruct.UART_Div      = CBaudrateTable[UART_BAUDRATE].div;  
    UART_InitStruct.UART_Ovsr     = CBaudrateTable[UART_BAUDRATE].ovsr;  
    UART_InitStruct.UART_OvsrAdj   = CBaudrateTable[UART_BAUDRATE].ovsr_adj;  
    UART_InitStruct.rxTriggerLevel = 16;           //1~29  
    UART_InitStruct.idle_time     = UART_RX_IDLE_2BYTE; //idle interrupt wait time  
    UART_InitStruct.UART_DmaEn    = UART_DMA_ENABLE;
```

```
UART_InitStruct.UART_TxDmaEn      = ENABLE;
UART_InitStruct.UART_RxDmaEn      = ENABLE;
UART_InitStruct.UART_TxWaterLevel = 15; //Better to equal TX_FIFO_SIZE(16)- GDMA_MSize
UART_InitStruct.UART_RxWaterLevel = 1;   //Better to equal GDMA_MSize
UART_InitStruct.UART_TxDmaEn      = ENABLE;
UART_InitStruct.UART_RxDmaEn      = ENABLE;
UART_Init(UART, &UART_InitStruct);
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时, 执行 app\_handle\_dev\_state\_evt 函数, 执行 io\_uart\_demo 函数, 发送数组 send\_data, 实现数据连续发送, 并初始化 GDMA 外设:

- 执行函数 UART\_SendData\_Continuous, 发送数组 send\_data 数据到 PC 端;
- 执行 UART\_SendDataByDMA, 初始化 DMA 发送数据外设;
- 执行 UART\_ReceiveDataByDMA, 初始化 DMA 接收数据外设

```
void io_uart_demo(void)
{
    .....
    UART_SendData_Continuous(UART0, send_data, sizeof(send_data));
    UART_SendDataByDMA();
    UART_ReceiveDataByDMA();
}
```

在 UART\_SendDataByDMA 中, 对 DMA 通道 0 外设进行初始化:

- 使用 DMA 通道 0;
  - DMA 的传输方向为内存到外设传输; DMA\_SourceAddr 设置为 DMA\_SendData\_Buffer; DMA\_DestinationAddr 设置为(&(UART0->RB\_THR));
- 使能 DMA 通道 0 总传输完成中断 (DMA\_INT\_Transfer);  
使能 DMA 通道 0 传输。

```
void UART_SendDataByDMA(void)
{
    .....
    DMA_InitStruct.GDMA_ChannelNum     = UART_TX_GDMA_CHANNEL_NUM;
    DMA_InitStruct.GDMA_DIR           = DMA_DIR_MemoryToPeripheral;
    DMA_InitStruct.GDMA_BufferSize    = UART_TX_GDMA_BUFFER_SIZE;//determine total
transfer size
```

```

.....
GDMA_InitStruct.GDMA_SourceAddr      = (uint32_t)GDMA_SendData_Buffer;
GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t)(&(UART0->RB_THR));
.....
GDMA_INTConfig(UART_TX_GDMA_CHANNEL_NUM, GDMA_INT_Transfer, ENABLE);
GDMA_Cmd(UART_TX_GDMA_CHANNEL_NUM, ENABLE);
}

```

在 UART\_ReceiveDataByGDMA 中，对 GDMA 通道 1 外设进行初始化：

- 使用 GDMA 通道 1；
- GDMA 的传输方向为外设到内存传输； GDMA\_SourceAddr 设置为(&(UART0->RB\_THR));
- GDMA\_DestinationAddr 设置为 GDMA\_ReceiveData\_Buffer；
- 使能 GDMA 通道 1 总传输完成中断 (GDMA\_INT\_Transfer)；
- 使能 GDMA 通道 1 传输。

```

void UART_ReceiveDataByGDMA (void)
{
    .....
    GDMA_InitStruct.GDMA_ChannelNum      = UART_RX_GDMA_CHANNEL_NUM;
    GDMA_InitStruct.GDMA_DIR            = GDMA_DIR_PeripheralToMemory;
    .....
    GDMA_InitStruct.GDMA_SourceAddr      = (uint32_t)(&(UART0->RB_THR));
    GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t)GDMA_ReceiveData_Buffer;
    .....
    GDMA_INTConfig(UART_RX_GDMA_CHANNEL_NUM, GDMA_INT_Transfer, ENABLE);
    GDMA_Cmd(UART_RX_GDMA_CHANNEL_NUM, ENABLE);
}

```

当 UART 通过 GDMA 传输发送数据时， GDMA 搬运数据到目的端地址 (&(UART0->RB\_THR))，当 GDMA 发送数据总传输完成时，触发 GDMA\_INT\_Transfer 中断，进入中断处理函数 UART\_TX\_GDMA\_Handler：

- 清除 GDMA 通道 0 GDMA\_INT\_Transfer 中断挂起位；
- 定义消息类型 IO\_MSG\_TYPE\_GDMA，子类型为 0，保存发送数据，发送 msg 给 task；

```

void UART_TX_GDMA_Handler (void)
{
    GDMA_ClearINTPendingBit(UART_TX_GDMA_CHANNEL_NUM, GDMA_INT_Transfer);

    T_IO_MSG int_uart_msg;
}

```

```
int_uart_msg.type = IO_MSG_TYPE_GDMA;
int_uart_msg.subtype = 0;
int_uart_msg.u.buf = (void*)(GDMA_SendData_Buffer);
.....
}
```

当 UART 通过 GDMA 搬运接收数据时， GDMA 搬运数据到内存，当 GDMA 接收数据总传输完成时，触发 GDMA\_INT\_Transfer 中断，进入中断处理函数 UART\_RX\_GDMA\_Handler：

- 清除 GDMA 通道 1GDMA\_INT\_Transfer 中断挂起位；
- 定义消息类型 IO\_MSG\_TYPE\_GDMA，子类型为 1，发送 msg 给 task；

```
void UART_RX_GDMA_Handler (void)
{
    GDMA_ClearINTPendingBit(UART_RX_GDMA_CHANNEL_NUM, GDMA_INT_Transfer);

    T_IO_MSG int_uart_msg;
    int_uart_msg.type = IO_MSG_TYPE_GDMA;
    int_uart_msg.subtype = 1;
    int_uart_msg.u.buf = (void*)(GDMA_SendData_Buffer);
    .....
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg) 函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_GDMA，执行 io\_handle\_gdma\_msg(&io\_msg) 函数，执行 io\_handle\_gdma\_msg：

判断 subtype 类型：

- 如果 subtype == 0，即 TX 传输，打印 GDMA 搬运发送数据完成信息；
- 如果 subtype == 1，即 RX 传输，打印 GDMA 搬运接收数据完成信息，S-Bee2 收到数据并回相同数据给 PC 端。

```
void io_handle_gdma_msg(T_IO_MSG *io_gdma_msg)
{
    uint16_t subtype = io_gdma_msg->subtype;
    if (subtype == 0)
    {
        APP_PRINT_INFO0("io_handle_gdma_msg: uart tx done ");
    }
    else if (subtype == 1)
    {
        APP_PRINT_INFO0("io_handle_gdma_msg: uart rx done ");
    }
}
```

```
APP_PRINT_INFO0("io_handle_gdma_msg: uart rx done ");
UART_SendData_Continuous(UART0,
                           GDMA_ReceiveData_Buffer,
                           UART_RX_GDMA_BUFFER_SIZE);
}
}
```

S-Bee2 发送数据，GDMA 搬运发送数据完成后产生中断，在 DebugAnalyser 工具上，打印 GDMA 搬运发送数据完成信息，串口调试助手收到 S-Bee2 发送的数据；

使用串口调试助手发送数据到 S-Bee2，通过 GDMA 外设搬运接收数据完成后产生中断，在 DebugAnalyser 工具上，打印接收数据完成信息，S-Bee2 收到数据并回相同数据，串口调试助手收到 S-Bee2 回的数据。

## 4.5 UART 通过 GDMA 接收不定长数据

通过 UART 实现 S-Bee2 与 PC 端的数据通信。PC 端发送数据给 S-Bee2 时，S-Bee2 通过 UART 中断和 GDMA 中断搬运接收数据，并回相同数据给 PC 端，PC 端收到 S-Bee2 回的数据。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ UART\ DMA\_unfixedlen。

### 4.5.1 硬件设计

硬件连接：P3\_0 -> RX， P3\_1 -> TX。

EVB 外接 FT232 模块，连接 P3\_0 和 FT232 的 RX， P3\_1 和 FT232 的 TX。

### 4.5.2 软件流程

引脚定义：

```
#define UART_TX_PIN          P3_0
#define UART_RX_PIN          P3_1
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出失能、输出高；

配置 PINMUX：分配引脚 P3\_0、P3\_1 分别为 UART0\_TX、UART0\_RX 功能。

使能 UART0 时钟；

初始化 UART0 外设：

- 设置波特率 115200；FIFO 阈值为 8；idle\_time 设置为 UART\_RX\_IDLE\_2BYTE；
- 使能 UART\_TxDmaEn 和 UART\_RxDmaEn；
- UART\_DmaEn 设置为 UART\_DMA\_ENABLE，即使能 UART\_DMA 传输；

使能 UART 接收空闲中断(UART\_INT\_RX\_IDLE)以及线接收状态中断(UART\_INT\_RX\_LINE\_STS)。

```
void driver_uart_init(void)
{
```

```
.....
RCC_PeriphClockCmd(APBPeriph_UART0, APBPeriph_UART0_CLOCK, ENABLE);
.....
UART_InitStruct.UART_RxDmaEn      = ENABLE;
UART_InitStruct.UART_TxDmaEn      = ENABLE;
UART_InitStruct.UART_DmaEn        = UART_DMA_ENABLE;
UART_Init(UART0, &UART_InitStruct);
.....
UART_INTConfig(UART0, UART_INT_RX_IDLE | UART_INT_RX_LINE_STS, ENABLE);
.....
}
```

执行 driver\_gdma4\_init 函数，对 GDMA 接收数据外设进行初始化：

- 使用 GDMA 通道 4；
  - GDMA 的传输方向为外设到内存传输； GDMA\_SourceAddr 设置为(&(UART0->RB\_THR));  
GDMA\_DestinationAddr 设置为 GDMA\_Rx\_Buf;
- 使能 GDMA 通道 4 总传输完成中断 (GDMA\_INT\_Transfer)；使能 GDMA 通道 4 传输。

```
void driver_gdma4_init (void)
{
    .....
    GDMA_InitStruct.GDMA_ChannelNum      = DMA_RX_CHANNEL_NUM;
    GDMA_InitStruct.GDMA_DIR             = GDMA_DIR_PeripheralToMemory;
    .....
    GDMA_InitStruct.GDMA_SourceAddr      = (uint32_t)(&(UART0->RB_THR));
    GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t)GDMA_Rx_Buf;
    .....
    GDMA_INTConfig(UART_RX_GDMA_CHANNEL_NUM, GDMA_INT_Transfer, ENABLE);
    GDMA_Cmd(UART_RX_GDMA_CHANNEL_NUM, ENABLE);
}
```

执行 driver\_gdma3\_init 函数，对 GDMA 发送数据外设进行初始化：

- 使用 GDMA 通道 3；
  - GDMA 的传输方向为内存到外设传输； GDMA\_SourceAddr 设置为 GDMA\_Tx\_Buf；  
GDMA\_DestinationAddr 设置为(&(UART0->RB\_THR));
- 使能 GDMA 通道 3 总传输完成中断 (GDMA\_INT\_Transfer)；

```
void driver_gdma3_init (void)
{
```

```

.....
GDMA_InitStruct.GDMA_ChannelNum          = DMA_TX_CHANNEL_NUM;
GDMA_InitStruct.GDMA_DIR                 = GDMA_DIR_MemoryToPeripheral;
.....
GDMA_InitStruct.GDMA_SourceAddr          = (uint32_t)GDMA_Tx_Buf;
GDMA_InitStruct.GDMA_DestinationAddr     = (uint32_t)(&(UART0->RB_THR));
.....
GDMA_INTConfig(UART_RX_GDMA_CHANNEL_NUM, GDMA_INT_Transfer, ENABLE);
}

```

当 GDMA 接收数据总传输完成时，触发 GDMA\_INT\_Transfer 中断，进入中断处理函数 GDMA0\_Channel4\_Handler:

- 失能 GDMA 通道 4 传输;
- 清除 GDMA 通道 4 全部中断挂起;
- 保存 GDMA\_Rx\_Buf 数据;
- 清除 GDMA 通道 4 GDMA\_INT\_Transfer 中断挂起位;
- 重新设置目的端地址;
- 使能 GDMA 通道 4 传输。

```

void GDMA0_Channel4_Handler (void)
{
    GDMA_Cmd(DMA_RX_CHANNEL_NUM, DISABLE);
    GDMA_ClearAllTypeINT(DMA_RX_CHANNEL_NUM);
    .....
    memcpy(GDMA_Tx_Buf + GDMA_BLOCK_SIZE * (count - 1), GDMA_Rx_Buf,
GDMA_BLOCK_SIZE);

    GDMA_ClearINTPendingBit(DMA_RX_CHANNEL_NUM, GDMA_INT_Transfer);
    GDMA_SetDestinationAddress(DMA_RX_CHANNEL, (uint32_t)GDMA_Rx_Buf);
    GDMA_Cmd(DMA_RX_CHANNEL_NUM, ENABLE);
}

```

当 UART 接收空闲时触发 UART\_INT\_RX\_IDLE 中断，当 UART 线状态出错误时触发 UART\_INT\_RX\_LINE\_STS 中断，进入中断处理函数 UART0\_Handler:

获取中断标志；

判断 UART\_FLAG\_RX\_IDLE 状态为 SET 时，即 UART 空闲（UART 接收数据结束，进入空闲状态）：

- 悬挂 GDMA 通道 4 传输;
- 失能 UART\_FLAG\_RX\_IDLE 中断;

- 保存 GDMA 通道 4 中接收数据长度;
  - 打印接收数据信息;
  - 判断 GDMA 通道 4 中是否有数据: 若有数据, 则将数据保存到 `GDMA_Tx_Buf` 中, 失能 GDMA 通道 4 传输, 清除 GDMA 通道 4 传输悬挂; 执行 `driver_gdma4_init` 函数, 将接收数据标志 `receiveflg` 置位; 若无数据, 则清除 GDMA 通道 4 传输悬挂, 将接收数据标志 `receiveflg` 置位;
  - 清除 UART 的 RXFIFO 中的数据;
  - 重新使能 `UART_INT_IDLE` 中断;
- 判断 Interrupt Identifier:
- 当 id 为 `UART_INT_ID_RX_TMEOUT` 时 (数据长度未达到 `rxTriggerLevel`, 产生该类型中断), 检测接收到数据状态, 状态为 SET 时, 接收并保存数据;
  - 当 id 为 `UART_INT_ID_LINE_STATUS` 时 (线状态错误, 产生该类型中断), 打印线状态错误信息;

```
void UART0_Handler()
{
    /* read interrupt id */
    uint32_t int_status = UART_GetIID(UART0);

    if (UART_GetFlagStatus(UART0, UART_FLAG_RX_IDLE) == SET)
    {
        GDMA_SuspendCmd(DMA_RX_CHANNEL, ENABLE);
        UART_INTConfig(UART0, UART_INT_RX_IDLE, DISABLE);

        data_len = GDMA_GetTransferLen(DMA_RX_CHANNEL);
        for (uint32_t i = 0; i < data_len; i++)
        {
            DBG_DIRECT("value is 0x%x", GDMA_Rx_Buf[i]);
        }

        if (data_len)
        {
            receive_offset += data_len;
            memcpy(GDMA_Tx_Buf + GDMA_BLOCK_SIZE * count, GDMA_Rx_Buf, receive_offset);
            .....
            GDMA_Cmd(DMA_RX_CHANNEL, DISABLE);
            GDMA_SuspendCmd(DMA_RX_CHANNEL, DISABLE);
        }
    }
}
```

```
    driver_gdma4_init ();
    receiveflg = true;
}
else
{
    GDMA_SuspendCmd(DMA_RX_CHANNEL, DISABLE);
    receiveflg = true;
}
UART_ClearRx FIFO(UART0);
UART_INTConfig(UART0, UART_INT_RX_IDLE, ENABLE);
}
UART_INTConfig(UART0, UART_INT_RD_AVA | UART_INT_RX_LINE_STS, DISABLE);
switch (int_status & 0x0E)
{
case UART_INT_ID_RX_DATA_TIMEOUT:
{
    while (UART_GetFlagStatus(UART0, UART_FLAG_RX_DATA_AVA) == SET)
    {
        UART_ReceiveData(UART0, &tmp, 1);
    }
    break;
}
case UART_INT_ID_LINE_STATUS:
{
    .....
}
}
```

检测接收数据标志 receiveflg，状态为 TRUE 时，设置 GDMA 通道 3BufferSize 为接收数据总量，使能 GDMA 通道 3 传输，将 receiveflg 状态设置为 false：

```
while (1)
{
    .....
    if (receiveflg)
    {
```

```
GDMA_SetBufferSize(DMA_TX_CHANNEL, receive_offset);
GDMA_Cmd(DMA_TX_CHANNEL_NUM, ENABLE);
.....
receiveflg = false;
}
}
```

当 UART 通过 GDMA 传输发送数据时， GDMA 搬运数据到(&(UART0->RB\_THR)), 当 GDMA 发送数据总传输完成时，触发 GDMA\_INT\_Transfer 中断，进入中断处理函数 GDMA0\_Channel3\_Handler:

- 打印 GDMA 搬运 UART 接收数据信息；
- 失能 GDMA 通道 3 传输；
- 清除 GDMA 通道 3 全部中断挂起。

```
void GDMA0_Channel3_Handler (void)
{
    DBG_DIRECT("UART_TX_GDMA_Handler");
    GDMA_Cmd(DMA_TX_CHANNEL_NUM, DISABLE);
    GDMA_ClearAllTypeINT(DMA_TX_CHANNEL_NUM);
}
```

使用串口调试助手发送不定长数据到 S-Bee2， S-Bee2 通过 GDMA 中断和 UART 中断收到数据，再通过 GDMA 搬运接收数据到 PC 端，串口调试助手收到 S-Bee2 回的数据，在 DebugAnalyser 工具上，打印接收数据以及 GDMA 搬运数据完成信息。

## 5 定时器和脉冲宽度调制(TIM&PWM)

专用硬件设备：逻辑分析仪

专用测试软件：DSview V1.1.2

### 5.1 脉冲宽度调制输出 (PWM)

使用 TIM2，实现输出周期为 2s、占空比为 50% 的 PWM 功能，为了便于观察，使用 P0\_1 (LED0) 作为 PWM 输出，实现 LED0 的闪烁。

同时实现 PWM 互补输出功能，使用 P0\_2 作为 PWM\_P 输出，使用 P2\_2 作为 PWM\_N 输出。互补输出功能打开后，当不设置死区时间时，PWM\_P 与 PWM 同相，PWM\_N 与 PWM 反相；当设置死区后，PWM\_P 和 PWM\_N 在输出高电平时会比 PWM 延迟一个死区时长。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\TIM\PWM。

#### 5.1.1 硬件设计

硬件连接：P0\_1 -> LED0, P0\_2 -> LED1 或 P2\_2 -> LED1。

在 EVB 上，连接 P0\_1 和 LED0，观察 LED0 的输出；如需观察 P0\_2 或 P2\_2 的输出，连接 P0\_2 和 LED1 或 P2\_2 和 LED1，观察 LED1 的输出。

#### 5.1.2 软件流程

引脚定义：

```
#define PWM_OUT_PIN          P0_1  
#define PWM_OUT_P_PIN        P0_2  
#define PWM_OUT_N_PIN        P2_2
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、无内部上拉、输出使能、输出高；

配置 PINMUX：分配引脚分别为 timer\_pwm2、PWM2\_P、PWN2\_N 功能。

```
void board_pwm_init(void)  
{  
    Pad_Config(PWM_OUT_PIN,  PAD_PINMUX_MODE,  PAD_IS_PWRON,  PAD_PULL_NONE,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);  
  
    Pad_Config(PWM_OUT_P_PIN,  PAD_PINMUX_MODE,  PAD_IS_PWRON,  PAD_PULL_NONE,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);  
  
    Pad_Config(PWM_OUT_N_PIN,  PAD_PINMUX_MODE,  PAD_IS_PWRON,  PAD_PULL_NONE,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);
```

```
Pinmux_Config(PWM_OUT_PIN, PWM_OUT_PIN_PINMUX);
Pinmux_Config(PWM_OUT_P_PIN, PWM_OUT_P_PIN_PINMUX);
Pinmux_Config(PWM_OUT_N_PIN, PWM_OUT_N_PIN_PINMUX);
}
```

使能 TIMER 时钟；

初始化 TIMER 外设：

- TIM\_PWM\_En 设置为 PWM\_ENABLE，即 PWM 模式；
  - TIM\_Mode 设置为 TIM\_Mode\_UserDefine，即用户定义模式；
  - 设置 TIM\_PWM\_High\_Count 和 TIM\_PWM\_Low\_Count，用户可以根据需求分配高电平周期值和低电平周期值；
  - PWM\_Deazone\_Size 设置为 PWM\_DEAD\_ZONE\_SIZE，即配置双路互补输出死区的大小；
  - PWMDeadZone\_En 设置为 DEADZONE\_ENABLE，使能 PWM 互补输出(带死区功能)功能；
- 使能 TIM2。

```
void driver_pwm_init(void)
{
    RCC_PeriphClockCmd(APBPeriph_TIMER, APBPeriph_TIMER_CLOCK, ENABLE);

    TIM_TimeBaseInitTypeDef TIM_InitStruct;
    TIM_StructInit(&TIM_InitStruct);

    TIM_InitStruct.TIM_PWM_En      = PWM_ENABLE;
    TIM_InitStruct.TIM_Mode        = TIM_Mode_UserDefine;
    TIM_InitStruct.TIM_PWM_High_Count = PWM_HIGH_COUNT;
    TIM_InitStruct.TIM_PWM_Low_Count = PWM_LOW_COUNT;
    TIM_InitStruct.PWM_Deazone_Size = PWM_DEAD_ZONE_SIZE;
    TIM_InitStruct.PWMDeadZone_En   = DEADZONE_ENABLE;

    TIM_TimeBaseInit(TIM2, &TIM_InitStruct);

    TIM_Cmd(TIM2, ENABLE);
}
```

LED0 每秒闪烁；当连接 P0\_2 和 LED1 时，LED1 与 LED0 同相闪烁，当连接 P2\_2 和 LED1，LED1 与 LED0 反相闪烁。

## 5.2 定时器中断

使用 TIM6，实现定时 1s 的定时功能。定时时间到，触发中断，在中断处理函数中翻转 P0\_1 (LED0)，实现 LED0 每秒闪烁。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ TIM\ Timer\_interrupt。

## 5.2.1 硬件设计

硬件连接: P0\_1 -> LED0。

在 EVB 上, 连接 P0\_1 和 LED0, 观察 LED0 的输出。

## 5.2.2 软件流程

配置 PAD、PINMUX, 设置 P0\_1 为 GPIO 输出功能; 使能 GPIO 时钟; 初始化 GPIO 外设: 不配置 GPIO 中断模式, 初始状态输出低电平等 (详情参考: GPIO -> Output\_led)。

初始化 TIMER 外设:

- TIM\_PWM\_En 设置为 PWM\_DISABLE, 即 TIMER 模式;
- TIM\_Period 设置为 TIMER\_PERIOD (40000000-1), 即定时 1s;
- TIM\_Mode 设置为 TIM\_Mode\_UserDefine, 即用户定义模式;

清除 TIM6 中断; 使能 TIM6 中断; 使能 TIM6。

```
void driver_timer_init(void)
{
    .....
    TIM_InitStruct.TIM_PWM_En = PWM_DISABLE;
    TIM_InitStruct.TIM_Period = TIMER_PERIOD ;
    TIM_InitStruct.TIM_Mode = TIM_Mode_UserDefine;
    TIM_TimeBaseInit(TIMER_NUM, &TIM_InitStruct);
    .....
    TIM_ClearINT(TIMER_NUM);
    TIM_INTConfig(TIMER_NUM, ENABLE);
    TIM_Cmd(TIMER_NUM, ENABLE);
}
```

TIM6 定时时间到, 触发中断, 进入中断处理函数 Timer6\_Handler:

- 清除 TIM6 中断, 失能 TIM6;
- 判断当前 LED 状态, 翻转 LED0 电平;
- 重新使能 TIM6。

```
void Timer6_Handler(void)
{
    TIM_ClearINT(TIM6);
    TIM_Cmd(TIM6, DISABLE);
    if (!LED_Status)
```

```
{  
    GPIO_SetBits(GPIO_PIN_OUTPUT);  
    LED_Status = 1;  
}  
else  
{  
    GPIO_ResetBits(GPIO_PIN_OUTPUT);  
    LED_Status = 0;  
}  
TIM_Cmd(TIM6, ENABLE);  
}
```

LED0 每秒闪烁。

## 5.3 数字信号任意波形输出

使用 TIM2，定时周期 10us，定时产生 GDMA 请求，实现 GDMA 搬运数据到 GPIO 寄存器。通过改变 GPIO 输出高低电平时间（GDMA 源数据搬运次数\*TIM 周期），模拟动态改变 PWM 波形占空比的功能。

例如：如果定义数据：{0xFFFFFFFF, 0xFFFFFFFF, 0x0}，则 GPIO 输出高电平时间为  $2 * 10\text{us} = 20\text{us}$ ，GPIO 输出低电平时间为  $1 * 10\text{us} = 10\text{us}$ ，PWM 周期为 30us，占空比 2/3。0xFFFFFFFF 表示高电平，0x0 表示低电平。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ TIM\ GDMA+PWM+GPIO。

### 5.3.1 硬件设计

硬件连接：P2\_2 -> 逻辑分析仪 Pin；

EVB 外接逻辑分析仪，连接 P2\_2 和 Pin，观察 GPIO 输出的波形图。

### 5.3.2 软件流程

配置 PAD、PINMUX，设置 P2\_2 为 GPIO 输出；初始化 GPIO 外设：GPIO\_ControlMode 设置为 GPIO\_HARDWARE\_MODE，即硬件模式，用于 GDMA 搬数据到 GPIO。

```
void driver_gpio_init(void)  
{  
    .....  
    GPIO_InitStruct.GPIO_ControlMode = GPIO_HARDWARE_MODE;  
    .....  
}
```

```
}
```

配置 PAD、PINMUX，设置 P2\_3 为 timer\_pwm5；初始化 TIMER 外设：

- 设置 PWM 模式；
- 配置高、低电平周期为 5us，即输出周期为 10us、占空比为 50% 的 PWM；
- 用户定义模式。

```
void driver_pwm_init(void)
{
    .....
    TIM_InitStruct.TIM_PWM_En = PWM_ENABLE;
    TIM_InitStruct.TIM_PWM_High_Count = PWM_HIGH_COUNT;
    TIM_InitStruct.TIM_PWM_Low_Count = PWM_LOW_COUNT;
    TIM_InitStruct.TIM_Mode = TIM_Mode_UserDefine;
    .....
}
```

初始化 GDMA 外设：

- 使用 GDMA 通道 2；
- GDMA 的传输方向为内存到外设传输；源端地址为 GPIO\_Ctl\_AutoReload，数组中定义 32 位数据，对应 GPIO0~GPIO31；目的端地址为 0x40011200（固定使用）；
- 设置目的端 GDMA\_Handshake\_TIM5；
- 使能 Multi-block 传输，根据需求设置 GDMA\_Multi\_Block\_Mode，如果需要 GPIO 反复输出，则自动加载源端地址(GPIO\_Ctl\_AutoReload)；否则自动加载 LLI 结构体中源端地址(&(GPIO\_Ctl\_LLI[i]))；使能 GDMA 通道 2 GDMA 总传输完成中断 (GDMA\_INT\_Transfer)；使能 GDMA 传输；使能 TIM5。

```
void driver_gdma_multiblock_init(void)
{
    .....
    GDMA_InitStruct.GDMA_ChannelNum = GDMA_CHANNEL_NUM;
    GDMA_InitStruct.GDMA_DIR = GDMA_DIR_MemoryToPeripheral;
    .....
    GDMA_InitStruct.GDMA_SourceAddr = (uint32_t)GPIO_Ctl_AutoReload;
    GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t)(0x40011200); //vendor gpio reg address
    GDMA_InitStruct.GDMA_DestHandshake = GDMA_Handshake_TIM5;
    GDMA_InitStruct.GDMA_Multi_Block_En = 1;
    GDMA_InitStruct.GDMA_Multi_Block_Struct = (uint32_t)GDMA_LLIStruct;
#if(GPIO_OUT_REPEAT == 1)
    GDMA_InitStruct.GDMA_Multi_Block_Mode = LLI_WITH_AUTO_RELOAD_SAR;

```

```
#else
    GDMA_InitStruct.GDMA_Multi_Block_Mode = LLI_TRANSFER;
#endif
for (int i = 0; i < GDMA_LLI_SIZE; i++)
{
    GDMA_LLIStruct[i].SAR = (uint32_t)(&(GPIO_Ctl_LLI[i]));
    GDMA_LLIStruct[i].DAR = (uint32_t)(0x40011200);
    .....
}
GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Transfer, ENABLE);
GDMA_Cmd(GDMA_CHANNEL_NUM, ENABLE);
TIM_Cmd(PWM_TIMER_NUM, ENABLE);
}
```

TIM2 输出 PWM，每个周期 GDMA 搬运数据到 0x40011200，当 GDMA 总传输完成时，触发 GDMA\_INT\_Transfer 中断，进入中断处理函数 GDMA\_Channel\_Handler：

- 清除 GDMA 通道 2 GDMA\_INT\_Transfer 中断挂起位；

```
void GDMA_Channel_Handler(void)
{
    GDMA_ClearINTPendingBit(GDMA_CHANNEL_NUM, GDMA_INT_Transfer);
}
```

搬运 GPIO\_Ctl\_LLI[24]数据时，使用逻辑分析仪检测 GPIO 输出，得到预期占空比 PWM 波形。



图 5-1 GPIO 输出 PWM 波形

搬运 GPIO\_Ctl\_AutoReload [24]数据时，使用逻辑分析仪检测 GPIO 输出，得到预期占空比 PWM 波形。



图 5-2 GPIO 输出 PWM 波形

## 6 直接内存存取控制器(GDMA)

### 6.1 单 Block 传输 (Single Block)

使用 GDMA Single Block 功能 (Block 最大传输数据为 4095)，实现 memory 到 memory 搬运数据。在 DebugAnalyser 工具上，打印 GDMA 打印传输数据完成信息。

工程目录: \HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ GDMA\ Mem2Mem\_single\_block。

#### 6.1.1 软件流程

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_gdma\_init 函数，对 GDMA 外设进行初始化：

- GDMA\_ChannelNum 设置为 GDMA\_CHANNEL\_NUM，使用 GDMA 通道 4；
- GDMA\_DIR 设置为 GDMA\_DIR\_MemoryToMemory，即 GDMA 的传输方向为内存到内存传输；
- GDMA\_BufferSize 设置为 GDMA\_TRANSFER\_SIZE，即完成一次 GDMA 传输的数据个数为 200 (数据大小为: GDMA\_TRANSFER\_SIZE\*GDMA\_DataSize\_Word/4 Bytes)；
- 设置 GDMA\_SourceInc，源端地址递增，设置 GDMA\_DestinationInc，目的端地址递增；
- 设置 GDMA\_SourceDataSize，源端的数据宽度为 32 位，设置 GDMA\_DestinationDataSize，目的端的数据宽度为 32 位；
- 设置 GDMA\_SourceMsize，源端单次 burst 传输的数据个数为 8，设置 GDMA\_DestinationMsize，目的端单次 burst 传输的数据个数为 8；
- GDMA\_SourceAddr 设置为 GDMA\_Send\_Buf; GDMA\_DestinationAddr 设置为 GDMA\_Recv\_Buf; 使能 GDMA 通道 4 总传输完成中断 (GDMA\_INT\_Transfer)。

```
void driver_gdma_init(void)
{
    .....
    GDMA_InitTypeDef GDMA_InitStruct;
    GDMA_StructInit(&GDMA_InitStruct);
    GDMA_InitStruct.GDMA_ChannelNum      = GDMA_CHANNEL_NUM;
    GDMA_InitStruct.GDMA_DIR             = GDMA_DIR_MemoryToMemory;
    GDMA_InitStruct.GDMA_BufferSize     = GDMA_TRANSFER_SIZE;
    GDMA_InitStruct.GDMA_SourceInc      = DMA_SourceInc_Inc;
    GDMA_InitStruct.GDMA_DestinationInc = DMA_DestinationInc_Inc;
```

```
GDMA_InitStruct.GDMA_SourceDataSize      = GDMA_DataSize_Word;
GDMA_InitStruct.GDMA_DestinationDataSize = GDMA_DataSize_Word;
GDMA_InitStruct.GDMA_SourceMsize         = GDMA_Msize_8;
GDMA_InitStruct.GDMA_DestinationMsize   = GDMA_Msize_8;
GDMA_InitStruct.GDMA_SourceAddr          = (uint32_t)GDMA_Send_Buf;
GDMA_InitStruct.GDMA_DestinationAddr    = (uint32_t)GDMA_Recv_Buf;
GDMA_Init(GDMA_Channel, &GDMA_InitStruct);
.....
GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Transfer, ENABLE);
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 GDMA\_Cmd 函数，使能 GDMA 传输。

```
GDMA_Cmd(GDMA_CHANNEL_NUM, ENABLE);
```

数据从 GDMA\_Send\_Buf 搬到 GDMA\_Recv\_Buf 完成时，触发 GDMA\_INT\_Transfer 中断，进入中断处理函数 GDMA\_Channel\_Handler：

- 失能 GDMA 通道 4 GDMA\_INT\_Transfer 中断；
- 定义消息类型 IO\_MSG\_TYPE\_GDMA，发送 msg 给 task；
- 清除 GDMA 通道 4 GDMA\_INT\_Transfer 中断挂起位。

```
void GDMA_Channel_Handler(void)
{
    GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Transfer, DISABLE);

    T_IO_MSG int_gdma_msg;
    int_gdma_msg.type = IO_MSG_TYPE_GDMA;
    int_gdma_msg.subtype = 0;
    if (false == app_send_msg_to_apptask(&int_gdma_msg))
    .....
    GDMA_ClearINTPendingBit(GDMA_CHANNEL_NUM, GDMA_INT_Transfer);
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg) 函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_GDMA，执行 io\_handle\_gdma\_msg：

- 打印传输数据完成信息；
- 判断 GDMA\_Recv\_Buf 与 GDMA\_Send\_Buf 是否相等，不等则打印错误数据（搬运出错）。

```
void io_handle_gdma_msg(T_IO_MSG *io_gdma_msg)
```

```

{
    APP_PRINT_INFO0("[io_gdma] io_handle_gdma_msg: GDMA transfer data completion!");
    for (uint32_t i = 0; i < GDMA_TRANSFER_SIZE; i++)
    {
        if (GDMA_Send_Buf[i] != GDMA_Recv_Buf[i])
        {
            APP_PRINT_INFO2("[io_gdma]      io_handle_gdma_msg:      Data      transmission      error!
GDMA_Send_Buf = %d, GDMA_Recv_Buf = %d", GDMA_Send_Buf[i], GDMA_Recv_Buf[i]);
        }
    }
}

```

在 DebugAnalyser 工具上，打印 GDMA 打印传输数据完成信息。

## 6.2 多 Block 传输（Multi Block）

当 GDMA 连续传输数据大于 4095 时，需使用 Multi Block 功能。使用 GDMA Multi Block 功能（仅 GDMA2、GDMA4 支持），实现 memory 到 memory 搬运数据。在 DebugAnalyser 工具上，打印 GDMA 搬运数据信息。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\GDMA\Mem2Mem\_multi\_block。

### 6.2.1 软件流程

根据需求，开启 GDMA block 传输完成中断（GDMA\_INT\_Block）或 GDMA 总传输完成中断（GDMA\_INT\_Transfer）。

```
#define INT_TRANSFER          0
#define INT_BLOCK              1
#define GDMA_INTERRUPT_MODE    INT_BLOCK //INT_TRANSFER
```

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_gdma\_init 函数，对 GDMA 外设进行初始化：

- 使用 GDMA 通道 2；
- 传输方向为内存到内存传输，源端地址为 GDMA\_Send\_Buffer；目的端地址为 GDMA\_Recv\_Buffer；
- 完成一次 GDMA 传输的数据个数为 200，源端地址递增，目的端地址递增，源端数据 8 位，目的端数据 8 位，源端单次 burst 传输的数据个数为 1，目的端单次 burst 传输的数据个数为 1；
- 设置每次 block 传输后自动加载 LLI 结构体中源地址与目的地址值；

- 使能 Multi-block 传输;
  - 设置传输 LLI 类型结构体首地址;
- 配置每次 block 传输后 LLI 结构体中源地址、目的地址、链表指针、控制寄存器;
- 根据需求，使能 GDMA 通道 2 GDMA\_INT\_Block 中断或 GDMA\_INT\_Transfer 中断;

```
void driver_gdma_init(void)
{
    .....
    GDMA_InitStruct.GDMA_Multi_Block_Mode = GDMA_MULTIBLOCK_MODE;
    GDMA_InitStruct.GDMA_Multi_Block_En = 1;
    GDMA_InitStruct.GDMA_Multi_Block_Struct = (uint32_t)GDMA_LLIStruct;
    for (uint32_t i = 0; i < GDMA_MULTIBLOCK_SIZE; i++)
    {
        if (i == GDMA_MULTIBLOCK_SIZE - 1)
        {
            GDMA_LLIStruct[i].SAR = (uint32_t)GDMA_Send_Buffer[i];
            GDMA_LLIStruct[i].DAR = (uint32_t)GDMA_Recv_Buffer[i];
            GDMA_LLIStruct[i].LLP = 0;
            GDMA_LLIStruct[i].CTL_LOW = BIT(0) | ....;
            GDMA_LLIStruct[i].CTL_HIGH = GDMA_InitStruct.GDMA_BufferSize;
        }
        else
        {
            GDMA_LLIStruct[i].SAR = (uint32_t)GDMA_Send_Buffer[i];
            GDMA_LLIStruct[i].DAR = (uint32_t)GDMA_Recv_Buffer[i];
            GDMA_LLIStruct[i].LLP = (uint32_t)&GDMA_LLIStruct[i + 1];
            GDMA_LLIStruct[i].CTL_LOW = BIT(0) | ....;
            GDMA_LLIStruct[i].CTL_HIGH = GDMA_InitStruct.GDMA_BufferSize;
        }
    }
    .....
}

#if(GDMA_INTERRUPT_MODE == INT_TRANSFER)
    GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Transfer, ENABLE);
#elif(GDMA_INTERRUPT_MODE == INT_BLOCK)
    GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Block, ENABLE);
#endif
```

{

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 GDMA\_Cmd 函数，使能 GDMA 传输。  
GDMA\_Cmd(GDMA\_CHANNEL\_NUM, ENABLE);

1. 如果开启 GDMA\_INT\_Block 中断：

当一次 block 传输完成时，触发 GDMA\_INT\_Block 中断，进入中断处理函数 GDMA\_Channel\_Handler：

- 失能 GDMA 通道 2 GDMA\_INT\_Block 中断；
- 定义消息类型 IO\_MSG\_TYPE\_GDMA，保存数据，发送 msg 给 task。

```
void GDMA_Channel_Handler(void)
{
    GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Block, DISABLE);

    T_IO_MSG int_gdma_msg;
    int_gdma_msg.type = IO_MSG_TYPE_GDMA;
    int_gdma_msg.subtype = 0;
    int_gdma_msg.u.buf = (void *)&GDMA_INT_Block_Counter;
    if (false == app_send_msg_to_apptask(&int_gdma_msg))
        .....
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg) 函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_GDMA，执行 io\_handle\_gdma\_msg：

- 打印 block 传输次数和传输数据完成信息；
- 判断 GDMA\_Recv\_Buf 与 GDMA\_Send\_Buf 是否相等，不等则打印错误数据（搬运出错）。
- 使能 GDMA 通道 2 GDMA\_INT\_Block 中断，直到 GDMA\_MULTIBLOCK\_SIZE 次 block 传输完成。

```
void io_handle_gdma_msg(T_IO_MSG *io_gdma_msg)
{
    uint8_t *p_buf = io_gdma_msg->u.buf;
    APP_PRINT_INFO1("[io_gdma] io_handle_msg: GDMA block%d transfer data completion!", *p_buf);
    for (uint32_t j = 0; j < GDMA_TRANSFER_SIZE; j++)
    {
        if (GDMA_Send_Buffer[*p_buf][j] != GDMA_Recv_Buffer[*p_buf][j])
        {
            APP_PRINT_INFO2("[io_gdma]io_handle_msg: Data transmission error! GDMA_Send_Buffer");
        }
    }
}
```

```
= %d, GDMA_Recv_Buffer = %d", GDMA_Send_Buffer[*p_buf][j], GDMA_Recv_Buffer[*p_buf][j]);  
    }  
    GDMA_Recv_Buffer[*p_buf][j] = 0;  
}  
GDMA_INT_Block_Counter++;  
if(GDMA_INT_Block_Counter < GDMA_MULTIBLOCK_SIZE)  
{  
    GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Block, ENABLE);  
}  
else  
{  
    GDMA_INT_Block_Counter = 0;  
}  
}
```

## 2. 如果开启 GDMA 通道 2 GDMA\_INT\_Transfer 中断:

当总传输完成时，触发 GDMA\_INT\_Transfer 中断，进入中断处理函数 GDMA\_Channel\_Handler:

- 失能 GDMA 通道 2 GDMA\_INT\_Transfer 中断；
- 定义消息类型 IO\_MSG\_TYPE\_GDMA，发送 msg 给 task；
- 清除 GDMA 通道 2 GDMA\_INT\_Transfer 中断挂起位。

```
void GDMA_Channel_Handler(void)  
{  
    GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Transfer, DISABLE);  
  
    T_IO_MSG int_gdma_msg;  
    int_gdma_msg.type = IO_MSG_TYPE_GDMA;  
    int_gdma_msg.subtype = 0;  
    if(false == app_send_msg_to_apptask(&int_gdma_msg))  
        .....  
    GDMA_ClearINTPendingBit(GDMA_CHANNEL_NUM, GDMA_INT_Transfer);  
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg) 函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_GDMA，执行 io\_handle\_gdma\_msg:

- 打印传输数据完成信息；
- 判断 GDMA\_Recv\_Buf 与 GDMA\_Send\_Buf 是否相等，不等则打印错误数据（搬运出错）。

```
void io_handle_gdma_msg(T_IO_MSG *io_gdma_msg)
```

```
{  
    APP_PRINT_INFO0("[io_gdma] io_handle_gdma_msg: GDMA transfer data completion!");  
    for (uint32_t i = 0; i < GDMA_MULTIBLOCK_SIZE; i++)  
    {  
        for (uint32_t j = 0; j < GDMA_TRANSFER_SIZE; j++)  
        {  
            if (GDMA_Send_Buffer[i][j] != GDMA_Recv_Buffer[i][j])  
            {  
                APP_PRINT_INFO2("[io_gdma]io_handle_gdma_msg: Data transmission error!  
GDMA_Send_Buffer = %d, GDMA_Recv_Buffer = %d", GDMA_Send_Buffer[i][j],  
GDMA_Recv_Buffer[i][j]);  
            }  
            GDMA_Recv_Buffer[i][j] = 0;  
        }  
    }  
}
```

在 DebugAnalyser 工具上，打印 GDMA 搬运数据信息。

## 6.3 分散和收集 (scatter&gather)

使用 GDMA Scatter&Gather 功能（仅 GDMA4 支持），实现 memory 到 memory 搬运数据。在 DebugAnalyser 工具上，打印 GDMA 搬运数据信息。

工程目录：\HoneyComb\ sdk\board\evb\_stack\_img\io\_sample\GDMA\Mem2Mem\_scatter\_gather.

### 6.3.1 软件流程

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,  
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_gdma\_init 函数，对 GDMA 外设进行初始化：

- 使用 GDMA 通道 4；
- 传输方向为内存到内存传输，源端地址为 GDMA\_Send\_Buffer；目的端地址为 GDMA\_Recv\_Buffer；
- 完成一次 GDMA 传输的数据个数为 20，源端地址递增，目的端地址递增，源端数据 8 位，目的端数据 8 位，源端单次 burst 传输的数据个数为 1，目的端单次 burst 传输的数据个数为 1；
- 使能 scatter 传输；设置 GDMA 在 scatter 传输中连续存数据的长度和地址间隔；
- 使能 gather 传输；设置 GDMA 在 gather 传输中连续取数据的长度和地址间隔；

使能 GDMA 通道 4 总传输完成中断 (GDMA\_INT\_Transfer);

```
void driver_gdma_init(void)
{
    .....
    GDMA_InitStruct.GDMA_Scatter_En      = GDMA_SCATTER_EN;
    GDMA_InitStruct.GDMA_ScatterCount    = GDMA_SCATTER_COUNT;
    GDMA_InitStruct.GDMA_ScatterInterval = GDMA_SCATTER_INTERVAL;
    GDMA_InitStruct.GDMA_Gather_En       = GDMA_GATHER_EN;
    GDMA_InitStruct.GDMA_GatherCount    = GDMA_GATHER_COUNT;
    GDMA_InitStruct.GDMA_GatherInterval = GDMA_GATHER_INTERVAL;
    .....
    GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Transfer, ENABLE);
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 GDMA\_Cmd 函数，使能 GDMA 传输。

```
GDMA_Cmd(GDMA_CHANNEL_NUM, ENABLE);
```

当 GDMA 总传输完成时，触发 GDMA\_INT\_Transfer 中断，进入中断处理函数 GDMA\_Channel\_Handler:

- 清除 GDMA 通道 4 GDMA\_INT\_Transfer 中断挂起位，失能 GDMA 外设；
- 定义消息类型 IO\_MSG\_TYPE\_GDMA，发送 msg 给 task。

```
void GDMA_Channel_Handler(void)
{
    GDMA_ClearINTPendingBit(GDMA_CHANNEL_NUM, GDMA_INT_Transfer);
    GDMA_Cmd(GDMA_CHANNEL_NUM, DISABLE);

    T_IO_MSG int_gdma_msg;
    int_gdma_msg.type = IO_MSG_TYPE_GDMA;
    int_gdma_msg.subtype = 0;
    if (false == app_send_msg_to_apptask(&int_gdma_msg))
    .....
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg) 函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_GDMA，执行 io\_handle\_gdma\_msg：

- 打印传输完成信息、源端和目的端数据大小、GDMA\_Recv\_Buffer 数据；
- 重新设置源端地址和目的端地址；

- 使能 GDMA 外设。

```
void io_handle_gdma_msg(T_IO_MSG *io_gdma_msg)
{
    APP_PRINT_INFO0("[io_gdma] io_handle_gdma_msg: GDMA transfer data completion!");
    APP_PRINT_INFO1("[io_gdma]io_handle_gdma_msg: %d", SOURCE_DATA_SIZE);
    APP_PRINT_INFO1("[io_gdma]io_handle_gdma_msg: %d", DESTINATION_DATA_SIZE);
    for (uint32_t i = 0; i < DESTINATION_DATA_SIZE; i++)
    {
        APP_PRINT_INFO2("GDMA receive data[%d] = %d", i, GDMA_Recv_Buffer[i]);
        GDMA_Recv_Buffer[i] = 0;
    }
    for (uint32_t i = 0; i < 10000000; i++);
    GDMA_SetSourceAddress(GDMA_Channel, (uint32_t)GDMA_Send_Buffer);
    GDMA_SetDestinationAddress(GDMA_Channel, (uint32_t)GDMA_Recv_Buffer);
    GDMA_Cmd(GDMA_CHANNEL_NUM, ENABLE);
}
```

在 DebugAnalyser 工具上，打印 GDMA 搬运数据信息。

## 7 模数转换器(ADC)

专用硬件设备：直流稳压源

ADC 的详细说明请参考： RTL8762C ADC 应用实例<sup>[5]</sup>。

### 7.1 单次采样模式-轮询方式

使用 ADC 外设的 one shot 模式进行电压检测，以轮询方式检测 P2\_2 输入电压，并在 DebugAnalyser 工具上，打印 16 个 ADC 通道的采样数据和采样电压。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\ADC\OneShotMode\_Polling。

#### 7.1.1 硬件设计

硬件连接：P2\_2 -> 外部电压输入。

EVB 外接直流稳压源，连接 P2\_2 和直流稳压源。P2\_2 配置为模电电压输入引脚，若 ADC 采样模式配置为 divide 模式，输入电压范围为 0~3.3V；若配置为 bypass 模式，输入电压范围为 0~0.9V。

#### 7.1.2 软件流程

通过宏定义，配置 ADC 采样模式为 divide mode 或 bypass mode。

```
#define ADC_DIVIDE_MODE          0  
#define ADC_BYPASS_MODE         1  
#define ADC_MODE_DIVIDE_OR_BYPASS ADC_DIVIDE_MODE
```

引脚定义。

```
#define ADC_SAMPLE_PIN_0          P2_2
```

执行 adc\_demo 函数，初始化电压换算公式函数。

```
void adc_demo(void)  
{  
    adc_k_status = ADC_CalibrationInit();  
    .....  
}
```

配置 PAD：设置引脚、SW 模式、PowerOn、无内部上拉、输出失能、输出低等。

```
void board_adc_init(void)  
{  
    Pad_Config(ADC_SAMPLE_PIN_0,  PAD_SW_MODE,  PAD_IS_PWRON,  PAD_PULL_NONE,  
    PAD_OUT_DISABLE, PAD_OUT_LOW);
```

{

使能 ADC 时钟；

初始化 ADC 外设：

- adcSamplePeriod 设为 255， 默认值；
- schIndex[0]~schIndex[15]设为 EXT\_SINGLE\_ENDED(ADC\_SAMPLE\_CHANNEL\_0)， 即 16 个 ADC 通道均选择 ADC 单端模式的引脚 P2\_2； schIndex[0]~schIndex[15]对应的 bitmap 为 0xFFFF；
- adcPowerAlwaysOnCmd 设为 ADC\_POWER\_ALWAYS\_ON\_ENABLE， 即模拟 power 常开；若配置为 bypass 模式， 使能相应通道的高阻态模式。

```
void driver_adc_init(void)
{
    RCC_PeriphClockCmd(APBPeriph_ADC, APBPeriph_ADC_CLOCK, ENABLE);
    ADC_InitTypeDef ADC_InitStruct;
    ADC_StructInit(&ADC_InitStruct);
    ADC_InitStruct.adcSamplePeriod      = 255;
    for (uint8_t i = 0; i < 16; i++)
    {
        ADC_InitStruct.schIndex[i] = EXT_SINGLE_ENDED(ADC_SAMPLE_CHANNEL_0);
    }
    ADC_InitStruct.bitmap            = 0xFFFF;
    ADC_InitStruct.adcPowerAlwaysOnCmd = ADC_POWER_ALWAYS_ON_ENABLE;
    ADC_Init(ADC, &ADC_InitStruct);

#if(ADC_MODE_DIVIDE_OR_BYPASS == ADC_BYPASS_MODE)
    ADC_BypassCmd(ADC_SAMPLE_CHANNEL_0, ENABLE);
#endif
}
```

使能 ADC 单次转换模式。

```
void adc_demo(void)
{
    .....
    ADC_Cmd(ADC, ADC_One_Shot_Mode, ENABLE);
}
```

循环执行 adc\_sample\_demo 函数， 每次转换完成， 重新使能 ADC 单次转换模式。

```
while (1)
{
```

```
DBG_DIRECT("[ADC] Polling reads adc sample data !");
adc_sample_demo();
for (uint32_t i = 500000; i > 0; i--);
ADC_Cmd(ADC, ADC_One_Shot_Mode, ENABLE);
}
```

adc\_sample\_demo 实现 ADC 数据采样，计算电压，打印信息：

- 循环判断 ADC 单次转换完成中断状态，RESET 时继续判断，SET 时 ADC 单次转换完成；
- 清除 ADC 单次转换完成中断挂起位；
- 读 schIndex[0]~schIndex[15]采样数据；
- 根据采样数据计算出采样电压（电压计算方法随着配置不同的 ADC 模式而不同）；
- 打印 16 个 ADC 通道的采样数据和采样电压。

```
adc_sample_demo(void)
{
    .....
    while (ADC_GetIntFlagStatus(ADC, ADC_INT_ONE_SHOT_DONE) == RESET);
    ADC_ClearINTPendingBit(ADC, ADC_INT_ONE_SHOT_DONE);
    for (uint8_t i = 0; i < 16; i++)
    {
        sample_data[i] = ADC_ReadByScheduleIndex(ADC, ADC_Schedule_Index_0 + i);
    }
    for (uint8_t i = 0; i < 16; i++)
    {
        #if (ADC_MODE_DIVIDE_OR_BYPASS == ADC_BYPASS_MODE)
            sample_voltage[i] = ADC_GetVoltage(BYPASS_SINGLE_MODE, (int32_t)sample_data[i], &error_status);
        #else
            sample_voltage[i] = ADC_GetVoltage(DIVIDE_SINGLE_MODE, (int32_t)sample_data[i], &error_status);
        #endif
        .....
        DBG_DIRECT("[ADC]adc_sample_demo: ADC one shot mode sample data_%-4d = %d, voltage_%-4d
= %fmV ", i, sample_data[i], i, sample_voltage[i]);
    }
}
```

在 DebugAnalyser 工具上，打印 16 个 ADC 通道的采样数据和采样电压。

## 7.2 单次采样模式-中断方式

使用 ADC 外设的 one shot 模式进行电压检测，以中断的方式检测 P2\_2 输入电压，并在 DebugAnalyser 工具上，打印采样数据和采样电压。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\ADC\OneShotMode。

### 7.2.1 硬件设计

硬件连接：P2\_2 -> 外部电压输入。

EVB 外接直流稳压源，连接 P2\_2 和直流稳压源。P2\_2 配置为模电电压输入引脚，若 ADC 采样模式配置为 divide 模式，输入电压范围为 0~3.3V；若配置为 bypass 模式，输入电压范围为 0~0.9V。

### 7.2.2 软件流程

初始化电压换算公式函数，初始化 ADC 全局数据。

```
void global_data_adc_init(void)
{
    adc_k_status = ADC_CalibrationInit();
    .....
    memset(&ADC_Global_Data, 0, sizeof(ADC_Global_Data));
}
```

配置 PAD，设置 P2\_2 为 ADC 功能。

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_adc\_init 函数，对 ADC 外设进行初始化：

- timerTriggerEn 设为 ENABLE，使能定时器触发采样功能；
- ADC 通道 schIndex[0]选择 ADC 单端模式的引脚 P2\_2；schIndex[0]对应的 bitmap 为 0x0001；
- 使能 ADC 单次转换完成中断（ADC\_INT\_ONE\_SHOT\_DONE）。

```
void driver_adc_init(void)
{
    .....
    ADC_InitStruct.timerTriggerEn      = ENABLE;
    ADC_InitStruct.schIndex[0]          = EXT_SINGLE_ENDED(ADC_SAMPLE_CHANNEL_0);
    ADC_InitStruct.bitmap             = 0x01;
    .....
```

```
    ADC_INTConfig(ADC, ADC_INT_ONE_SHOT_DONE, ENABLE);  
}
```

执行 driver\_adc\_timer\_init 函数，对 TIM7 外设进行初始化：

- 配置为 TIMER 模式，用户模式；
- 设置 TIM\_Period 为 1000000，TIM\_SOURCE\_DIV 为 40 分频，即定时时间为 1s。

```
void driver_adc_timer_init(void)  
{  
    .....  
  
    TIM_InitStruct.TIM_PWM_En = PWM_DISABLE;  
  
    TIM_InitStruct.TIM_Period = ADC_ONE_SHOT_SAMPLE_PERIOD - 1; //sampling once 1 second  
    TIM_InitStruct.TIM_SOURCE_DIV = TIM_CLOCK_DIVIDER_40;  
    .....  
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 adc\_sample\_start：

- 执行 ADC\_Cmd 函数：使能 ADC 单次采样模式；
- 执行 TIM\_Cmd 函数：使能 TIM7 定时功能。

```
void adc_sample_start(void)  
{  
    ADC_Cmd(ADC, ADC_ONE_SHOT_MODE, ENABLE);  
    TIM_Cmd(TIM7, ENABLE);  
}
```

定时时间到，触发 ADC 采样，当 ADC 单次转换完成时，触发 ADC\_INT\_ONE\_SHOT\_DONE 中断，进入中断处理函数 ADC\_Handler：

判断 ADC 单次转换完成中断状态为 SET 时：

- 清除 ADC\_INT\_ONE\_SHOT\_DONE 中断挂起位；
- 读 schIndex[0]采样数据；
- 定义消息类型 IO\_MSG\_TYPE\_ADC，保存数据，发送 msg 给 task。

```
void ADC_Handler(void)  
{  
    if (ADC.GetIntFlagStatus(ADC, ADC_INT_ONE_SHOT_DONE) == SET)  
    {  
        ADC_ClearINTPendingBit(ADC, ADC_INT_ONE_SHOT_DONE);  
        sample_data = ADC_ReadRawData(ADC, ADC_Schedule_Index_0);  
    }  
}
```

```
T_IO_MSG int_adc_msg;
int_adc_msg.type = IO_MSG_TYPE_ADC;
int_adc_msg.subtype = 0;
ADC_Global_Data.RawDataLen = 1;
ADC_Global_Data.RawData[0] = sample_data;
int_adc_msg.u.buf = (void *)(ADC_Global_Data.RawData);
if(false == app_send_msg_to_apptask(&int_adc_msg))
.....
}
```

```
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，利用 app\_handle\_io\_msg 函数处理。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_ADC，执行 io\_handle\_adc\_msg 函数，执行 io\_adc\_voltage\_calculate：

- 从 msg 中取出采样数据；
- 根据采样数据计算出采样电压（电压计算方法随着配置不同的 ADC 模式而不同）；
- 打印采样数据和采样电压。

```
static void io_adc_voltage_calculate(T_IO_MSG *io_adc_msg)
{
.....
ADC_Data_TypeDef *p_buf = io_adc_msg->u.buf;
sample_data_len = p_buf->RawDataLen;
for (uint8_t i = 0; i < sample_data_len; i++)
{
    sample_data = p_buf->RawData[i];
    DBG_DIRECT("io_adc_voltage_calculate: raw_data = 0x%X", sample_data);
#if(ADC_DATA_HW_AVERAGE )
    sample_data = (p_buf->RawData[i] & 0x3FFC) >> 2;
    uint16_t sample_data_decimal = (p_buf->RawData[i] & 0x3);

    float cacl_result = sample_data;
    float cacl_result_dec = 0;
    cacl_result_dec = (float)(sample_data_decimal & 0x1) / 2 + (float)((sample_data_decimal >> 1) &
0x1)
    / 2;

```

```

    cacl_result += cacl_result_dec;
    DBG_DIRECT("io_adc_voltage_calculate: sample_data = %d, cacl_result = %f\r\n",
               sample_data, cacl_result);
}

```

在 DebugAnalyser 工具上，打印采样数据和采样电压。

## 7.3 单次、差分采样模式

使用 ADC 外设的 one shot 模式进行差分电压检测，以中断的方式检测 P2\_2 和 P2\_3 输入差分电压，并在 DebugAnalyser 工具上，打印采样数据和采样电压。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ ADC\ OneShotMode\_DifferentialMode。

### 7.3.1 硬件设计

硬件连接：P2\_2 -> 外部电压输入 P 端，P2\_3 -> 外部电压输入 N 端。

EVB 外接直流稳压源，连接 P2\_2、P2\_3 和直流稳压源。P2\_2 和 P2\_3 配置为电压输入引脚，若 ADC 采样模式配置为 divide 模式，输入电压范围为 0~3.3V；若配置为 bypass 模式，输入电压范围为 0~0.9V。

### 7.3.2 软件流程

初始化电压换算公式函数，初始化 ADC 全局数据。

```

void global_data_adc_init(void)
{
    adc_k_status = ADC_CalibrationInit();
    .....
    memset(&ADC_Global_Data, 0, sizeof(ADC_Global_Data));
}

```

配置 PAD，设置 P2\_2、P2\_3 为 ADC 功能。

app\_task 初始化。

```

os_task_create(&app_task_handle, "app", app_main_task, 0, APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);

```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_adc\_init 函数，对 ADC 外设进行初始化：

- 使能定时器触发采样功能；
- ADC 通道 schIndex[0]、schIndex[1]选择 ADC 差分模式的引脚 P2\_2、P2\_3，schIndex[0] 和 schIndex[1] 对应的 bitmap 为 0x0003；
- 若配置 bypass 模式，需将差分引脚 P2\_2、P2\_3 设置为 bypass 模式；
- 使能 ADC 单次转换完成中断 (ADC\_INT\_ONE\_SHOT\_DONE)；使能 ADC 单次转换模式。

```
void driver_adc_init(void)
{
    .....
    ADC_InitStruct.timerTriggerEn      = ADC_TIMER_TRIGGER_ENABLE;
    ADC_InitStruct.schIndex[0]          = EXT_DIFFERENTIAL(ADC_SAMPLE_CHANNEL_0);
    ADC_InitStruct.schIndex[1]          = EXT_DIFFERENTIAL(ADC_SAMPLE_CHANNEL_1);
    ADC_InitStruct.bitmap             = 0x03;
    .....
#if(ADC_MODE_DIVIDE_OR_BYPASS == ADC_BYPASS_MODE)
    uint8_t temp = ADC_SAMPLE_CHANNEL_0 & 0x06;
    switch (temp)
    {
        case 0x0:
            ADC_BypassCmd(0, ENABLE);
            ADC_BypassCmd(1, ENABLE);
            break;
    }
#endif
    .....
    ADC_INTConfig(ADC, ADC_INT_ONE_SHOT_DONE, ENABLE);
    ADC_Cmd(ADC, ADC_One_Shot_Mode, ENABLE);
}
```

执行 driver\_adc\_timer\_init 函数，对 TIM7 外设进行初始化：

- 配置为 TIMER 模式，用户模式；定时时间为 1s；使能 TIM7 外设。

详情参考：ADC -> One short mode。

```
void driver_adc_timer_init(void)
{
    .....
}
```

开始任务调度。

```
os_sched_start();
```

定时时间到，触发 ADC 采样，当 ADC 单次转换完成时，触发 ADC\_INT\_ONE\_SHOT\_DONE 中断，进入中断处理函数 ADC\_Handler：判断 ADC 单次转换完成中断状态为 SET 时：

- 清除 ADC\_INT\_ONE\_SHOT\_DONE 中断挂起位；读 schIndex[0]、schIndex[1]采样数据；定义消息类型 IO\_MSG\_TYPE\_ADC，保存数据，发送 msg 给 task。

详情参考：ADC -> One short mode。

```
void ADC_Handler(void)
{
    .....
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，利用 app\_handle\_io\_msg 函数处理。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_ADC，执行 io\_handle\_adc\_msg 函数，执行 io\_adc\_voltage\_calculate：

➤ 从 msg 中取采样数据；根据采样数据计算出采样电压；打印采样数据和采样电压。

详情参考：ADC -> One short mode。

```
static void io_adc_voltage_calculate(T_IO_MSG *io_adc_msg)
{
    .....
}
```

在 DebugAnalyser 工具上，打印采样数据和采样电压。

## 7.4 单次采样模式-GDMA 搬运采样数据

使用 ADC 外设的 one shot 模式进行双通道电压检测，以 GDMA 的方式同时搬运检测 P2\_2 和 P2\_3 输入电压，并在 DebugAnalyser 工具上，打印采样数据和采样电压。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ ADC\ OneShotMode+GDMA。

### 7.4.1 硬件设计

硬件连接：P2\_2 -> 外部电压输入，P2\_3 -> 外部电压输入。

EVB 外接直流稳压源，连接 P2\_2、P2\_3 和直流稳压源。P2\_2、P2\_3 配置为模电电压输入引脚，若 ADC 采样模式配置为 divide 模式，输入电压范围为 0~3.3V；若配置为 bypass 模式，输入电压范围为 0~0.9V。

### 7.4.2 软件流程

通过宏定义，配置 ADC 采样模式为 divide mode 或 bypass mode。

```
#define ADC_DIVIDE_MODE          0
#define ADC_BYPASS_MODE           1
#define ADC_MODE_DIVIDE_OR_BYPASS ADC_DIVIDE_MODE
```

引脚定义。

```
#define ADC_SAMPLE_PIN_0         P2_2
#define ADC_SAMPLE_PIN_1         P2_3
```

配置 PAD，设置 P2\_2、P2\_3 为 ADC 功能。

使能 ADC 时钟；

初始化 ADC 外设：

- 开启 ADC 采样数据写入 FIFO；
- 设置用于触发 GDMA 的 ADC FIFO 的 burst size 大小；
- ADC 通道 schIndex[0]、schIndex[1]选择 ADC 单端模式的引脚 P2\_2、P2\_3；schIndex[0] 和 schIndex[1] 对应的 bitmap 为 0x0003；
- 设置采样周期 20ms；
- timerTriggerEn 设为 ENABLE，使能定时器触发采样功能；
- powerAlwaysOnCmd 设为 ADC\_POWER\_ALWAYS\_ON\_ENABLE，即模拟 power 常开；
- 使能 ADC 误读中断（ADC\_INT\_FIFO\_RD\_ERR）；
- 使能 ADC FIFO 溢出中断（ADC\_INT\_FIFO\_OVERFLOW）。

```
void driver_adc_init(void)
{
    .....
    ADC_InitStruct.ADC_DataWriteToFifo = ADC_DATA_WRITE_TO_FIFO_ENABLE;
    ADC_InitStruct.ADC_WaterLevel      = 0x4;
    ADC_InitStruct.ADC_SchIndex[0]     = EXT_SINGLE_ENDED(ADC_SAMPLE_CHANNEL_0);
    ADC_InitStruct.ADC_SchIndex[1]     = EXT_SINGLE_ENDED(ADC_SAMPLE_CHANNEL_1);
    ADC_InitStruct.ADC_Bitmap        = 0x03;
    ADC_InitStruct.ADC_TimerTriggerEn = ADC_TIMER_TRIGGER_ENABLE;
    ADC_InitStruct.ADC_PowerAlwaysOnEn = ADC_POWER_ALWAYS_ON_ENABLE;
    ADC_InitStruct.ADC_SampleTime     = 20 -1;
    .....
    ADC_Init(ADC, &ADC_InitStruct);
    ADC_INTConfig(ADC, ADC_INT_FIFO_RD_ERR, ENABLE);
    ADC_INTConfig(ADC, ADC_INT_FIFO_OVERFLOW, ENABLE);
    .....
}
```

执行 driver\_adc\_timer\_init 函数，对 TIM7 外设进行初始化：

- 配置为 TIMER 模式，用户模式；定时时间为 1s；使能 TIM7 外设。

详情参考：ADC -> One short mode。

```
void driver_adc_timer_init(void)
{
```

}

执行 driver\_gdma\_adc\_init 函数，对 GDMA 外设进行初始化：

- 使用 GDMA 通道 2；
- GDMA 的传输方向为外设到内存传输；
- GDMA\_SourceAddr 设置为 ADC 的 FIFO；GDMA\_DestinationAddr 设置为 ADC\_Recv\_Buffer；
- 使能 GDMA 通道 2 总传输完成中断（GDMA\_INT\_Transfer）。

```
void driver_gdma_adc_init(void)
{
    GDMA_InitStruct.GDMA_ChannelNum      = ADC_GDMA_CHANNEL_NUM;
    GDMA_InitStruct.GDMA_DIR              = GDMA_DIR_PeripheralToMemory;
    .....
    GDMA_InitStruct.GDMA_SourceAddr       = (uint32_t)&(ADC->FIFO);
    GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t)ADC_Recv_Buffer;
    .....
    GDMA_INTConfig(ADC_GDMA_CHANNEL_NUM, GDMA_INT_Transfer, ENABLE);
    .....
}
```

清除 ADC 外设 FIFO 中的数据；

使能 ADC 单次转换模式。

```
void adc_demo(void)
{
    .....
    ADC_ClearFIFO(ADC);
    ADC_Cmd(ADC, ADC_ONE_SHOT_MODE, ENABLE);
}
```

定时时间到，触发 ADC 采样；

当 ADC FIFO 数据读取错误时，触发 ADC\_INT\_FIFO\_RD\_ERR 中断，当 ADC FIFO 溢出错误时，触发 ADC\_INT\_FIFO\_OVERFLOW，进入中断处理函数 ADC\_Handler：

- 当 ADC\_INT\_FIFO\_RD\_ERR 时，打印"ADC\_INT\_FIFO\_RD\_ERR"信息，清除 ADC FIFO 数据读取错误中断挂起位；
- 当 ADC\_INT\_FIFO\_RD\_ERR 时，打印"ADC\_INT\_FIFO\_OVERFLOW"信息，清除 ADC FIFO 溢出错误中断挂起位；

```
void ADC_Handler (void)
{
```

```
.....  
if (ADC_GetINTStatus(ADC, ADC_INT_FIFO_RD_ERR) == SET)  
{  
    DBG_DIRECT("ADC_INT_FIFO_RD_ERR");  
    ADC_ClearINTPendingBit(ADC, ADC_INT_FIFO_RD_ERR);  
    // Add user code here!  
}  
  
if (ADC_GetINTStatus(ADC, ADC_INT_FIFO_OVERFLOW) == SET)  
{  
    DBG_DIRECT("ADC_INT_FIFO_OVERFLOW");  
    ADC_ClearINTPendingBit(ADC, ADC_INT_FIFO_OVERFLOW);  
    // Add user code here!  
}  
}
```

当 GDMA 搬运数据完成时，触发 GDMA\_INT\_Transfer 中断，进入 GDMA 中断处理函数 ADC\_GDMA\_Channel\_Handler:

- 失能 TIM7 外设；
- 打印 GDMA 搬运的采样数据；
- 重新设置 GDMA 搬运数据的源地址和目的端地址，传输数据数量；
- 清除 GDMA 通道 3 GDMA\_INT\_Transfer 中断挂起位；
- 重新使能 GDMA 通道 3 传输；
- 清除 ADC FIFO 中的数据；
- 使能 TIM7 外设。

```
void ADC_GDMA_Channel_Handler(void)  
{  
    TIM_Cmd(TIM7, DISABLE);  
    DBG_DIRECT("ADC_GDMA_Channel_Handler!");  
    for (uint32_t i = 0; i < GDMA_TRANSFER_SIZE; i++)  
    {  
        DBG_DIRECT("ADC_GDMA_Channel_Handler: data[%d] = %d", i, ADC_Recv_Data_Buffer[i]);  
    }  
    GDMA_SetSourceAddress(ADC_GDMA_Channel, (uint32_t)&(ADC->FIFO));  
    GDMA_SetDestinationAddress(ADC_GDMA_Channel, (uint32_t)(ADC_Recv_Data_Buffer));  
    GDMA_SetBufferSize(ADC_GDMA_Channel, GDMA_TRANSFER_SIZE);  
    GDMA_ClearINTPendingBit(ADC_GDMA_CHANNEL_NUM, GDMA_INT_Transfer);
```

```
    GDMA_Cmd(ADC_GDMA_CHANNEL_NUM, ENABLE);  
    ADC_ClearFIFO(ADC);  
    TIM_Cmd(TIM7, ENABLE);  
}
```

在 DebugAnalyser 工具上，打印采样数据。

## 7.5 单次采样模式-唤醒深度低功耗状态（DLPS）

使用 ADC 外设的 one shot 模式进行内部 VBAT 电压的检测；

同时，系统处于 IDLE 状态时，会自动进入 DLPS；当引脚 P4\_0 输入电平为低电平时，唤醒系统。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ ADC\ DLPS。

### 7.5.1 硬件设计

硬件连接： P4\_0->外部电平输入，低电平唤醒系统。

系统处于 DLPS 状态下，使用杜邦线短接 P4\_0 和 GND，唤醒系统。

ADC 通道选择 ADC 内部 VBAT 模式，对内部 VBAT 电压进行采样。

### 7.5.2 软件流程

开启 DLPS 功能。

```
#define DLPS_EN 1  
#define USE_UART_DLPS 1  
#define USE_USER_DEFINE_DLPS_EXIT_CB 1  
#define USE_USER_DEFINE_DLPS_ENTER_CB 1
```

初始化 ADC 全局数据。

```
void global_data_adc_init(void)  
{  
    IO_ADC_DLPS_Enter_Allowed = true;  
    ADC_DATA_Length = 0;  
    memset(ADC_DATA_Buffer, 0, sizeof(ADC_DATA_Buffer));  
}
```

配置 PAD、PINMUX：设置 P4\_0 为 GPIO 功能。

执行 pwr\_mgr\_init 函数，初始化电源管理：

- 注册用户进入 DLPS 回调函数 app\_enter\_dlps\_config，注册用户退出 DLPS 回调函数 app\_exit\_dlps\_config；
- 注册硬件控制回调函数 DLPS\_IO\_EnterDlpsCb 和 DLPS\_IO\_ExitDlpsCb，进入 DLPS 会保存 CPU、

PINMUX、Peripheral 等，退出 DLPS 会恢复 CPU、PINMUX、Peripheral 等；

- 设置 DLPS 模式；
- 设置唤醒 DLPS 方式为 ADC\_DLPS\_WAKEUP\_PIN 低电平唤醒。

```
void pwr_mgr_init(void)
{
    .....
    DLPS_IORegUserDlpsEnterCb(app_enter_dlps_config);
    DLPS_IORegUserDlpsExitCb(app_exit_dlps_config);
    DLPS_IORegister();
    lps_mode_set(LPM_DLPS_MODE);
    System_WakeUpPinEnable(ADC_DLPS_WAKEUP_PIN, PAD_WAKEUP_POL_LOW, 0);
    .....
}
```

在 app\_enter\_dlps\_config 中，执行 io\_adc\_dlps\_enter 函数，设置 ADC\_DLPS\_WAKEUP\_PIN 为 SW 模式，设置唤醒 DLPS 方式为 ADC\_DLPS\_WAKEUP\_PIN 低电平唤醒。

```
void io_adc_dlps_enter(void)
{
    .....
    Pad_ControlSelectValue(ADC_DLPS_WAKEUP_PIN, PAD_SW_MODE);
    System_WakeUpPinEnable(ADC_DLPS_WAKEUP_PIN, PAD_WAKEUP_POL_LOW, 0);
}
```

在 app\_exit\_dlps\_config 中，执行 io\_adc\_dlps\_exit 函数，设置 ADC\_DLPS\_WAKEUP\_PIN 为 PINMUX 模式。

```
void io_adc_dlps_exit(void)
{
    .....
    Pad_ControlSelectValue(ADC_DLPS_WAKEUP_PIN, PAD_PINMUX_MODE);
}
```

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_adc\_init 函数，对 ADC 外设进行初始化：

- samplePeriod 设为 255，默认值；
- schIndex[0]设为 INTERNAL\_VBAT\_MODE，即 ADC 通道选择 ADC 内部 VBAT 模式；
- schIndex[0]对应的 bitmap 为 0x01；

- dataAvgEn 设置为 ADC\_DATA\_AVERAGE\_ENABLE，使能单次采样模式的平均值的计算；
  - dataAvgSel 设置为 ADC\_DATA\_AVERAGE\_OF\_4，计算平均值的个数为 4 个；
  - powerAlwaysOnCmd 设为 ADC\_POWER\_ALWAYS\_ON\_ENABLE，即模拟 power 常开；
  - 使能 ADC 单次转换完成中断 (ADC\_INT\_ONE\_SHOT\_DONE)；
- 初始化中断：设置中断优先级，并使能 ADC\_IRQn 通道。

```
void driver_adc_init(void)
{
    .....
    ADC_InitStruct.ADC_SampleTime      = 255; /* (n + 1) cycle of 10MHz,n = 0~255 or n =
2048~14591 */
    ADC_InitStruct.ADC_SchIndex[0]     = INTERNAL_VBAT_MODE;
    ADC_InitStruct.ADC_Bitmap        = 0x01;
    ADC_InitStruct.ADC_DataAvgEn    = ADC_DATA_AVERAGE_ENABLE;
    ADC_InitStruct.ADC_DataAvgSel   = ADC_DATA_AVERAGE_OF_4;
    ADC_InitStruct.ADC_PowerAlwaysOnEn = ADC_POWER_ALWAYS_ON_ENABLE;
    ADC_Init(ADC, &ADC_InitStruct);
    ADC_INTConfig(ADC, ADC_INT_ONE_SHOT_DONE, ENABLE);
    NVIC_InitTypeDef NVIC_InitStruct;
    NVIC_InitStruct.NVIC_IRQChannel = ADC_IRQn;
    NVIC_InitStruct.NVIC_IRQChannelPriority = 3;
    NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
    NVIC_Init(&NVIC_InitStruct);
}
```

初始化 GPIO 外设：

- GPIO\_Pin 设置为 GPIO\_PIN\_INPUT；GPIO\_Mode 设置为 GPIO\_Mode\_IN；
- GPIO\_ITTrigger 设置为 GPIO\_INT\_Trigger\_EDGE，即边沿触发中断；
- GPIO\_ITPolarity 设置为 GPIO\_INT\_POLARITY\_ACTIVE\_LOW，即下降沿触发；
- ITDebounce 设置为 GPIO\_INT\_DEBOUNCE\_ENABLE，即使能 GPIO 中断去抖动；
- GPIO\_DebounceTime 设置为 64，即去抖时间设置为 64ms；

初始化中断：设置中断优先级，并使能 GPIO\_PIN\_INPUT\_IRQN 通道；

取消屏蔽 GPIO 外部中断；使能 GPIO 中断。

```
void driver_gpio_init(void)
{
    .....
    GPIO_InitTypeDef GPIO_InitStruct;
```

```
GPIO_StructInit(&GPIO_InitStruct);
GPIO_InitStruct.GPIO_Pin      = GPIO_PIN_INPUT;
GPIO_InitStruct.GPIO_Mode     = GPIO_Mode_IN;
GPIO_InitStruct.GPIO_ITTrigger = GPIO_INT_Trigger_EDGE;
GPIO_InitStruct.GPIO_ITPolarity = GPIO_INT_POLARITY_ACTIVE_LOW;
GPIO_InitStruct.GPIO_ITDebounce = GPIO_INT_DEBOUNCE_ENABLE;
GPIO_InitStruct.GPIO_DebounceTime = 10; /* unit:ms , can be 1~64 ms */
GPIO_Init(&GPIO_InitStruct);

NVIC_InitTypeDef NVIC_InitStruct;
NVIC_InitStruct.NVIC_IRQChannel = GPIO_PIN_INPUT_IRQN;
NVIC_InitStruct.NVIC_IRQChannelPriority = 3;
NVIC_InitStruct.NVIC_IRQChannelCmd = ENABLE;
NVIC_Init(&NVIC_InitStruct);

GPIO_MaskINTConfig(GPIO_PIN_INPUT, DISABLE);
GPIO_INTConfig(GPIO_PIN_INPUT, ENABLE);
}
```

开始任务调度。

```
os_sched_start();
```

当 ADC\_DLPS\_WAKEUP\_PIN 接入低电平后，系统退出 DLPS 状态，触发 GPIO\_PIN\_INPUT 中断，进入中断处理函数 GPIO\_Input\_Handler:

- 关 GPIO 中断，屏蔽 GPIO 中断；
- 定义消息类型 IO\_MSG\_TYPE\_GPIO，发送 msg 给 task；
- 清除 GPIO 中断挂起位，取消屏蔽 GPIO 中断，使能 GPIO 中断。

```
void GPIO_Input_Handler(void)
{
    GPIO_INTConfig(GPIO_PIN_INPUT, DISABLE);
    GPIO_MaskINTConfig(GPIO_PIN_INPUT, ENABLE);
    T_IO_MSG int_gpio_msg;

    int_gpio_msg.type = IO_MSG_TYPE_GPIO;
    int_gpio_msg.subtype = 0;
    if (false == app_send_msg_to_apptask(&int_gpio_msg))
        .....
}
```

```
GPIO_ClearINTPendingBit(GPIO_PIN_INPUT);  
GPIO_MaskINTConfig(GPIO_PIN_INPUT, DISABLE);  
GPIO_INTConfig(GPIO_PIN_INPUT, ENABLE);  
}
```

当 ADC 单次转换完成时, 触发 ADC\_INT\_ONE\_SHOT\_DONE 中断, 进入中断处理函数 ADC\_Handler:

判断 ADC 单次转换完成中断状态为 SET 时:

- 清除 ADC\_INT\_ONE\_SHOT\_DONE 中断挂起位; 读 schIndex[0]采样数据; 定义消息类型 IO\_MSG\_TYPE\_ADC, 保存数据, 发送 msg 给 task。

详情参考: ADC -> One short mode。

```
void ADC_Handler(void)  
{  
    .....  
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时, 执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中, 判断消息类型为 IO\_MSG\_TYPE\_GPIO, 执行 io\_adc\_sample\_start 函数, 使能 ADC 单次采样模式:

```
void io_adc_sample_start(void)  
{  
    ADC_Cmd(ADC, ADC_ONE_SHOT_MODE, ENABLE);  
}
```

在 app\_handle\_io\_msg 函数中, 判断消息类型为 IO\_MSG\_TYPE\_ADC, 执行 io\_handle\_adc\_msg 函数, 执行 io\_adc\_voltage\_calculate:

- 从 msg 中取采样数据; 根据采样数据计算出采样电压; 打印采样数据和采样电压。
- 详情参考: ADC -> One short mode。

```
void io_adc_handle_msg(T_IO_MSG *io_adc_msg)  
{  
    .....  
}
```

系统处于 IDLE 状态时, 会自动进入 DLPS;

ADC\_DLPS\_WAKEUP\_PIN 接入低电平时, 唤醒 DLPS, 执行 System\_Handler:

- 清除 ADC\_DLPS\_WAKEUP\_PIN 的唤醒中断挂起位;
- 失能 DC\_DLPS\_WAKEUP\_PIN 的唤醒功能;
- IO\_ADC\_DLPS\_Enter\_Allowed 值设置为 false。

```
void System_Handler(void)  
{
```

```
if (System_WakeUpInterruptValue(ADC_DLPS_WAKEUP_PIN) == SET)
{
    APP_PRINT_INFO0("System_Handler");
    Pad_ClearWakeUpINTPendingBit(ADC_DLPS_WAKEUP_PIN);
    System_WakeUpPinDisable(ADC_DLPS_WAKEUP_PIN);
    IO_ADC_DLPS_Enter_Allowed = false;
}
}
```

系统默认进入 DLPS 状态，在 DLPS 状态下，P4\_0 输入低电平唤醒系统，开启 ADC 检测内部 VBAT 电压模式，在 DebugAnalyser 工具上，打印采样数据。

## 7.6 连续采样模式

使用 ADC 外设的 continuous 模式进行双通道电压检测，以中断的方式同时检测 P2\_2 和 P2\_3 输入电压，并在 DebugAnalyser 工具上，打印采样数据和采样电压。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ ADC\ ContinuousMode。

### 7.6.1 硬件设计

硬件连接：P2\_2 -> 外部电压输入，P2\_3 -> 外部电压输入。

EVB 外接直流稳压源，连接 P2\_2、P2\_3 和直流稳压源。P2\_2、P2\_3 配置为模电电压输入引脚，若 ADC 采样模式配置为 divide 模式，输入电压范围为 0~3.3V；若配置为 bypass 模式，输入电压范围为 0~0.9V。

### 7.6.2 软件流程

初始化电压换算公式函数，初始化 ADC 全局数据。

```
void global_data_adc_init(void)
{
    adc_k_status = ADC_CalibrationInit();
    .....
    memset(&ADC_Global_Data, 0, sizeof(ADC_Global_Data));
}
```

配置 PAD，设置 P2\_2、P2\_3 为 ADC 功能；

app\_task 初始化。

```
os_task_create(&app_task_handle, "app", app_main_task, 0, APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_adc\_init 函数，对 ADC 外设进行初始化：

- ADC 通道 schIndex[0]、schIndex[1]选择 ADC 差分模式的引脚 P2\_2、P2\_3，schIndex[0]和 schIndex[1]对应的 bitmap 为 0x0003；
- 设置连续采样率为 199，实际采样时间为  $(199+1) / 10K = 20\text{ms}$ ；
- 设置 FIFO 阈值为 16；
- 使能 ADC FIFO 超过给定值中断 (ADC\_INT\_FIFO\_TH)；使能 ADC 连续转换模式。

```
void driver_adc_init(void)
{
    .....
    ADC_InitStruct.schIndex[0]      = EXT_SINGLE_ENDED(ADC_SAMPLE_CHANNEL_0);
    ADC_InitStruct.schIndex[1]      = EXT_SINGLE_ENDED(ADC_SAMPLE_CHANNEL_1);
    ADC_InitStruct.bitmap          = 0x03;
    ADC_InitStruct.adcSamplePeriod = ADC_CONTINUOUS_SAMPLE_PERIOD;
    ADC_InitStruct.adcFifoThd     = 16;
    .....
    ADC_INTConfig(ADC, ADC_INT_FIFO_TH, ENABLE);
    ADC_Cmd(ADC, ADC_Continuous_Mode, ENABLE);
}
```

开始任务调度。

```
os_sched_start();
```

当 ADC FIFO 数目超过阈值时，触发 ADC\_INT\_FIFO\_TH 中断，进入中断处理函数 ADC\_Handler；判断 ADC FIFO 数目超过给定值的中断状态为 SET 时：

- 失能 ADC 连续转换模式；
- 获取 FIFO 长度及 FIFO 中的采样数据；
- 清除 FIFO，清除 ADC\_INT\_FIFO\_TH 中断挂起位；
- 定义消息类型 IO\_MSG\_TYPE\_ADC，保存数据，发送 msg 给 task。

```
void ADC_Handler(void)
{
    if(ADC.GetIntFlagStatus(ADC, ADC_INT_FIFO_TH) == SET)
    {
        ADC_Cmd(ADC, ADC_Continuous_Mode, DISABLE);
        data_len = ADC_GetFifoLen(ADC);
        ADC_GetFifoData(ADC, sample_data, data_len);
        ADC_ClearFifo(ADC);
        ADC_ClearINTPendingBit(ADC, ADC_INT_FIFO_TH);
}
```

```

T_IO_MSG int_adc_msg;
int_adc_msg.type = IO_MSG_TYPE_ADC;
int_adc_msg.subtype = 0;
ADC_Global_Data.RawData[0] = data_len;
for (uint8_t i = 0; i < data_len; i++)
{
    ADC_Global_Data.RawData[i + 1] = sample_data[i];
}
int_adc_msg.u.buf = (void*)(ADC_Global_Data.RawData);
if (false == app_send_msg_to_apptask(&int_adc_msg))
.....
}
}

```

app\_main\_task 循环检测 msg queue。当有 msg 时，利用 app\_handle\_io\_msg 函数处理。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_ADC，执行 io\_handle\_adc\_msg 函数，执行 io\_adc\_voltage\_calculate：

- 从 msg 中取出采样数据；
- 根据采样数据计算出采样电压（电压计算方法随着配置不同的 ADC 模式而不同）；
- 打印采样数据和采样电压。
- 重新使能 ADC 连续转换模式。

```

static void io_adc_voltage_calculate(T_IO_MSG *io_adc_msg)
{
    .....
    uint16_t *p_buf = io_adc_msg->u.buf;
    sample_data_len = p_buf[0];
    for (uint8_t i = 0; i < sample_data_len; i++)
    {
        sample_data[i] = p_buf[i + 1];
    }
    for (uint8_t i = 0; i < sample_data_len; i++)
    {
#if(ADC_MODE_DIVIDE_OR_BYPASS == ADC_BYPASS_MODE)
        sample_voltage[i] = ADC_GetVoltage(BYPASS_SINGLE_MODE, (int32_t)sample_data[i], &error_status);
#else
        sample_voltage[i] = ADC_GetVoltage(DIVIDE_SINGLE_MODE, (int32_t)sample_data[i], &error_status);

```

```
#endif
.....
APP_PRINT_INFO4("[io_adc] io_adc_voltage_calculate: ADC rawdata_%-4d = %d, voltage_%-4d
= %dmV ", i, sample_data[i], i, (uint32_t)sample_voltage[i]);
ADC_Cmd(ADC, ADC_Continuous_Mode, ENABLE);
}
```

在 DebugAnalyser 工具上，打印采样数据和采样电压。

## 7.7 连续采样模式-GDMA 搬运采样数据

使用 ADC 外设的 continuous 模式进行双通道电压检测，以 GDMA 的方式同时检测 P2\_2 和 P2\_3 输入电压，并在 DebugAnalyser 工具上，打印采样数据和采样电压。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ ADC\ ContinuousMode+GDMA。

### 7.7.1 硬件设计

硬件连接：P2\_2 -> 外部电压输入，P2\_3 -> 外部电压输入。

EVB 外接直流稳压源，连接 P2\_2、P2\_3 和直流稳压源。P2\_2、P2\_3 配置为模电电压输入引脚，若 ADC 采样模式配置为 divide 模式，输入电压范围为 0~3.3V；若配置为 bypass 模式，输入电压范围为 0~0.9V。

### 7.7.2 软件流程

初始化电压换算公式函数，初始化 ADC 全局数据。

```
void global_data_adc_init(void)
{
    adc_k_status = ADC_CalibrationInit();
    .....
    memset(&ADC_Global_Data, 0, sizeof(ADC_Global_Data));
}
```

配置 PAD，设置 P2\_2、P2\_3 为 ADC 功能；

app\_task 初始化。

```
os_task_create(&app_task_handle, "app", app_main_task, 0, APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_adc\_init 函数，对 ADC 外设进行初始化：

- ADC 通道 schIndex[0]、schIndex[1]选择 ADC 差分模式的引脚 P2\_2、P2\_3；对应的 bitmap 为 0x0003；采样周期为 20ms；设置 FIFO 阈值为 16；
- 使能 ADC 误读中断 (ADC\_INT\_FIFO\_RD\_ERR)。

```
void driver_adc_init(void)
{
    .....
    ADC_INTConfig(ADC, ADC_INT_FIFO_RD_ERR, ENABLE);
    .....
}
```

执行 `driver_gdma_adc_init` 函数，对 GDMA 外设进行初始化：

- 使用 GDMA 通道 2；
- GDMA 的传输方向为外设到内存传输； `GDMA_SourceAddr` 设置为 ADC 的 FIFO； `GDMA_DestinationAddr` 设置为 `ADC_Recv_Buffer`；
- 使能 GDMA 通道 2 总传输完成中断（`GDMA_INT_Transfer`）。

```
void driver_gdma_adc_init(void)
{
    GDMA_InitStruct.GDMA_ChannelNum      = ADC_GDMA_CHANNEL_NUM;
    GDMA_InitStruct.GDMA_DIR             = GDMA_DIR_PeripheralToMemory;
    .....
    GDMA_InitStruct.GDMA_SourceAddr     = (uint32_t)&(ADC->FIFO);
    GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t)ADC_Recv_Buffer;
    .....
    GDMA_INTConfig(ADC_GDMA_CHANNEL_NUM, GDMA_INT_Transfer, ENABLE);
    .....
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 `app_handle_dev_state_evt` 函数：

- 执行 `GDMA_Cmd` 函数：使能 GDMA 传输；
- 执行 `ADC_Cmd` 函数：使能 ADC 连续转换模式。

```
GDMA_Cmd(ADC_GDMA_CHANNEL_NUM, ENABLE);
ADC_Cmd(ADC, ADC_Continuous_Mode, ENABLE);
```

当 ADC FIFO 数据读取错误时，触发 `ADC_INT_FIFO_RD_ERR` 中断，进入中断处理函数 `ADC_Handler`； 定义消息类型 `IO_MSG_TYPE_ADC`，消息子类型为 `IO_MSG_ADC_FIFO_READ_ERR`，发送 msg 给 task。

```
void ADC_Handler(void)
{
    if(ADC_GetIntFlagStatus(ADC, ADC_INT_FIFO_RD_ERR) == SET)
    {
```

```
T_IO_MSG int_adc_msg;
int_adc_msg.type = IO_MSG_TYPE_ADC;
int_adc_msg.subtype = IO_MSG_ADC_FIFO_READ_ERR;
if(false == app_send_msg_to_apptask(&int_adc_msg))
.....
}
```

当 GDMA 搬运数据完成时，触发 GDMA\_INT\_Transfer 中断，进入 GDMA 中断处理函数 ADC\_GDMA\_Channel\_Handler:

- 失能 ADC 连续采样模式；清除 GDMA 通道 2 GDMA\_INT\_Transfer 中断挂起位；
- 定义消息类型 IO\_MSG\_TYPE\_GDMA，保存 ADC\_Recv\_Buffer 数据，发送 msg 给 task。

```
void ADC_GDMA_Channel_Handler(void)
{
    ADC_Cmd(ADC, ADC_Continuous_Mode, DISABLE);
    GDMA_ClearINTPendingBit(ADC_GDMA_CHANNEL_NUM, GDMA_INT_Transfer);

    T_IO_MSG int_gdma_msg;
    int_gdma_msg.type = IO_MSG_TYPE_GDMA;
    int_gdma_msg.subtype = 0;
    int_gdma_msg.u.buf = (void*)(ADC_Recv_Buffer);
    if(false == app_send_msg_to_apptask(&int_gdma_msg))
    .....
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，利用 app\_handle\_io\_msg 函数处理。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_ADC，执行 io\_handle\_adc\_msg 函数，执行 io\_handle\_adc\_error：打印 ADC 错误信息。

```
static void io_handle_adc_error(T_IO_MSG *io_adc_msg)
{
    .....
    case IO_MSG_ADC_FIFO_READ_ERR:
    {
        APP_PRINT_ERROR0("[io_adc]io_handle_adc_error: IO_MSG_ADC_FIFO_READ_ERR!");
    }
    .....
}
```

{

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_GDMA，执行 io\_handle\_gdma\_msg 函数，执行 io\_adc\_voltage\_calculate：

- 从 msg 中取出采样数据；
- 根据采样数据计算出采样电压（电压计算方法随着配置不同的 ADC 模式而不同）；
- 打印采样数据和采样电压。
- 重新使能 GDMA 传输，使能 ADC 连续采样模式。

```
static void io_adc_voltage_calculate(T_IO_MSG *io_adc_msg)
{
    .....
    uint16_t *p_buf = io_adc_msg->u.buf;
    for (uint32_t i = 0; i < GDMA_TRANSFER_SIZE; i++)
    {
        sample_data[i] = p_buf[i];
    }
    for (uint32_t i = 0; i < GDMA_TRANSFER_SIZE; i++)
    {
        #if(ADC_MODE_DIVIDE_OR_BYPASS == ADC_BYPASS_MODE)
            sample_voltage[i] = ADC_GetVoltage(BYPASS_SINGLE_MODE, (int32_t)sample_data[i],
            &error_status);
        #else
            sample_voltage[i] = ADC_GetVoltage(DIVIDE_SINGLE_MODE, (int32_t)sample_data[i],
            &error_status);
        #endif
        .....
        APP_PRINT_INFO4("[io_adc]io_adc_voltage_calculate: ADC rawdata_%-4d = %d, voltage_%-4d
= %dmV ", i, sample_data[i], i, (uint32_t)sample_voltage[i]);
    }
    for (uint32_t i = 0; i < 1000000; i++);
    GDMA_Cmd(ADC_GDMA_CHANNEL_NUM, ENABLE);
    ADC_Cmd(ADC, ADC_Continuous_Mode, ENABLE);
}
```

在 DebugAnalyser 工具上，打印采样数据和采样电压。

## 8 串型外设接口(SPI)

专用硬件设备：FM25Q16 模块、APS6404L 模块

### 8.1 EEPROM 模式读取 ID

使用 SPI0 与 FM25Q16 以 EEPROM 模式进行数据传输。通过轮询方式获取 FM25Q16 ID，在 DebugAnalyser 工具上，打印 Flash ID 信息。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ SPI\ Polling\_eeprom。

#### 8.1.1 硬件设计

硬件连接：P4\_0 -> CLK, P4\_1 -> DI, P4\_2 -> DO, P4\_3 -> CS#。

EVB 外接 FM25Q16 模块，连接 P4\_0 和 CLK, P4\_1 和 DI, P4\_2 和 DO, P4\_3 和 CS#, 同时连接 VDD 和模块的 VDD、WP#、HOLD#, 连接 GND 和模块的 VSS。

#### 8.1.2 软件流程

引脚定义：

```
#define SPI0_SCK_PIN      P4_0  
#define SPI0_MOSI_PIN     P4_1  
#define SPI0_MISO_PIN     P4_2  
#define SPI0_CS_PIN       P4_3
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出使能、输出高；

配置 PINMUX：分配引脚分别为 SPI0\_CLK\_MASTER、SPI0\_MO\_MASTER、SPI0\_MI\_MASTER、SPI0\_SS\_N\_0\_MASTER 功能。

```
void board_spi_init(void)  
{  
    Pad_Config(SPI0_SCK_PIN,      PAD_PINMUX_MODE,      PAD_IS_PWRON,      PAD_PULL_UP,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);  
  
    Pad_Config(SPI0_MOSI_PIN,     PAD_PINMUX_MODE,      PAD_IS_PWRON,      PAD_PULL_UP,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);  
  
    Pad_Config(SPI0_MISO_PIN,     PAD_PINMUX_MODE,      PAD_IS_PWRON,      PAD_PULL_UP,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);  
  
    Pad_Config(SPI0_CS_PIN,       PAD_PINMUX_MODE,      PAD_IS_PWRON,      PAD_PULL_UP,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);
```

```
Pinmux_Config(SPI0_SCK_PIN, SPI0_CLK_MASTER);
Pinmux_Config(SPI0_MOSI_PIN, SPI0_MO_MASTER);
Pinmux_Config(SPI0_MISO_PIN, SPI0_MI_MASTER);
Pinmux_Config(SPI0_CS_PIN, SPI0_SS_N_0_MASTER);
}
```

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_spi\_init 函数，对 SPI0 外设进行初始化：

- SPI\_Direction 设置为 SPI\_Direction\_EEPROM，即 SPI 的数据传输模式为 EEPROM 模式；
- SPI\_Mode 设置为 SPI\_Mode\_Master，即 SPI 的工作模式为主设备模式；
- SPI\_DataSize 设置为 SPI\_DataSize\_8b，即 SPI 的数据大小为 8 位帧结构；
- SPI\_CPOL 设置为 SPI\_CPOL\_High，即串行时钟的稳态为时钟悬空高；
- SPI\_CPHA 设置为 SPI\_CPHA\_2Edge，即位捕获的时钟活动沿为数据捕获于第二个时钟沿；
- SPI\_BaudRatePrescaler 设置为 100，即 SPI 外设的时钟分频系数为 100；
- SPI\_RxThresholdLevel 设置为 1，即发送 FIFO 的阈值为 2；
- SPI\_NDF 设置为 0，即读取的数据长度阈值为 1；
- SPI\_FrameFormat 设置为 SPI\_Frame\_Motorola，即 SPI 的数据传输格式为 Motorola 传输格式；使能 SPI0 外设。

```
void driver_spi_init(void)
{
    .....
    SPI_InitTypeDef SPI_InitStruct;
    SPI_StructInit(&SPI_InitStruct);
    SPI_InitStruct.SPI_Direction = SPI_Direction_EEPROM;
    SPI_InitStruct.SPI_Mode = SPI_Mode_Master;
    SPI_InitStruct.SPI_DataSize = SPI_DataSize_8b;
    SPI_InitStruct.SPI_CPOL = SPI_CPOL_High;
    SPI_InitStruct.SPI_CPHA = SPI_CPHA_2Edge;
    SPI_InitStruct.SPI_BaudRatePrescaler = 100;
    SPI_InitStruct.SPI_RxThresholdLevel = 2 - 1;
    SPI_InitStruct.SPI_NDF = 1 - 1;
    SPI_InitStruct.SPI_FrameFormat = SPI_Frame_Motorola;
    SPI_Init(SPI0, &SPI_InitStruct);
    SPI_Cmd(SPI0, ENABLE);
```

{

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 spi\_demo：

- 依次以 EEPROM 模式读取 DEVICE\_ID、MF\_DEVICE\_ID 以及 JEDEC\_ID 的数据信息；
- 打印 Flash\_ID 的类型、长度和 ID 数据；
- 以 SPI\_FLASH\_READ\_DATA 方式读取 Flash 中的数据信息并打印；
- 以 SPI\_FLASH\_FAST\_READ 方式读取 Flash 中的数据信息并打印。

```
void spi_demo(void)
```

{

```
    uint8_t flash_id[10] = {0};
```

```
    while (flash_id_type < 3)
```

{

```
        spi_flash_read_id((Flash_ID_Type)flash_id_type, flash_id);
```

```
        APP_PRINT_INFO3("[io_spi] spi_demo: flash_id_type = %d, data_lenth = %d, flash_id = %b ",  
                      flash_id_type, flash_id[0], TRACE_BINARY(flash_id[0], &flash_id[1]));
```

```
        flash_id_type++;
```

```
        memset(flash_id, 0, sizeof(flash_id));
```

}

```
    flash_id_type = 0;
```

```
    uint8_t read_data[100] = {0};
```

```
    spi_flash_read(SPI_FLASH_READ_DATA, 0x000101, read_data, 100);
```

```
    APP_PRINT_INFO1("[io_spi] spi_demo: read_data = %b,", TRACE_BINARY(100, read_data));
```

```
    spi_flash_read(SPI_FLASH_FAST_READ, 0x000210, read_data, 100);
```

```
    APP_PRINT_INFO1("[io_spi] spi_demo: read_data = %b,", TRACE_BINARY(100, read_data));
```

}

在 DebugAnalyser 工具上，依此打印 Flash ID 以及 Flash 内部数据等信息。

## 8.2 FullDuplex 模式读取 ID

使用 SPI0 与 FM25Q16 以 FullDuplex 模式进行数据传输。通过轮询方式获取 FM25Q16 ID，在 DebugAnalyser 工具上，打印 Flash ID 信息。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img \io\_sample\ SPI\ Polling\_fullduplex。

## 8.2.1 硬件设计

硬件连接: P4\_0 -> CLK, P4\_1 -> DI, P4\_2 -> DO, P4\_3 -> CS#。

EVB 外接 FM25Q16 模块, 连接 P4\_0 和 CLK, P4\_1 和 DI, P4\_2 和 DO, P4\_3 和 CS#, 同时连接 VDD 和模块的 VDD、WP#、HOLD#, 连接 GND 和模块的 VSS。

## 8.2.2 软件流程

引脚定义:

```
#define SPI0_SCK_PIN      P4_0  
#define SPI0_MOSI_PIN     P4_1  
#define SPI0_MISO_PIN     P4_2  
#define SPI0_CS_PIN       P4_3
```

配置 PAD: 设置引脚、PINMUX 模式、PowerOn、内部上拉、输出使能、输出高;

配置 PINMUX: 分配引脚分别为 SPI0\_CLK\_MASTER、SPI0\_MO\_MASTER、SPI0\_MI\_MASTER、  
SPI0\_SS\_N\_0\_MASTER 功能 (详情参考: SPI -> EEPROM)。

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,  
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数, 执行 driver\_spi\_init 函数, 对 SPI0 外设进行初始化:

- SPI\_Direction 设置为 SPI\_Direction\_EEPROM, 即 SPI 的数据传输模式为 FullDuplex 模式;
- SPI\_Mode 设置工作模式为主设备模式; 数据大小为 8 位帧结构; 串行时钟的稳态为时钟悬空高;  
位捕获的时钟活动沿为数据捕获于第二个时钟沿; 时钟分频系数为 100;
- SPI\_RxThresholdLevel 设置为 2, 即发送 FIFO 的阈值为 3;
- SPI\_NDF 设置为 2, 即读取的数据长度阈值为 3;
- SPI\_FrameFormat 设置为 SPI\_Frame\_Motorola, 即 SPI 的数据传输格式为 Motorola 传输格式;  
使能 SPI0 外设。

```
void driver_spi_init(void)  
{  
    .....  
    SPI_InitTypeDef  SPI_InitStruct;  
    SPI_StructInit(&SPI_InitStruct);  
    SPI_InitStruct.SPI_Direction = SPI_Direction_FullDuplex;  
    SPI_InitStruct.SPI_Mode     = SPI_Mode_Master;  
    SPI_InitStruct.SPI_DataSize = SPI_DataSize_8b;
```

```
SPI_InitStruct.SPI_CPOL      = SPI_CPOL_High;
SPI_InitStruct.SPI_CPHA       = SPI_CPHA_2Edge;
SPI_InitStruct.SPI_BaudRatePrescaler = 100;
SPI_InitStruct.SPI_RxThresholdLevel = 3 - 1;
SPI_InitStruct.SPI_NDF        = 3 - 1;
SPI_InitStruct.SPI_FrameFormat = SPI_Frame_Motorola;
SPI_Init(SPI0, &SPI_InitStruct);
SPI_Cmd(SPI0, ENABLE);
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 spi\_demo：

- 依次以 FullDuplex 模式读取 DEVICE\_ID、MF\_DEVICE\_ID 以及 JEDEC\_ID 的长度以及数据；
- 打印 Flash\_ID 的类型、长度和 ID 数据；
- 以 SPI\_FLASH\_READ\_DATA 方式读取 Flash 中的数据信息并打印；
- 以 SPI\_FLASH\_FAST\_READ 方式读取 Flash 中的数据信息并打印。

```
void spi_demo(void)
{
    uint8_t flash_id[10] = {0};
    while (flash_id_type < 3)
    {
        spi_flash_read_id((Flash_ID_Type)flash_id_type, flash_id);
        APP_PRINT_INFO3("[io_spi] spi_demo: flash_id_type = %d, data_lenth = %d, flash_id = %b ",
                      flash_id_type, flash_id[0], TRACE_BINARY(flash_id[0], &flash_id[1]));
        flash_id_type++;
        memset(flash_id, 0, sizeof(flash_id));
    }
    flash_id_type = 0;

    uint8_t read_data[100] = {0};
    spi_flash_read(SPI_FLASH_READ_DATA, 0x000101, read_data, 100);
    APP_PRINT_INFO1("[io_spi] spi_demo: read_data = %b.", TRACE_BINARY(100, read_data));
    spi_flash_read(SPI_FLASH_FAST_READ, 0x000210, read_data, 100);
    APP_PRINT_INFO1("[io_spi] spi_demo: read_data = %b.", TRACE_BINARY(100, read_data));
}
```

在 DebugAnalyser 工具上，打印依此打印 Flash ID 以及读取数据信息。

## 8.3 中断方式读取 ID

使用 SPI0 与 FM25Q16 进行数据传输。获取 FM25Q16 ID，通过 SPI0 中断向 app\_task 发送消息事件，app\_task 检测到消息事件，在 app 层解析消息，并执行用户程序：在 DebugAnalyser 工具上，打印 Flash ID 信息。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\SPI\Interrupt。

### 8.3.1 硬件设计

硬件连接：P4\_0 -> CLK, P4\_1 -> DI, P4\_2 -> DO, P4\_3 -> CS#。

EVB 外接 FM25Q16 模块，连接 P4\_0 和 CLK, P4\_1 和 DI, P4\_2 和 DO, P4\_3 和 CS#, 同时连接 VDD 和模块的 VDD、WP#、HOLD#, 连接 GND 和模块的 VSS。

### 8.3.2 软件流程

引脚定义：

```
#define SPI0_SCK_PIN      P4_0
#define SPI0_MOSI_PIN      P4_1
#define SPI0_MISO_PIN      P4_2
#define SPI0_CS_PIN        P4_3
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出使能、输出高；

配置 PINMUX：分配引脚分别为 SPI0\_CLK\_MASTER、SPI0\_MO\_MASTER、SPI0\_MI\_MASTER、  
SPI0\_SS\_N\_0\_MASTER 功能（详情参考：SPI -> EEPROM）。

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_spi\_init 函数，对 SPI0 外设进行初始化：

- SPI 的数据传输模式为 EEPROM 模式；工作模式为主设备模式；串行时钟的稳态为时钟悬空高；  
捕获的时钟活动沿为数据捕获于第二个时钟沿；时钟分频系数为 100；送 FIFO 的阈值为 1；读取  
的数据长度阈值为 1；数据传输格式为 Motorola 传输格式（详情参考：SPI -> Polling\_eeprom）；  
使能 SPI0 外设；使能 SPI0 接收缓冲区已满中断（SPI\_INT\_RXF）；

```
void driver_spi_init(void)
{
    .....
    SPI_Cmd(SPI0, ENABLE);
```

```
SPI_INTCConfig(SPI0, SPI_INT_RXF, ENABLE);  
.....  
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 spi\_demo：

- 获取 DEVICE\_ID 信息。

```
void spi_demo(void)  
{  
    .....  
    spi_flash_read_id(DEVICE_ID, id);  
}
```

当 SPI0 接收缓冲区已满时（即收到 1bytes 数据），触发 SPI\_INT\_RXF 中断，进入中断处理函数 SPI0\_Handler：

- 清除 SPI\_INT\_RXF 中断挂起位；
- 判断直到满足 FIFO 中数据长度等于 Flash\_ID 长度；
- 获取 FIFO 中 Flash\_ID 数据长度；
- 定义消息类型 IO\_MSG\_TYPE\_SPI，保存数据，发送 msg 给 task。

```
void SPI0_Handler(void)  
{  
    .....  
    SPI_ClearINTPendingBit(SPI0, SPI_INT_RXF);  
  
    while (SPI_GetRxFIFOLen(SPI0) < Flash_ID_Length);  
    data_len = SPI_GetRxFIFOLen(SPI0);  
    Flash_Data[0] = data_len;  
    for (uint8_t i = 0; i < data_len; i++)  
    {  
        Flash_Data[1 + i] = SPI_ReceiveData(SPI0);  
    }  
  
    T_IO_MSG int_spi_msg;  
    int_spi_msg.type = IO_MSG_TYPE_SPI;  
    int_spi_msg.subtype = 0;  
    int_spi_msg.u.buf = (void*)(Flash_Data);
```

```
if (false == app_send_msg_to_apptask(&int_spi_msg))  
.....  
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_SPI，执行 io\_spi\_handle\_msg：

- 打印 Flash ID 信息；
- 读取 Flash ID 的其他类型 ID 信息，通过中断方式依此打印获取 ID 信息。

```
void io_spi_handle_msg(T_IO_MSG *io_spi_msg)  
{  
    uint8_t *p_buf = io_spi_msg->u.buf;  
    uint8_t data_lenth = p_buf[0];  
    APP_PRINT_INFO2("[io_spi] io_spi_handle_msg: data_lenth = %d, data = %b ", data_lenth,  
                  TRACE_BINARY(data_lenth, &p_buf[1]));      for (uint8_t i = 0; i < data_lenth; i++)  
    {  
        APP_PRINT_INFO1("[io_spi]io_spi_handle_msg: data = 0x%x ", p_buf[i + 1]);  
    }  
    .....  
    if (flash_id_type < 3)  
    {  
        spi_flash_read_id((Flash_ID_Type)flash_id_type, id);  
    }  
}
```

在 DebugAnalyser 工具上，依此打印 Flash ID 信息。

## 8.4 闪存 (Flash)

使用 SPI0 与 FM25Q16 进行数据传输。获取 FM25Q16 ID，通过 SPI0 擦除 Flash 扇区数据，进行数据的读出和写入操作，并执行用户程序：在 DebugAnalyser 工具上，打印 Flash ID 以及 FLASH 内部数据信息的读取和写入信息。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\SPI\SPI\_Flash。

### 8.4.1 硬件设计

硬件连接：P4\_0 -> CLK， P4\_1 -> DI， P4\_2 -> DO， P4\_3 -> CS#。

EVB 外接 FM25Q16 模块，连接 P4\_0 和 CLK，P4\_1 和 DI，P4\_2 和 DO，P4\_3 和 CS#，同时连接 VDD 和模块的 VDD、WP#、HOLD#，连接 GND 和模块的 VSS。

## 8.4.2 软件流程

引脚定义：

```
#define SPI0_SCK_PIN      P4_0
#define SPI0_MOSI_PIN      P4_1
#define SPI0_MISO_PIN      P4_2
#define SPI0_CS_PIN        P4_3
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出使能、输出高；

配置 PINMUX：分配引脚分别为 SPI0\_CLK\_MASTER、SPI0\_MO\_MASTER、SPI0\_MI\_MASTER、  
SPI0\_SS\_N\_0\_MASTER 功能（详情参考：SPI -> Polling\_fullduplex）。

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_spi\_init 函数，对 SPI0 外设进行初始化：

- SPI 的数据传输模式为全双工模式；工作模式为主设备模式；数据大小为 8 位帧结构；串行时钟的稳态为时钟悬空高；位捕获的时钟活动沿为数据捕获于第二个时钟沿；时钟分频系数为 100；发送 FIFO 的阈值为 2；读取的数据长度阈值为 1；数据传输格式为 Motorola 传输格式（详情参考：SPI -> Polling\_fullduplex）；

使能 SPI0 外设。

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 spi\_demo：

- 通过 SPI0 读取 Flash\_ID 信息并打印 ID 信息；
- 通过 SPI0 擦除 Flash 扇区；
- 通过 SPI0 读取 Flash 内部扇区内数据信息并打印数据信息；
- 通过 SPI0 将 write\_data 数据写入 Flash 扇区；
- 分别打印写入数据前后 Flash 内部数据信息。

```
void spi_demo(void)
{
    .....
    for (uint16_t i = 0; i < 100; i++)
    {
        write_data[i] = i & 0xFF;
    }
}
```

```
spi_flash_read_id(flash_id);
APP_PRINT_INFO1("[io_spi] spi_demo: flash_id = %b ", TRACE_BINARY(3, flash_id));

spi_flash_sector_erase(0x001000);
APP_PRINT_INFO0("[io_spi] spi_demo: spi_flash_sector_erase done");

spi_flash_read(SPI_FLASH_FAST_READ, 0x001000, read_data, 100);
APP_PRINT_INFO1("[io_spi] spi_demo: read_data = %b ", TRACE_BINARY(100, read_data));

spi_flash_page_write(0x001000, write_data, 100);

spi_flash_read(SPI_FLASH_FAST_READ, 0x001000, read_data, 100);
APP_PRINT_INFO1("[io_spi] spi_demo: read_data = %b ", TRACE_BINARY(100, read_data));
}
```

在 DebugAnalyser 工具上，打印 Flash ID 以及 Flash 内部数据的读取以及写入信息。

## 8.5 GDMA 方式搬运数据

使用 SPI0 与 FM25Q16 以 GDMA 搬运数据方式进行数据传输。通过 GDMA 搬运 Flash 内部数据信息，再通过 UART 发送数据到 PC 端，并执行用户程序：在 DebugAnalyser 工具上，打印读取 FLASH 内部数据信息。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ SPI\ GDMA。

### 8.5.1 硬件设计

硬件连接： P3\_0 -> RX， P3\_1 -> TX；  
P4\_0 -> CLK， P4\_1 -> DI， P4\_2 -> DO， P4\_3 -> CS#。

EVB 外接 FT232 模块，连接 P3\_0 和 FT232 的 RX， P3\_1 和 FT232 的 TX；

EVB 外接 FM25Q16 模块，连接 P4\_0 和 CLK， P4\_1 和 DI， P4\_2 和 DO， P4\_3 和 CS#，同时连接 VDD 和模块的 VDD、WP#、HOLD#，连接 GND 和模块的 VSS。

### 8.5.2 软件流程

引脚定义：

```
#define UART_TX_PIN          P3_0
#define UART_RX_PIN          P3_1
```

开启 UART\_GDMA 功能。

```
#define UART_GDMA_TX_ENABLE           EVB_ENABLE  
#define UART_GDMA_RX_ENABLE           EVB_ENABLE
```

配置 PAD: 设置引脚、PINMUX 模式、PowerOn、内部上拉、输出失能、输出高;

配置 PINMUX: 分配引脚分别为 UART0\_TX、UART0\_RX 功能(详情参考: UART->Polling)。

引脚定义:

```
#define SPI0_SCK_PIN      P4_0  
#define SPI0_MOSI_PIN     P4_1  
#define SPI0_MISO_PIN     P4_2  
#define SPI0_CS_PIN       P4_3
```

配置 PAD: 设置引脚、PINMUX 模式、PowerOn、内部上拉、输出使能、输出高;

配置 PINMUX: 分配引脚分别为 SPI0\_CLK\_MASTER、SPI0\_MO\_MASTER、SPI0\_MI\_MASTER、  
SPI0\_SS\_N\_0\_MASTER 功能 (详情参考: SPI -> EEPROM)。

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,  
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数, 执行 driver\_UART\_init 函数, 对 UART0 外设进行初始化:

- 默认波特率 115200, 无奇偶校验, 停止位 1 位, 数据长度 8bits, FIFO 阈值为 16, 空闲时间为 2byte 等 (详情参考: UART->Polling)。

```
void driver_uart_init(void)  
{  
    .....  
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时, 执行 app\_handle\_dev\_state\_evt 函数, 通过 UART0 发送字符串"IO-SPI+GDMA read flash data demo!"到 PC 端, 并执行 spi\_demo 函数, 执行 driver\_spi\_gdma\_init, 初始化 SPI0 以及 GDMA 通道 2 外设:

初始化 SPI0 外设:

- SPI 的数据传输模式为 EEPROM 模式; 工作模式为主设备模式; 数据大小为 8 位帧结构; 串行时钟的稳态为时钟悬空高; 位捕获的时钟活动沿为数据捕获于第二个时钟沿; 时钟分频系数为 100; 数据传输格式为 Motorola 传输格式; 读取的数据长度阈值为 GDMA\_READ\_SIZE\_MAX; 开启接收数据 GDMA 通道; SPI\_RxWaterlevel 设置为 1 (详情参考: SPI -> Polling\_eeprom);

初始化 GDMA 外设:

- 使用 GDMA 通道 2;
- 传输方向为外设到内存传输, 源端地址为 FLASH\_SPI->DR; 目的端地址为 GDMA\_Recv\_Buffer;

- 使能 Multi-block 传输;
  - 设置每次 block 传输后自动加载 LLI 结构体中源地址与目的地址值; 使能 Multi-block 传输; 设置传输 LLI 类型结构体首地址;
  - 配置每次 block 传输后 LLI 结构体中源地址、目的地址、链表指针、控制寄存器 (详情参考: GDMA->Mem2Mem\_multi\_block);
- 使能 GDMA 通道 2 GDMA\_INT\_Block 中断; 使能 GDMA 传输; 使能 SPI0 外设; 通过 SPI0 外设发送读取数据命令以及地址。

```
void driver_spi_init(void)
{
    .....
    /*-----SPI init-----*/
    SPI_InitTypeDef SPI_InitStruct;
    SPI_StructInit(&SPI_InitStruct);
    .....
    SPI_InitStruct.SPI_NDF          = GDMA_READ_SIZE_MAX - 1;
    SPI_InitStructure.SPI_RxDmaEn   = ENABLE;
    SPI_InitStructure.SPI_RxWaterlevel = 1;
    .....
    /*-----SPI init-----*/
    GDMA_StructInit(&GDMA_InitStruct);
    GDMA_InitStruct.GDMA_ChannelNum      = GDMA_CHANNEL_NUM;
    GDMA_InitStruct.GDMA_DIR           = GDMA_DIR_PeripheralToMemory;
    .....
    GDMA_InitStruct.GDMA_SourceAddr     = (uint32_t)FLASH_SPI->DR;
    GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t)GDMA_Recv_Buffer;

    GDMA_InitStruct.GDMA_Multi_Block_Mode   = LLI_TRANSFER;
    GDMA_InitStruct.GDMA_Multi_Block_En     = 1;
    GDMA_InitStruct.GDMA_Multi_Block_Struct = (uint32_t)GDMA_LLIStruct;
    for (int i = 0; i < GDMA_MULTIBLOCK_SIZE; i++)
    {
        .....
    }
    .....
}
```

```

/* Enable transfer interrupt */
GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Block, ENABLE);
GDMA_Cmd(GDMA_CHANNEL_NUM, ENABLE);
SPI_Cmd(FLASH_SPI, ENABLE);

SPI_SendBuffer(FLASH_SPI, GDMA_WriteCmdBuffer, 5);
}

```

当一次 block 传输完成时, 触发 GDMA\_INT\_Block 中断, 进入中断处理函数 GDMA\_Channel\_Handler:

- 失能 GDMA 通道 2 GDMA\_INT\_Block 中断;
- 定义消息类型 IO\_MSG\_TYPE\_GDMA, 保存数据, 发送 msg 给 task;
- 清除 GDMA 中断悬挂位, 使能 GDMA\_INT\_Block 中断。

```

void GDMA_Channel_Handler(void)
{
    GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Block, DISABLE);

    T_IO_MSG int_gdma_msg;
    int_gdma_msg.type = IO_MSG_TYPE_GDMA;
    int_gdma_msg.subtype = 0;
    int_gdma_msg.u.buf = (void *)GDMA_Recv_Buffer;
    if (false == app_send_msg_to_apptask(&int_gdma_msg))
        .....
    GDMA_ClearINTPendingBit(GDMA_CHANNEL_NUM, GDMA_INT_Block);
    GDMA_INTConfig(GDMA_CHANNEL_NUM, GDMA_INT_Block, ENABLE);
}

```

app\_main\_task 循环检测 msg queue。当有 msg 时, 执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中, 判断消息类型为 IO\_MSG\_TYPE\_GDMA, 执行 io\_handle\_gdma\_msg:

- 打印 block 传输次数, 数据长度和传输数据完成信息;
- 通过 UART0 发送 GDMA 搬运的数据信息到 PC 端。

```

void io_handle_gdma_msg(T_IO_MSG *io_gdma_msg)
{
    .....
    APP_PRINT_INFO2("[io_gdma] io_handle_gdma_msg: read data complete, num = %d, data_len = %d",
                   gdma_transfer_num, data_len);
    uart_senddata_continuous(UART0, &p_buff[gdma_transfer_num * data_len], data_len);
    gdma_transfer_num++;
}

```

{}

在 DebugAnalyser 工具上，打印 GDMA 搬运数据信息，并在串口调试助手收到 S-Bee2 搬运的数据。

## 8.6 伪静态随机存储（PSRAM）

使用 SPI0 与 APS6404L 模块以 GDMA 方式进行数据传输。

通过 GDMA 搬运内存数据信息到 PARAM 芯片中，通过 GDMA 传输完成中断向 app\_task 发送消息事件，app\_task 检测到消息事件，在 app 层解析消息，并执行用户程序：在 DebugAnalyser 工具上，打印发送数据完成信息；

通过 GDMA 搬运 PARAM 芯片内部数据信息到内存中，通过 GDMA 传输完成中断向 app\_task 发送消息事件，app\_task 检测到消息事件，在 app 层解析消息，并执行用户程序：在 DebugAnalyser 工具上，打印接收数据完成信息，并通过 UART 外设发送接收数据到 PC 端。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ SPI\ PSRAM。

### 8.6.1 硬件设计

硬件连接：P3\_0 -> RX， P3\_1 -> TX。

EVB 外接 FT232 模块，连接 P3\_0 和 FT232 的 RX， P3\_1 和 FT232 的 TX。

硬件连接：P4\_0 -> CLK， P4\_1 -> DI， P4\_2 -> DO， P4\_3 -> CS#。

EVB 外接 FM25Q16 模块，连接 P4\_0 和 CLK， P4\_1 和 DI， P4\_2 和 DO， P4\_3 和 CS#，同时连接 VDD 和模块的 VDD、WP#、HOLD#，连接 GND 和模块的 VSS。

### 8.6.2 软件流程

引脚定义：

```
#define UART_TX_PIN          P3_0  
#define UART_RX_PIN          P3_1
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出失能、输出高；

配置 PINMUX：分配引脚分别为 UART0\_TX、UART0\_RX 功能(详情参考：UART->Polling)。

引脚定义：

```
#define SPI0_SCK_PIN        P4_0  
#define SPI0_MOSI_PIN       P4_1  
#define SPI0_MISO_PIN       P4_2  
#define SPI0_CS_PIN         P4_3
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出使能、输出高；

配置 PINMUX：分配引脚分别为 SPI0\_CLK\_MASTER、SPI0\_MO\_MASTER、SPI0\_MI\_MASTER、SPI0\_SS\_N\_0\_MASTER 功能（详情参考：SPI -> EEPROM）。

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,  
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_UART\_init 函数，对 UART0 外设进行初始化：

- 默认波特率 115200，无奇偶校验，停止位 1 位，数据长度 8bits，FIFO 阈值为 16，空闲时间为 2byte 等（详情参考：UART->Polling）。

```
void driver_uart_init(void)  
{  
    .....  
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 spi\_demo 函数，执行 driver\_spi\_gdma\_tx\_init，初始化 SPI0 以及 GDMA 通道 0 外设：

初始化 SPI0 外设：

- SPI 的数据传输模式为发送数据模式；工作模式为主设备模式；数据大小为 8 位帧结构；串行时钟的稳态为时钟悬空高；位捕获的时钟活动沿为数据捕获于第二个时钟沿；时钟分频系数为 100；数据传输格式为 Motorola 传输格式；读取数据长度阈值为 1；SPI\_TxDmaEn 设置为 ENABLE；

初始化 GDMA 外设：

- 使用 GDMA 通道 0；
- 传输方向为内存到外设传输，源端地址为 GDMA\_Send\_Buffer；目的端地址为 SPI0->DR；
- 使能 GDMA 通道 0 总传输完成中断 (GDMA\_INT\_Transfer)；
- 使能 SPI0 外设传输；
- 使能 GDMA 通道 0 传输。

```
void driver_spi_gdma_tx_init (void)  
{  
    .....  
    /*-----SPI init-----*/  
    SPI_InitTypeDef SPI_InitStruct;  
    SPI_StructInit(&SPI_InitStruct);  
    SPI_InitStructure.SPI_Direction = SPI_Direction_TxOnly;  
    .....  
    SPI_InitStructure.SPI_NDF = 0;  
    SPI_InitStructure.SPI_TxDmaEn = ENABLE;  
    SPI_InitStructure.SPI_TxWaterlevel = 15;
```

```

/*-----GDMA initial-----*/
GDMA_StructInit(&GDMA_InitStruct);
.....
GDMA_InitStruct.GDMA_ChannelNum      = GDMA_SPI_TX_NUM;
GDMA_InitStruct.GDMA_DIR            = GDMA_DIR_MemoryToPeripheral;
.....
GDMA_InitStruct.GDMA_SourceAddr     = (uint32_t)GDMA_Send_Buffer;
GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t)SPI0->DR;
.....
/* Enable transfer interrupt */
GDMA_INTConfig(GDMA_SPI_TX_NUM, GDMA_INT_Transfer, ENABLE);
SPI_Cmd(SPI0, ENABLE);
GDMA_Cmd(GDMA_SPI_TX_NUM, ENABLE);
}

```

当 GDMA 通道 0 搬运数据完成时，触发 GDMA\_INT\_Transfer 中断，进入 GDMA 中断处理函数 GDMA\_SPI\_TX\_Handler：

- 失能 GDMA 通道 0 GDMA\_INT\_Transfer 中断；
- 定义消息类型 IO\_MSG\_TYPE\_GDMA，子类型为 0，保存 GDMA\_Send\_Buffer 数据，发送 msg 给 task；
- 清除 GDMA 通道 0 GDMA\_INT\_Transfer 中断挂起位；
- 失能 GDMA 通道 0 传输。

```

GDMA_SPI_TX_Handler()
{
    GDMA_INTConfig(GDMA_SPI_TX_NUM, GDMA_INT_Transfer, DISABLE);
    T_IO_MSG int_gdma_msg;
    int_gdma_msg.type = IO_MSG_TYPE_GDMA;
    int_gdma_msg.subtype = 0;
    int_gdma_msg.u.buf = (void *)GDMA_Send_Buffer;
    if (false == app_send_msg_to_apptask(&int_gdma_msg))
    .....
    GDMA_ClearINTPendingBit(GDMA_SPI_TX_NUM, GDMA_INT_Transfer);
    GDMA_Cmd(GDMA_SPI_TX_NUM, DISABLE);
}

```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg) 函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_GDMA，执行 io\_handle\_gdma\_msg 函数。

数：

- 判断子类型若为 0，打印发送数据完成信息；执行函数 driver\_spi\_gdma\_rx\_init()。

```
void io_handle_gdma_msg(T_IO_MSG *io_gdma_msg)
{
    .....
    if (sub_type == 0)
    {
        APP_PRINT_INFO0("[io_gdma] io_handle_gdma_msg: send data complete");
        driver_spi_gdma_rx_init();
    }
    .....
}
```

在 driver\_spi\_gdma\_rx\_init 中，初始化 SPI0 以及 GDMA 通道 2 外设：

初始化 SPI0 外设：

- SPI 的数据传输模式为 EEPROM；工作模式为主设备模式；数据大小为 8 位帧结构；串行时钟的稳态为时钟悬空高；位捕获的时钟活动沿为数据捕获于第二个时钟沿；时钟分频系数为 100；数据传输格式为 Motorola 传输格式；读取数据长度阈值为 1000；SPI\_RxDmaEn 设置为 ENABLE；

初始化 GDMA 外设：

- 使用 GDMA 通道 2；
- 传输方向为内存到外设传输，源端地址为 SPI0->DR；目的端地址为 GDMA\_Recv\_Buffer；
- 使能 GDMA 通道 2 总传输完成中断 (GDMA\_INT\_Transfer)；
- 使能 SPI0 外设传输；
- 使能 GDMA 通道 2 传输。

```
void driver_spi_gdma_tx_init (void)
{
    .....
    /*-----SPI init-----*/
    SPI_InitTypeDef SPI_InitStruct;
    SPI_StructInit(&SPI_InitStruct);
    SPI_InitStructure.SPI_Direction = SPI_Direction_EEPROM;
    .....
    SPI_InitStructure.SPI_NDF = 0;
    SPI_InitStructure.SPI_TxDmaEn = ENABLE;
    SPI_InitStructure.SPI_TxWaterlevel = 15;
```

```

/*-----GDMA initial-----*/
GDMA_StructInit(&GDMA_InitStruct);
.....
GDMA_InitStruct.GDMA_ChannelNum      = GDMA_SPI_RX_NUM;
GDMA_InitStruct.GDMA_DIR            = GDMA_DIR_PeripheralToMemory;
.....
GDMA_InitStruct.GDMA_SourceAddr     = (uint32_t)SPI0->DR;
GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t)GDMA_Recv_Buffer;
.....
/* Enable transfer interrupt */
GDMA_INTConfig(GDMA_SPI_RX_NUM, GDMA_INT_Transfer, ENABLE);
SPI_Cmd(SPI0, ENABLE);
GDMA_Cmd(GDMA_SPI_RX_NUM, ENABLE);
}

```

当 GDMA 通道 2 搬运数据完成时，触发 GDMA\_INT\_Transfer 中断，进入 GDMA 中断处理函数 GDMA\_SPI\_RX\_Handler：

- 失能 GDMA 通道 2 GDMA\_INT\_Transfer 中断；
- 定义消息类型 IO\_MSG\_TYPE\_GDMA，子类型为 1，保存 GDMA\_Recv\_Buffer 数据，发送 msg 给 task；
- 清除 GDMA 通道 2GDMA\_INT\_Transfer 中断挂起位。

```

GDMA_SPI_RX_Handler()
{
    GDMA_INTConfig(GDMA_SPI_RX_NUM, GDMA_INT_Transfer, DISABLE);

    T_IO_MSG int_gdma_msg;
    int_gdma_msg.type = IO_MSG_TYPE_GDMA;
    int_gdma_msg.subtype = 1;
    int_gdma_msg.u.buf = (void *)GDMA_Recv_Buffer;
    if (false == app_send_msg_to_apptask(&int_gdma_msg))
        .....
    GDMA_ClearINTPendingBit(GDMA_SPI_RX_NUM, GDMA_INT_Transfer);
}

```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_GDMA，执行 io\_handle\_gdma\_msg 函数：

- 判断子类型若为 1，打印读取数据完成信息；
- 执行函数 `uart_senddata_continuous(UART0, p_buf, GDMA_TRANSFER_SIZE)`，将读取的数据通过 UART0 发送到 PC 端。

```
void io_handle_gdma_msg(T_IO_MSG *io_gdma_msg)
{
    .....
    if(sub_type == 1)
    {
        APP_PRINT_INFO1("[io_gdma] io_handle_gdma_msg: read data complete data_len = %d",
                      GDMA_TRANSFER_SIZE);
        uart_senddata_continuous(UART0, p_buf, GDMA_TRANSFER_SIZE);
    .....
}
```

当 GDMA 搬运发送数据完成时，在 DebugAnalyser 工具上，打印发送数据完成信息；

当 GDMA 搬运接收数据完成时，在 DebugAnalyser 工具上，打印接收数据完成信息，并在串口调试助手收到 S-Bee2 接收的数据。

## 9 内部集成电路(I2C)

专用硬件设备：STK8321 模块

### 9.1 I2C 通信

使用 I2C0 与 STK8321 模块进行数据传输。获取 STK8321 ID、三轴输出数据，通过 I2C0 中断向 app\_task 发送消息事件，app\_task 检测到消息事件，在 app 层解析消息，执行用户程序：在 DebugAnalyser 工具上，打印 GSensor 三轴输出数据。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\I2C\STK8321。

#### 9.1.1 硬件设计

硬件连接：P4\_0 -> SCL，P4\_1 -> SDA。

EVB 外接 STK8321 模块，连接 P4\_0 和 SCL，P4\_1 和 SDA，连接 GND 和 VDD，VCC 和 CS、VCC。

#### 9.1.2 软件流程

引脚定义：

```
#define I2C0_SCL_PIN      P4_0
#define I2C0_SDA_PIN      P4_1
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出使能、输出高；

配置 PINMUX：分配引脚分别为 I2C0\_CLK、I2C0\_DAT 功能。

```
void board_i2c_init(void)
{
    Pad_Config(I2C0_SCL_PIN,      PAD_PINMUX_MODE,      PAD_IS_PWRON,      PAD_PULL_UP,
PAD_OUT_ENABLE, PAD_OUT_HIGH);

    Pad_Config(I2C0_SDA_PIN,      PAD_PINMUX_MODE,      PAD_IS_PWRON,      PAD_PULL_UP,
PAD_OUT_ENABLE, PAD_OUT_HIGH);

    Pinmux_Config(I2C0_SCL_PIN, I2C0_CLK);
    Pinmux_Config(I2C0_SDA_PIN, I2C0_DAT);
}
```

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_i2c\_init 函数，对 I2C0 外设进行初始化：

- I2C\_ClockSpeed 设置为 100000;
  - I2C\_DeviveMode 设置为 I2C\_DeviveMode\_Master，即主设备；
  - I2C\_AddressMode 设置为 I2C\_AddressMode\_7BIT，即 7 位地址模式；
  - I2C\_SlaveAddress 设置为 STK8321\_ADDRESS (0x0F)，即从机地址为 0x0F；
  - I2C\_Ack 设置为 I2C\_Ack\_Enable，即使能 Ack 功能；
- 使能 I2C0；

```
void driver_i2c_init(void)
{
    .....
    I2C_InitStruct.I2C_ClockSpeed      = 100000;
    I2C_InitStruct.I2C_DeviveMode     = I2C_DeviveMode_Master;
    I2C_InitStruct.I2C_AddressMode   = I2C_AddressMode_7BIT;
    I2C_InitStruct.I2C_SlaveAddress  = STK8321_ADDRESS;
    I2C_InitStruct.I2C_Ack          = I2C_Ack_Enable;
    .....
    I2C_Cmd(I2C0, ENABLE);
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt，执行 i2c\_demo，判断 ID 是否为 0x23，正确则：

- 使能 I2C0 检测到停止信号中断(I2C\_INT\_STOP\_DET)，清除 I2C\_INT\_STOP\_DET 中断挂起位；
- 获取 stk8321 三轴输出数据。

```
void i2c_demo(void)
{
    if(0x23 == stk8321_id_get())
    {
        APP_PRINT_INFO0("[io_i2c] i2c_demo: Get stk8321 chip id OK. ");
        nvic_i2c_config();
        stk8321_outdata_get();
    }
}
```

当检测到停止信号时，触发 I2C\_INT\_STOP\_DET 中断，进入中断处理函数 I2C0\_Handler：判断检测到停止信号状态为 SET 时：

- 定义消息类型 IO\_MSG\_TYPE\_I2C，保存 GSensor\_Data.OutData 数据，发送 msg 给 task；
- 清除 I2C\_INT\_STOP\_DET 中断挂起位；

```
void I2C0_Handler(void)
{
    if (I2C_GetINTStatus(I2C0, I2C_INT_STOP_DET) == SET)
    {
        T_IO_MSG int_i2c_msg;
        int_i2c_msg.type = IO_MSG_TYPE_I2C;
        int_i2c_msg.subtype = 0;
        int_i2c_msg.u.buf = (void *)(&GSensor_Data.OutData);
        if (false == app_send_msg_to_apptask(&int_i2c_msg))
        .....
        I2C_ClearINTPendingBit(I2C0, I2C_INT_STOP_DET);
    }
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_I2C，执行 io\_handle\_i2c\_msg 函数，执行 io\_i2c\_handle\_msg：打印三轴输出数据信息。

```
void io_i2c_handle_msg(T_IO_MSG *io_i2c_msg)
{
    uint8_t *p_buf = io_i2c_msg->u.buf;
    GSensor_Data.XData = p_buf[0] & 0xFF;
    GSensor_Data.XData = (GSensor_Data.XData + ((uint16_t)p_buf[1] << 8)) >> 4;
    GSensor_Data.YData = p_buf[2] & 0xFF;
    GSensor_Data.YData = (GSensor_Data.YData + ((uint16_t)p_buf[3] << 8)) >> 4;
    GSensor_Data.ZData = p_buf[4] & 0xFF;
    GSensor_Data.ZData = (GSensor_Data.ZData + ((uint16_t)p_buf[5] << 8)) >> 4;
    APP_PRINT_INFO3("[io_i2c]io_i2c_handle_msg: GSensor_x_axic_data = %d, y_axic_data = %d,
z_axic_data = %d", GSensor_Data.XData, GSensor_Data.YData, GSensor_Data.ZData);
}
```

在 DebugAnalyser 工具上，打印 GSensor 三轴输出数据。

# 10 按键扫描(KEYSCAN)

专用硬件设备：矩阵键盘

## 10.1 按键扫描

Keypress采用手动扫描模式，即扫描一次就结束。实现矩阵键盘扫描，通过按键中断向app\_task发送按键消息事件，app\_task检测到消息事件，在app层解析按键消息，并执行用户程序：在DebugAnalyser工具上，打印按键信息。

Key press debounce 及 key release debounce 使用软件定时器实现，本工程中 Key press debounce 根据按键的机械特性设置为 200ms，key release debounce 设置为 10ms。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\KEYSCAN\Keypress。

### 10.1.1 硬件设计

硬件连接：P2\_3 -> ROW0, P2\_4 -> ROW1, P4\_0 -> COLUMN0, P4\_1 -> COLUMN1。

EVB 外接矩阵键盘模块，连接 P2\_3 和 ROW0, P2\_4 和 ROW1, P4\_0 和 COLUMN0, P4\_1 和 COLUMN1，外接矩阵键盘如图 10.1 所示。

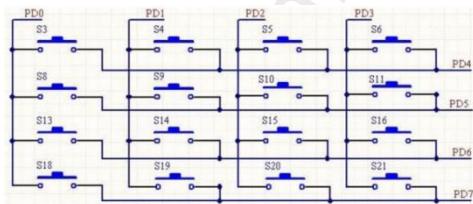


图 10-1 外接矩阵键盘

### 10.1.2 软件流程

引脚定义：定义  $2 \times 2$  矩阵键盘。

#define KEYBOARD_ROW_SIZE	2
#define KEYBOARD_COLUMN_SIZE	2
#define KEYBOARD_ROW_0	P2_3
#define KEYBOARD_ROW_1	P2_4
#define KEYBOARD_COLUMN_0	P4_0
#define KEYBOARD_COLUMN_1	P4_1

初始化 keyscale 全局数据。

```
void global_data_keyscale_init(void)
{
}
```

```
    memset(&Current_Key_Data, 0, sizeof(KeyScan_Data_TypeDef));  
}
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉或无上拉、输出使能或失能、输出低；

配置 PINMUX：分配引脚分别为 KEY\_ROW\_0、KEY\_ROW\_1、KEY\_COL\_0、KEY\_COL\_1 功能。

```
void board_keyboard_init(void)
```

```
{
```

```
    Pad_Config(KEYBOARD_ROW_0, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_UP,  
    PAD_OUT_DISABLE, PAD_OUT_LOW);
```

```
    Pad_Config(KEYBOARD_ROW_1, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_UP,  
    PAD_OUT_DISABLE, PAD_OUT_LOW);
```

```
    Pad_Config(KEYBOARD_COLUMN_0, PAD_PINMUX_MODE, PAD_IS_PWRON,  
    PAD_PULL_NONE, PAD_OUT_ENABLE, PAD_OUT_LOW);
```

```
    Pad_Config(KEYBOARD_COLUMN_1, PAD_PINMUX_MODE, PAD_IS_PWRON,  
    PAD_PULL_NONE, PAD_OUT_ENABLE, PAD_OUT_LOW);
```

```
    Pinmux_Config(KEYBOARD_ROW_0, KEY_ROW_0);
```

```
    Pinmux_Config(KEYBOARD_ROW_1, KEY_ROW_1);
```

```
    Pinmux_Config(KEYBOARD_COLUMN_0, KEY_COL_0);
```

```
    Pinmux_Config(KEYBOARD_COLUMN_1, KEY_COL_1);
```

```
}
```

初始化 Keyscan 软件定时器，定时时间 200ms，回调函数为 timer\_keysan\_callback。

```
os_timer_create(&KeyScan_Timer_Handle, "keyscan_timer", 1, KEYS defense_SCAN_SW_INTERVAL, false,  
timer_keysan_callback));
```

app\_task 初始化。

```
os_task_create(&app_task_handle, "app", app_main_task, 0, APP_TASK_STACK_SIZE,  
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_keyboard\_init 函数，对 KEYS defense SCAN 外设进行初始化：

- rowSize 设置为 KEYBOARD\_ROW\_SIZE， colSize 设置为 KEYBOARD\_COLUMN\_SIZE，即  $2 \times 2$  矩阵键盘；
- scanmode 设置为 KeyScan\_Manual\_Scan\_Mode，即手动扫描模式；
- debounceEn 设置为 vDebounce\_En，即开启 keysan 的硬件去抖动功能；
- 使能 KEYS defense SCAN；使能 KEYS defense SCAN 单次扫描结束中断 (KEYSCAN\_INT\_SCAN\_END)；清除 KEYS defense SCAN-END 中断挂起位；取消屏蔽 KEYS defense SCAN-END 中断；

```
void driver_keyboard_init(uint32_t vDebounce_En)
```

```
{
```

```
.....
KEYSCAN_InitStruct.rowSize = KEYBOARD_ROW_SIZE;
KEYSCAN_InitStruct.colSize = KEYBOARD_COLUMN_SIZE;
KEYSCAN_InitStruct.scanmode = KeyScan_Manual_Scan_Mode;
KEYSCAN_InitStruct.debounceEn = vDebounce_En;
.....
KeyScan_Cmd(KEYSCAN, ENABLE);
KeyScan_INTConfig(KEYSCAN, KEYSCAN_INT_SCAN_END, ENABLE);
KeyScan_ClearINTPendingBit(KEYSCAN, KEYSCAN_INT_SCAN_END);
KeyScan_INTMask(KEYSCAN, KEYSCAN_INT_SCAN_END, DISABLE);
.....
}
```

开始任务调度。

```
os_sched_start();
```

Keyscan 扫描键盘，键盘扫描结束触发中断，进入中断处理函数 Keyscan\_Handler：

- 屏蔽 KEYSCAN\_INT\_SCAN\_END 中断；
- 如果 Keyscan 的 FIFO 不为空（即有按键按下），则进行如下操作：
  - ◆ 从 KeyScan FIFO 中读取数据长度和数据，即按下按键的个数和键值，置位 Key\_Pressed\_Flag；
  - ◆ 启动软件定时器，OS 实现定时 200ms，Press debounce 时间，时间到进入 timer\_keysan\_callback 回调函数；
  - ◆ 定义消息类型 IO\_MSG\_TYPE\_KEYSCAN，子类型 IO\_MSG\_KEYSCAN\_RX\_PKT，保存数据，发送 msg 给 task；
- 清除 KEYSCAN\_INT\_SCAN\_END 中断挂起位，取消屏蔽 KEYSCAN\_INT\_SCAN\_END 中断。

```
void Keyscan_Handler(void)
{
.....
if (KeyScan_GetFlagState(KEYSCAN, KEYSCAN_INT_FLAG_SCAN_END) == SET)
{
    KeyScan_INTMask(KEYSCAN, KEYSCAN_INT_SCAN_END, ENABLE);
    memset(&Current_Key_Data, 0, sizeof(KeyScan_Data_TypeDef));
    if (KeyScan_GetFlagState(KEYSCAN, KEYSCAN_FLAG_EMPTY) != SET)
    {
        fifo_length = (uint32_t)KeyScan_GetFifoDataNum(KEYSCAN);
        KeyScan_Read(KEYSCAN, (uint16_t *)&Current_Key_Data.key[0], fifo_length);
        Current_Key_Data.length = fifo_length;
    }
}
```

```
Key_Pressed_Flag = true;

if (!os_timer_restart(&KeyScan_Timer_Handle, KEYSCAN_SW_INTERVAL))
    .....
    int_keyscan_msg.type = IO_MSG_TYPE_KEYSCAN;
    int_keyscan_msg.subtype = IO_MSG_KEYSCAN_RX_PKT;
    int_keyscan_msg.u.buf = (void *)&Current_Key_Data;
    if (false == app_send_msg_to_apptask(&int_keyscan_msg))
        .....
}
KeyScan_ClearINTPendingBit(KEYSCAN, KEYSCAN_INT_SCAN_END);
KeyScan_INTMask(KEYSCAN, KEYSCAN_INT_SCAN_END, DISABLE);
}
}
```

在 timer\_keyscan\_callback 回调函数中，判断 Key\_Pressed\_Flag：

- 标志位为 true 时：
  - ❖ 初始化 KEYSCAN 驱动，使能 KEYSCAN 功能，重新启动键盘扫描；
  - ❖ 同时启动软件定时器，定时 10ms，Release debounce 时间；
- 标志位为 false 时：
  - ❖ 定义消息类型 IO\_MSG\_TYPE\_KEYSCAN，子类型 IO\_MSG\_KEYSCAN\_ALLKEYRELEASE，并发送 msg 给 task；
  - ❖ 重新初始化 keyscan 全局数据，初始化 KEYSCAN 外设驱动。

```
void timer_keyscan_callback(void *p_xTimer)
{
    if (true == Key_Pressed_Flag)
    {
        Key_Pressed_Flag = false;
        driver_keyboard_init(KeyScan_Debounce_Disable);
        /* Start timer to check key status */
        os_timer_restart(&p_xTimer, KEYSCAN_SW_RELEASE_TIMEOUT);
    }
    else
    {
        T_IO_MSG int_keyscan_msg;
        int_keyscan_msg.type = IO_MSG_TYPE_KEYSCAN;
```

```
int _keyscan_msg.subtype = IO_MSG_KEYSCAN_ALLKEYRELEASE;
if (false == app_send_msg_to_apptask(&int _keyscan_msg))
.....
global_data_keyscan_init();
driver_keyboard_init(KeyScan_Debounce_Enable);
}
```

```
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_KEYSCAN 时，执行 io\_handle\_keyscan\_msg 函数，执行 io\_keyscan\_handle\_keys，判断子类型：

- 子类型为 IO\_MSG\_KEYSCAN\_RX\_PKT 时，获取消息中的按键数据，打印相应键值信息等；
- 子类型为 IO\_MSG\_KEYSCAN\_ALLKEYRELEASE 时，打印 All key release 信息。

```
static void io_keyscan_handle_keys(T_IO_MSG *io_keyscan_msg)
{
    uint16_t subtype = io_keyscan_msg->subtype;
    if (subtype == IO_MSG_KEYSCAN_RX_PKT)
    {
        KeyScan_Data_TypeDef *p_key_data = (KeyScan_Data_TypeDef *)io_keyscan_msg->u.buf;
        .....
    }
    else if (subtype == IO_MSG_KEYSCAN_ALLKEYRELEASE)
    {
        APP_PRINT_INFO0("[io_keyscan] io_keyscan_handle_keys: All keys release.");
    }
    .....
}
```

按下按键时，在 DebugAnalyser 工具上，打印按键信息。

# 11 红外(IR)

专用硬件设备：IR 收发模块、红外遥控编码分析仪、红外遥控器

专用测试软件：IRReader

## 11.1 红外发送

使用 IR 外设进行数据发送，实现 TX 功能。使用 IR 遥控编码分析仪接收 IR 发送的数据，并观察发送数据是否符合预期。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\IR\Tx。

### 11.1.1 硬件设计

硬件连接：P2\_5 -> IR 收发模块的发送端。

EVB 外接 IR 收发模块，连接 P2\_5 和 IR 收发模块的发送端，连接 VCC 和 IR 收发模块的 VCC，连接 GND 和 IR 收发模块的 GND。

### 11.1.2 软件流程

定义 IR 发送数据数组，并赋值：有载波数据用载波个数与 0x80000000 进行或运算表示，无载波数据用载波个数与 0x00000000 进行或运算表示，载波个数为：timing / CarrierFreq。

```
void ir_demo(void)
{
    IR_TxData.CarrierFreq = 38000;
    IR_TxData.DataLen = 67 + 1; //2+64+1;
    IR_TxData.DataBuf[0] = 0x80000000 | 0x156; //342 about 9ms
    IR_TxData.DataBuf[1] = 0x00000000 | 0xAB; //171 about 4.5ms
    for (uint16_t i = 2; i < IR_TxData.DataLen - 1;) {
        IR_TxData.DataBuf[i] = 0x80000000 | 0x15; //21 about 560us
        IR_TxData.DataBuf[i + 1] = 0x00000000 | 0x15; //21 about 560us
        i += 2;
    }
    IR_TxData.DataBuf[30] = 0x80000000 | 0x15; //21 about 560us
    IR_TxData.DataBuf[31] = 0x00000000 | 0x40; //64 about 1690us
    IR_TxData.DataBuf[62] = 0x80000000 | 0x15; //21 about 560us
```

```
IR_TxData.DataBuf[63] = 0x00000000 | 0x40; //64 about 1690us  
IR_TxData.DataBuf[64] = 0x80000000 | 0x15; //21 about 560us  
IR_TxData.DataBuf[65] = 0x00000000 | 0x40; //64 about 1690us  
IR_TxData.DataBuf[66] = 0x80000000 | 0x15; //21 about 560us  
IR_TxData.DataBuf[IR_TxData.DataLen - 1] = 0x00000000 | 0x15;  
.....  
}
```

引脚定义。

```
#define IR_TX_PIN P2_5
```

配置 PAD: 设置引脚、PINMUX 模式、PowerOn、无内部上拉、输出使能、输出低;

配置 PINMUX: 分配引脚为 IRDA\_TX 功能。

```
void board_ir_init(void)  
{  
    Pad_Config(IR_TX_PIN, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_NONE,  
    PAD_OUT_ENABLE, PAD_OUT_LOW);  
    Pinmux_Config(IR_TX_PIN, IRDA_TX);  
}
```

使能 IR 时钟;

初始化 IR 外设:

- IR\_Freq 设置为 vFreq, 即 IR 发送频率为 38KHz;
- IR\_DutyCycle 设置为 2, 即 IR 载波占空比为 1/2;
- IR\_Mode 设置为 IR\_MODE\_TX, 即 IR 模式为 IR 发送模式;
- IR\_TxInverse 设置为 IR\_TX\_DATA\_NORMAL, 即不反转 IR 发送数据;
- IR\_TxFIFOThrLevel 设置为 IR\_TX\_FIFO\_THR\_LEVEL, 即 IR 发送 FIFO 阈值为 2;

使能 IR 外设发送功能;

```
void driver_ir_init(uint32_t vFreq)  
{  
    RCC_PeriphClockCmd(APBPeriph_IR, APBPeriph_IR_CLOCK, ENABLE);  
    IR_InitTypeDef IR_InitStruct;  
    IR_StructInit(&IR_InitStruct);  
    IR_InitStruct.IR_Freq = vFreq;  
    IR_InitStruct.IR_DutyCycle = 2;  
    IR_InitStruct.IR_Mode = IR_MODE_TX;  
    IR_InitStruct.IR_TxInverse = IR_TX_DATA_NORMAL;  
    IR_InitStruct.IR_TxFIFOThrLevel = IR_TX_FIFO_THR_LEVEL;
```

```
IR_Init(&IR_InitStruct);
.....
IR_Cmd(IR_MODE_TX, ENABLE);
}
```

执行 IR\_SendBuf 函数，开始往 IR 发送 FIFO 中塞发送数据；

使能 IR 发送 FIFO 数据个数小于设置的发送阈值中断 (IR\_INT\_TF\_LEVEL)。

```
void ir_demo(void)
{
    .....
    IR_SendBuf(IR_TxData.DataBuf, IR_TX_FIFO_SIZE, DISABLE);
    IR_TX_Count = IR_TX_FIFO_SIZE;
    IR_INTConfig(IR_INT_TF_LEVEL, ENABLE);
}
```

当 IR 发送 FIFO 数据个数小于设置的发送阈值（此例中设为 2）时，触发 IR\_INT\_TF\_LEVEL 中断，进入中断处理函数 IR\_Handler：

- 屏蔽 IR\_INT\_TF\_LEVEL 中断；
- 判断 IR\_INT\_TF\_LEVEL 中断状态位是否为 SET，若为 SET（需往 IR 发送 FIFO 中塞数据）则：
  - ➔ 如果剩余发送数据个数大于发送 FIFO 大小，往 IR 发送 FIFO 中塞(IR\_TX\_FIFO\_SIZE - IR\_TX\_FIFO\_THR\_LEVEL)发送数据，清除 IR\_INT\_TF\_LEVEL 中断挂起位；
  - ➔ 否则如果剩余发送数据个数大于 0，往 IR 发送 FIFO 中塞剩余数据，清除 IR\_INT\_TF\_LEVEL 中断挂起位；
  - ➔ 否则失能 IR\_INT\_TF\_LEVEL 中断，清除 IR\_INT\_TF\_LEVEL 中断挂起位；
- 取消屏蔽 IR\_INT\_TF\_LEVEL 中断源。

```
void IR_Handler(void)
{
    IR_MaskINTConfig(IR_INT_TF_LEVEL, ENABLE);
    if (IR_GetINTStatus(IR_INT_TF_LEVEL) == SET)
    {
        if ((IR_TxData.DataLen - IR_TX_Count) >= IR_TX_FIFO_SIZE)
        {
            IR_SendBuf(IR_TxData.DataBuf+IR_TX_Count,(IR_TX_FIFO_SIZE-2),DISABLE);
            IR_TX_Count += (IR_TX_FIFO_SIZE - IR_TX_FIFO_THR_LEVEL);
            IR_ClearINTPendingBit(IR_INT_TF_LEVEL_CLR);
        }
        else if ((IR_TxData.DataLen - IR_TX_Count) > 0)
    }
```

```

{
    IR_SetTxThreshold(0);

    IR_SendBuf(IR_TxData.DataBuf+IR_TX_Count,IR_TxData.DataLen-IR_TX_Count,
DISABLE);

    IR_TX_Count += (IR_TxData.DataLen - IR_TX_Count);

    IR_ClearINTPendingBit(IR_INT_TF_LEVEL_CLR);

}

else

{
    IR_INTConfig(IR_INT_TF_LEVEL, DISABLE);

    IR_TX_Count = 0;

    IR_ClearINTPendingBit(IR_INT_TF_LEVEL_CLR);

}

}

IR_MaskINTConfig(IR_INT_TF_LEVEL, DISABLE);
}

```

使用 IR 遥控编码分析仪抓取 IR 发送的数据，并观察发送数据是否符合预期。

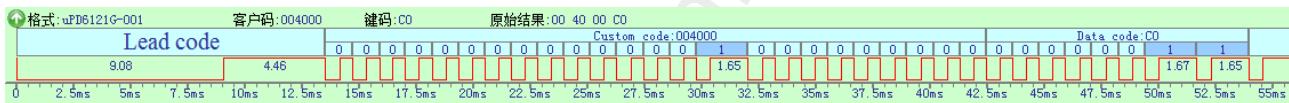


图 11-1 IR 发送数据编码

## 11.2 红外接收

使用 IR 外设进行数据接收，实现 RX 功能。使用 IR 遥控器向 IR 收发模块发送 IR 数据，通过 IR 中断接收 IR 数据并向 app\_task 发送消息事件，app\_task 检测到消息事件，在 app 层解析消息，执行用户程序：在 DebugAnalyser 工具上，打印接收到的 IR 数据信息。

工程目录：\HoneyComb\sdk\boardevb\_stack\_img\io\_sample\IR\Rx。

### 11.2.1 硬件设计

硬件连接：P2\_5 -> IR 收发模块的接收端。

EVB 外接 IR 收发模块，连接 P2\_5 和 IR 收发模块的接收端，连接 VCC 和 IR 收发模块的 VCC，连接 GND 和 IR 收发模块的 GND。

## 11.2.2 软件流程

初始化 IR 全局数据。

```
void global_data_ir_init(void)
{
    APP_PRINT_INFO0("[io_ir]global_data_ir_init");
    memset(&IR_Rx_Data, 0, sizeof(IR_Rx_Data));
    IR_RX_Count = 0;
}
```

配置 PAD、PINMUX，设置 P2\_5 为 IRDA\_RX 功能；

app\_task 初始化。

```
os_task_create(&app_task_handle, "app", app_main_task, 0, APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_ir\_init 函数，对 IR 外设进行初始化：

- 设置 IR 接收频率为 38KHz，IR 模式为 IR 接收模式，启动方式为 IR 自动接收模式，数据接收触发方式为下降沿触发；
  - 设置 IR 接收 FIFO 阈值为 30，当 IR 接收 FIFO 满时丢弃最新数据，过滤低于 50ns 的杂波数据；
  - 设置触发接收计数器中断的电平类型为高电平，IR\_RxCntThr 设置为 0x1F40；
- 使能 IR 外设接收功能，清除 IR 接收 FIFO；

```
void driver_ir_init(void)
{
    .....
    IR_InitStruct.IR_Freq          = 38000; /* IR carrier frequency is 38KHz */
    IR_InitStruct.IR_Mode          = IR_MODE_RX; /* IR receiveing mode */
    IR_InitStruct.IR_RxStartMode   = IR_RX_AUTO_MODE;
    IR_InitStruct.IR_RxFIFOThrLevel = IR_RX_FIFO_THR_LEVEL;
    IR_InitStruct.IR_RxFIFOFullCtrl = IR_RX_FIFO_FULL_DISCARD_NEWEST;
    IR_InitStruct.IR_RxFilterTime  = IR_RX_FILTER_TIME_50ns;
    IR_InitStruct.IR_RxTriggerMode = IR_RX_FALL_EDGE;
    IR_InitStruct.IR_RxCntThrType = IR_RX_Count_High_Level;
    IR_InitStruct.IR_RxCntThr     = 0x1F40;
    .....
    IR_Cmd(IR_MODE_RX, ENABLE);
    IR_ClearRxFIFO();
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 ir\_demo：

- 使能 IR 接收 FIFO 数据个数大于设置的接收阈值中断 (IR\_INT\_RF\_LEVEL) 和接收计数器达到设定阈值中断 (IR\_INT\_RX\_CNT\_THR);
- 取消屏蔽 IR\_INT\_RF\_LEVEL 和 IR\_INT\_RX\_CNT\_THR 中断。

```
void ir_demo(void)  
{  
    IR_INTConfig(IR_INT_RF_LEVEL | IR_INT_RX_CNT_THR, ENABLE);  
    IR_MaskINTConfig(IR_INT_RF_LEVEL | IR_INT_RX_CNT_THR, DISABLE);  
}
```

使用 IR 遥控器向 IR 收发模块发送 IR 数据。

当 IR 接收 FIFO 的数据个数达到设置的接收阈值（此例中设为 30）时，触发 IR\_INT\_RF\_LEVEL 中断，或当接收计数器的个数达到设置的阈值（此例中设为 0x1F40）时，触发 IR\_INT\_RX\_CNT\_THR 中断，进入中断处理函数 IR\_Handler：

- 屏蔽 IR\_INT\_RF\_LEVEL 和 IR\_INT\_RX\_CNT\_THR 中断；
- 判断 IR\_INT\_RF\_LEVEL 中断状态位是否为 SET，若为 SET（需从 IR 接收 FIFO 中取数据）则：
  - ➔ 从 IR 接收 FIFO 中取数据到 IR\_Rx\_Data 中；
  - ➔ 清除 IR\_INT\_RF\_LEVEL 中断挂起位；
- 判断 IR\_INT\_RX\_CNT\_THR 中断状态位是否为 SET，若为 SET（取剩余数据）则：
  - ➔ 从 IR 接收 FIFO 中取剩余数据到 IR\_Rx\_Data 中；
  - ➔ 定义消息类型 IO\_MSG\_TYPE\_IR，保存数据，发送 msg 给 task；
  - ➔ 清除 IR\_INT\_RX\_CNT\_THR 中断挂起位；
- 取消屏蔽 IR\_INT\_RF\_LEVEL 和 IR\_INT\_RX\_CNT\_THR 中断。

```
void IR_Handler(void)  
{  
    IR_MaskINTConfig(IR_INT_RF_LEVEL | IR_INT_RX_CNT_THR, ENABLE);  
    if (IR_GetINTStatus(IR_INT_RF_LEVEL) == SET)  
    {  
        len = IR_GetRxDataLen();  
        IR_ReceiveBuf(IR_Rx_Data.DataBuf + IR_RX_Count, len);  
        IR_Rx_Data.DataLen += len;  
        IR_RX_Count += len;  
        IR_ClearINTPendingBit(IR_INT_RF_LEVEL_CLR);  
    }  
}
```

```
if (IR_GetINTStatus(IR_INT_RX_CNT_THR) == SET)
{
    len = IR_GetRxDataLen();
    IR_ReceiveBuf(IR_Rx_Data.DataBuf + IR_RX_Count, len);
    IR_Rx_Data.DataLen += len;
    IR_RX_Count += len;

    T_IO_MSG int_ir_msg;
    int_ir_msg.type = IO_MSG_TYPE_IR;
    int_ir_msg.u.buf = (void *)(&IR_Rx_Data);
    if (false == app_send_msg_to_apptask(&int_ir_msg))
        .....
    IR_ClearINTPendingBit(IR_INT_RX_CNT_THR_CLR);
}

IR_MaskINTConfig(IR_INT_RF_LEVEL | IR_INT_RX_CNT_THR, DISABLE);
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中，当消息类型为 IO\_MSG\_TYPE\_IR 时，执行 io\_handle\_ir\_msg:

- 打印接收到的 IR 数据信息;

```
void io_handle_ir_msg(T_IO_MSG *io_ir_msg)
{
    IR_DataTypeDef *p_buf = io_ir_msg->u.buf;
    for (uint16_t i = 0; i < p_buf->DataLen; i++)
    {
        APP_PRINT_INFO2("[io_ir]io_handle_ir_msg: IR RX data[%d] = 0x%x", i, p_buf->DataBuf[i]);
    }
    .....
}
```

使用 IR 遥控器向 IR 收发模块发送 IR 数据。在 DebugAnalyser 工具上，打印接收到的 IR 数据信息。

## 11.3 红外编码协议发送

使用 IR 外设的 TX 功能和 IR 编码协议库实现 IR 编码协议发送。发送定义的 RAW 编码协议，发送完成，通过 IR 中断向 app\_task 发送消息事件，app\_task 检测到消息事件，在 app 层解析消息，执行用户程序：发送 IR 编码协议，并观察发送数据是否符合预期。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ IR\ SendCode。

### 11.3.1 硬件设计

硬件连接: P2\_5 -> IR 收发模块的发送端, P0\_1 -> LED0。

EVB 外接 IR 收发模块, 连接 P2\_5 和 IR 收发模块的发送端, 连接 VCC 和 IR 收发模块的 VCC, 连接 GND 和 IR 收发模块的 GND。

在 EVB 上, 使用跳帽短接 J24, 连接 P0\_1 和 LED0。

### 11.3.2 软件流程

配置 PAD、PINMUX, 设置 P0\_1 为 GPIO 输出功能; 初始化 GPIO 外设, 初始状态: P0\_1 置零。

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中, 执行 IR\_Send\_Init 函数:

- 执行 UIR\_TransInit: 初始化 IR 外设, 包括 PAD、PINMUX、DRIVER; 初始化 TIM 外设, 用于连续发送编码;
- 调用 UIR\_RegisterIRIntrHandlerCB(sendMsgToAppTaskFromISR)函数, 注册 IR 中断处理回调函数, 实现在 IR 中断处理函数中发送 msg 到 app task。

```
void IR_Send_Init(void)
{
    UIR_TransInit();
    UIR_RegisterIRIntrHandlerCB(sendMsgToAppTaskFromISR);
}
```

在 UIR\_TransInit 中, 执行 Board\_IR\_Init 函数, 配置 PAD, PINMUX, 设置 P2\_5 为 IRDA\_TX 功能。

在 UIR\_TransInit 中, 执行 Driver\_IR\_Init 函数, 对 IR 外设进行初始化:

- 设置 IR 发送频率为 38K, IR 载波占空比为 1/2, IR 模式为 IR 发送模式;
- 设置不反转 IR 发送数据, 设置空闲状态输出低电平;
- 设置 IR 发送 FIFO 的阈值为 2;

使能 IR 外设发送功能; 更新 IR 中断处理函数 IR\_Handler 到 IR 中断向量表 IR\_VECTORn。

```
void Driver_IR_Init(uint32_t freq)
{
    .....
    IR_InitStruct.IR_Freq        = freq;
    IR_InitStruct.IR_DutyCycle   = 2; /* !< 1/2 duty cycle */
    IR_InitStruct.IR_Mode        = IR_MODE_TX;
```

```
IR_InitStruct.IR_TxInverse      = IR_TX_DATA_NORMAL;
IR_InitStruct.IR_TxFIFOThrLevel = IR_TX_FIFO_THR_LEVEL;
IR_InitStruct.IR_TxIdleLevel    = IR_IDLE_OUTPUT_LOW;
.....
IR_Cmd(IR_MODE_TX, ENABLE);
RamVectorTableUpdate(IR_VECTORn, IR_Handler);
.....
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时, 执行 app\_handle\_dev\_state\_evt 函数, 执行 IR\_Raw\_Packet\_Send(ir\_raw\_data\_buf, ...) 函数, 发送 ir\_raw\_data\_buf 数据:

- 启动 IR 编码协议发送, 并使能 IR 发送 FIFO 数据个数小于设置的发送阈值中断 (IR\_INT\_TF\_LEVEL);
- 判断状态为 NO\_ERROR 时, 翻转 P0\_1 (LED0)。

```
void IR_Raw_Packet_Send(uint32_t *pBuf, uint32_t len)
{
    .....
    /* Enable IR threshold interrupt: IR_INTConfig(IR_INT_TF_LEVEL, ENABLE) */
    status = UIR_OneFrameTransmitting(&uir_param_info, DISABLE);
    if (UIR_STATUS_NO_ERROR == status)
    {
        LED_IR_Send_Swap();
    }
}
```

当 IR 发送 FIFO 数据个数小于设置的发送阈值 (此例中设为 2) 时, 触发 IR\_INT\_TF\_LEVEL 中断, 或当 IR 发送 FIFO 为空时, 触发 IR\_INT\_TF\_EMPTY 中断, 进入中断处理函数 IR\_Handler:

- 屏蔽 IR\_INT\_TF\_EMPTY 和 IR\_INT\_TF\_LEVEL 中断;
- 判断 IR\_INT\_TF\_LEVEL 中断状态位是否为 SET, 若为 SET 则: (详情参考: IR -> TX)
  - ➔ 如果剩余发送数据个数大于发送 FIFO 大小, 往 IR 发送 FIFO 中塞满数据; 否则如果剩余发送数据个数大于 0, 往 IR 发送 FIFO 中塞剩余数据; 否则失能 IR\_INT\_TF\_LEVEL 中断, 并使能 IR\_INT\_TF\_EMPTY 中断。
    - ➔ 清除 IR\_INT\_TF\_LEVEL 中断挂起位;
- 判断 IR\_INT\_TF\_EMPTY 中断状态位是否为 SET, 若为 SET 则(发送 IR 编码协议后, 发送 Repeat code, 发送 msg 给 task):

- ➔ 发送 Repeat code;
  - ➔ 失能 IR\_INT\_TF\_EMPTY 中断，清除 IR\_INT\_TF\_EMPTY 中断挂起位；
  - ➔ 调用 pfnIRIntrHandlerCB 函数：定义消息类型 IO\_MSG\_TYPE\_IR，子类型 IO\_MSG\_TYPE\_IR\_SEND\_COMPLETE，发送 msg 给 task；
- 取消屏蔽 IR\_INT\_TF\_EMPTY 和 IR\_INT\_TF\_LEVEL 中断。

```
void IR_Handler(void)
{
    IR_MaskINTConfig(IR_INT_TF_EMPTY | IR_INT_TF_LEVEL, ENABLE);
    if (IR_GetINTStatus(IR_INT_TF_LEVEL) == SET)
        .....
    if (IR_GetINTStatus(IR_INT_TF_EMPTY) == SET)
    {
        if (IR_DataStruct.isSendRepeatcode)
        {
            if ((uir_param_info_base.uir_protocol_index == UIR_CODESET_TYPE_LC7464M) ||
                (uir_param_info_base.uir_protocol_index == UIR_CODESET_TYPE_DVB) ||
                (uir_param_info_base.uir_protocol_index == UIR_CODESET_TYPE_MIT_C8D8) ||
                (uir_param_info_base.uir_protocol_index == UIR_CODESET_TYPE_KONICA))
                UIR_BurstSendCmd(ProtocolLib[uir_param_info_base.uir_protocol_index].repeat_interval,
ENABLE);
        }
        IR_INTConfig(IR_INT_TF_EMPTY, DISABLE);
        IR_ClearINTPendingBit(IR_INT_TF_EMPTY_CLR);
        .....
        if (pfnIRIntrHandlerCB)
        {
            pfnIRIntrHandlerCB();
        }
    }
    IR_MaskINTConfig(IR_INT_TF_LEVEL | IR_INT_TF_EMPTY, DISABLE);
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_IR，子类型为 IO\_MSG\_TYPE\_IR\_SEND\_COMPLETE，执行 ir\_send\_msg\_proc 函数，执行 NEC\_SendCode：

- 定义 uir\_param\_info 并赋值；

- 发送 1 帧 IR 编码协议;
- 发送连续 IR 编码协议。

```
void NEC_SendCode(void)
{
    memset(&uir_param_info, 0, sizeof(uir_param_info));
    uir_param_info.uir_protocol_index = UIR_CODESET_TYPE_NECK;
    uir_param_info.custom_code_value[0] = 0x55;
    uir_param_info.custom_code_value[1] = (uint8_t)(~(0x55));
    uir_param_info.custom_code_length = 16;
    uir_param_info.key_code_value[0] = 0xAA;
    uir_param_info.key_code_value[1] = (uint8_t)(~(0xAA));
    uir_param_info.key_code_length = 16;
    .....
    UIR_OneFrameTransmitting(&uir_param_info, DISABLE);
    status = UIR_StartContinuouslyTransmitting(&uir_param_info);
    status = UIR_StopContinuouslyTransmitting();
}
```

使用 IR 遥控编码分析仪接收 IR 发送的数据，验证发送数据是否符合预期。如图 10-2 所示，①为发送 1 帧 ir\_raw\_data\_buf 波形，②为连续发送 uir\_param\_info 波形，③为发送 1 帧 uir\_param\_info 波形。

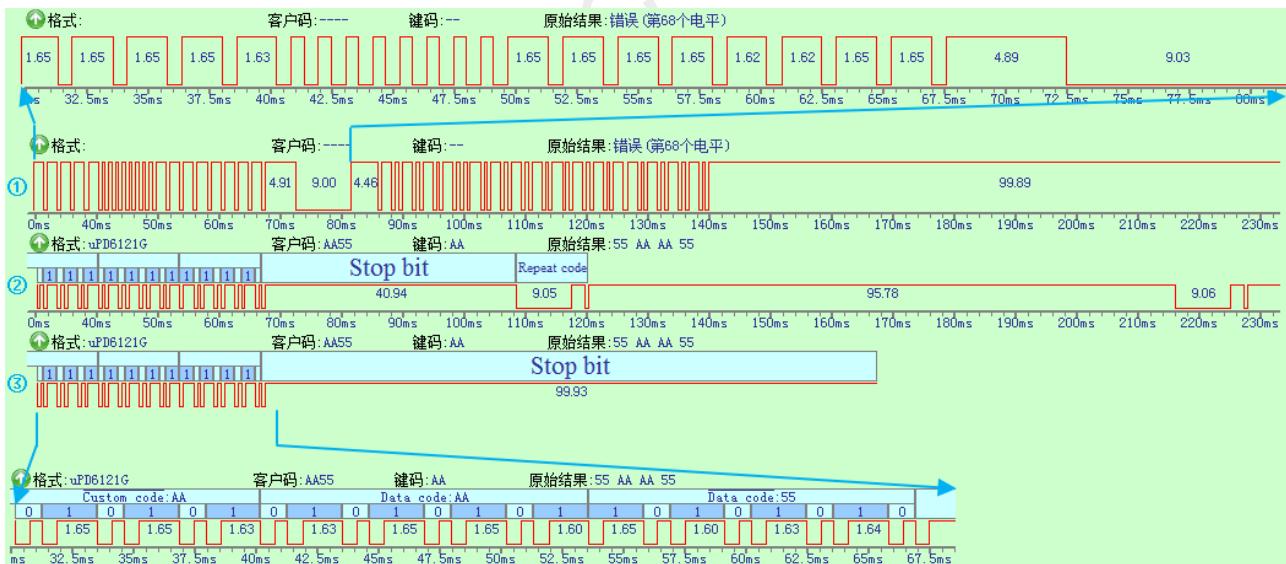


图 11-2 IR 发送数据编码

## 11.4 红外学习

使用 IR 外设的 Rx 功能实现 IR 学习。使用 IR 遥控器向 IR 收发模块发送 IR 数据，通过 IR 中断接收

IR 学习数据并向 app\_task 发送消息事件，app\_task 检测到消息事件，在 app 层解析消息，执行用户程序：对 IR 学习解码，在 DebugAnalyser 工具上，打印 IR 学习的解码后数据信息。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\IR\Learn。

## 11.4.1 硬件设计

硬件连接：P2\_5 -> IR 收发模块（带学习功能）的接收端，P0\_2 -> LED0。

EVB 外接 IR 收发模块，连接 P2\_5 和 IR 收发模块的接收端，连接 VCC 和 IR 收发模块的 VCC，连接 GND 和 IR 收发模块的 GND。

在 EVB 上，使用跳帽短接 J24，连接 P0\_2 和 LED0

## 11.4.2 软件流程

配置 PAD、PINMUX，设置 P0\_2 为 GPIO 输出功能；初始化 GPIO 外设，初始状态：P0\_2 置零。

配置 PAD、PINMUX：设置 P2\_5 为 IRDA\_RX 功能；

app\_task 初始化。

```
os_task_create(&app_task_handle, "app", app_main_task, 0, APP_TASK_STACK_SIZE,  
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 ir\_learn\_timer\_init 函数，初始化 IR 软件定时器，定时时间 20s，回调函数为 ir\_learn\_timer\_callback。

```
os_timer_create(&IR_Learn_Timer, "ir_learn_timer", 1, IR_LEARN_TIMEOUT, false,  
ir_learn_timer_callback);
```

在回调函数 ir\_learn\_timer\_callback 中，执行 ir\_learn\_exit 函数，用于学习超时退出 IR 学习。

```
static void ir_learn_timer_callback(TimerHandle_t pxTimer)  
{  
    ir_learn_exit();  
}
```

在 driver\_init 中，执行 ir\_learn\_module\_init 函数：

- 启动定时器：IR\_Learn\_Timer；
- 执行 ir\_trans\_rx\_handler\_cb(ir\_learn\_send\_msg\_from\_isr) 函数，注册 IR 中断处理回调函数，实现在 IR 中断处理函数中发送 msg 到 app task；
- 执行 ir\_learn\_init 函数，执行 ir\_trans\_rx\_init 函数：
  - ➔ 初始化 loop queen；
  - ➔ 更新 IR 中断处理函数 IR\_Handler 到 IR 中断向量表 IR\_VECTORn；
  - ➔ 初始化 IR 外设：
    - i. 设置 IR 接收频率为 40MHz，占空比为 1/2，接收模式，自动接收，下降沿触发，接收 FIFO

阈值为 20，计数器阈值为 240000 等；

ii. 使能 IR 外设接收功能，清除 IR 接收 FIFO；

iii. 使能 IR 接收 FIFO 的数据个数达到设置的接收阈值中断 (IR\_INT\_RF\_LEVEL) 和接收计数器的个数达到设置的阈值中断 (IR\_INT\_RX\_CNT\_THR)，同时取消屏蔽 IR\_INT\_RF\_LEVEL 和 IR\_INT\_RX\_CNT\_THR 中断。

```
void ir_learn_module_init(void)
{
    if(true == os_timer_start(&IR_Learn_Timer))
    .....
    ir_trans_rx_handler_cb(ir_learn_send_msg_from_isr);
    ir_learn_init();
}
```

开始任务调度。

```
os_sched_start();
```

使用 IR 遥控器向 IR 收发模块发送 IR 数据。

当 IR 接收 FIFO 的数据个数达到设置的接收阈值（此例中设为 20）时，触发 IR\_INT\_RF\_LEVEL 中断，或当接收计数器的个数达到设置的阈值（此例中设为 240000）时，触发 IR\_INT\_RX\_CNT\_THR 中断，进入中断处理函数 IR\_Handler：

- 屏蔽 IR\_INT\_RF\_LEVEL 和 IR\_INT\_RX\_CNT\_THR 中断；
- 判断 IR\_INT\_RF\_LEVEL 中断状态位是否为 SET，若为 SET 则：
  - ◆ 接收 IR 数据，入列到 IR\_RX\_Queue 中；
  - ◆ 清除 IR\_INT\_RF\_LEVEL 中断挂起位；
- 判断 IR\_INT\_RX\_CNT\_THR 中断状态位是否为 SET，若为 SET 则：
  - ◆ 接收 IR 剩余数据，入列到 IR\_RX\_Queue 中；
  - ◆ 清除 IR\_INT\_RX\_CNT\_THR 中断挂起位；
- 调用 pFn\_IR\_RX\_Handler\_CB 函数，定义消息类型 IO\_MSG\_TYPE\_IR，子类型 IO\_MSG\_TYPE\_IR\_LEARN\_STOP 或 IO\_MSG\_TYPE\_IR\_LEARN\_DATA，发送 msg 给 task；
- 取消屏蔽 IR\_INT\_RF\_LEVEL 和 IR\_INT\_RX\_CNT\_THR 中断。

```
void IR_RX_Handler(void)
{
    IR_MaskINTConfig(IR_INT_RF_LEVEL | IR_INT_RX_CNT_THR, ENABLE);
    if(IR_GetINTStatus(IR_INT_RF_LEVEL) == SET)
    {
        data_len = IR_GetRxDataLen();
        ir_loop_queue_data_in(&IR_RX_Queue, data_len);
    }
}
```

```
    IR_ClearINTPendingBit(IR_INT_RF_LEVEL_CLR);  
}  
  
if (IR_GetINTStatus(IR_INT_RX_CNT_THR) == SET)  
{  
    data_len = IR_GetRxDataLen();  
    ir_loop_queue_data_in(&IR_RX_Queue, data_len);  
    IR_ClearINTPendingBit(IR_INT_RX_CNT_THR_CLR);  
    ir_learn_end_flag = true;  
}  
  
if (pFn_IR_RX_Handler_CB)  
{  
    pFn_IR_RX_Handler_CB(ir_learn_end_flag);  
}  
  
IR_MaskINTConfig(IR_INT_RF_LEVEL | IR_INT_RX_CNT_THR, DISABLE);  
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 中，消息类型为 IO\_MSG\_TYPE\_IR，执行 ir\_learn\_handle\_msg，判断子类型：

- 子类型为 IO\_MSG\_TYPE\_IR\_LEARN\_DATA：如果 IR 学习解码状态为特定状态，停止 IR 学习。
- 子类型为 IO\_MSG\_TYPE\_IR\_LEARN\_STOP：
  - ❖ IR 学习数据解码、获取频率、数据转换等；
  - ❖ 打印 IR 学习的频率、数据长度、数据等信息；
  - ❖ 退出 IR 学习；
  - ❖ 重新执行 ir\_learn\_module\_init 函数：启动定时器，注册回调函数，初始化 IR 外设等。

```
bool ir_learn_handle_msg(T_IO_MSG *io_ir_msg)  
{  
    if (sub_type == IO_MSG_TYPE_IR_LEARN_DATA)  
    {  
        status = ir_learn_decode(&IR_Learn_Packet);  
        if (status == IR_LEARN_EXCEED_SIZE)  
            ir_learn_deinit();  
        .....  
    }  
    else if (sub_type == IO_MSG_TYPE_IR_LEARN_STOP)  
    {  
        ir_learn_decode(&IR_Learn_Packet); /* Pick up the last ir data*/  
    }  
}
```

```
    ir_learn_freq(&IR_Learn_Packet); /* Decode IR carrier frequency */  
    ir_learn_data_convert(&IR_Learn_Packet); /* Data reduction */  
    .....  
    for (uint32_t i = 0; i < IR_Learn_Packet.buf_index; i++) /* Print decode result */  
        APP_PRINT_INFO2("learn data%od: 0x%ox.", i, IR_Learn_Packet.ir_buf[i]);  
    ir_learn_exit();  
    ir_learn_module_init();  
}  
}
```

在 app\_handle\_io\_msg 中，消息类型为 IO\_MSG\_TYPE\_IR，执行 led\_ir\_learn\_swap：翻转 P0\_2。

```
void app_handle_io_msg(T_IO_MSG io_msg)  
{  
    case IO_MSG_TYPE_IR:  
    {  
        if (ir_learn_handle_msg(&io_msg))  
        {  
            extern void led_ir_learn_swap(void);  
            led_ir_learn_swap();  
        }  
    }  
    .....  
}
```

使用 IR 遥控器向 IR 收发模块发送 IR 数据，对 IR 学习解码，在 DebugAnalyser 工具上，打印 IR 学习的解码后数据信息。

## 11.5 红外 GDMA 发送

使用 IR 外设利用 GDMA 搬运发送数据，实现 TX 功能。使用红外压控编码分析仪接收 IR 发送的数据，并观察发送数据是否符合预期。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ IR\ Tx+GDMA。

### 11.5.1 硬件设计

硬件连接：P2\_5 -> IR 收发模块的发送端。

EVB 外接 IR 收发模块，连接 P2\_5 和 IR 收发模块的发送端，连接 VCC 和 IR 收发模块的 VCC，连接 GND 和 IR 收发模块的 GND。

## 11.5.2 软件流程

定义 IR 发送数据数组，并赋值：有载波数据用载波个数与 0x80000000 进行或运算表示，无载波数据用载波个数与 0x00000000 进行或运算表示，载波个数为： timing / CarrierFreq。

```
void ir_demo(void)
{
    IR_Send_Data.CarrierFreq = 38000;
    IR_Send_Data.DataLen = IO_TEST_GDMA_TRANSFER_SIZE;
    IR_Send_Data.DataBuff[0] = 0x80000000 | 0x200;
    IR_Send_Data.DataBuff[1] = 0x00000000 | 0x100;
    for (uint16_t i = 2; i < IR_Send_Data.DataLen - 1;)
    {
        IR_Send_Data.DataBuff[i] = 0x80000000 | (0x0A + i * 5);
        IR_Send_Data.DataBuff[i + 1] = 0x00000000 | (0x14 + i * 5);
        i += 2;
    }
    IR_Send_Data.DataBuff[IR_Send_Data.DataLen - 1] = 0x80000000 | 0x800;

    /* Test data buffer */
    for (uint32_t i = 0; i < IO_TEST_GDMA_TRANSFER_SIZE; i++)
    {
        GDMA_Send_Buf[i] = IR_Send_Data.DataBuf[i];
    }
    .....
}
```

引脚定义。

```
#define IR_TX_PIN P2_5
```

配置 PAD、PINMUX：设置引脚 P2\_5 为 IR 发送引脚、PINMUX 模式、PowerOn、无内部上拉、输出使能、输出低；分配引脚 P2\_5 为 IRDA\_TX 功能。

使能 IR 时钟；

初始化 IR 外设：

- IR\_Freq 设置为 vFreq，即 IR 发送频率为 38KHz；
- IR\_DutyCycle 设置为 3，即 IR 载波占空比为 1/3；
- IR\_Mode 设置为 IR\_MODE\_TX，即 IR 模式为 IR 发送模式；
- IR\_TxInverse 设置为 IR\_TX\_DATA\_NORMAL，即不反转 IR 发送数据；

- IR\_TxFIFOThrLevel 设置为 IR\_TX\_FIFO\_THR\_LEVEL，即 IR 发送 FIFO 阈值为 2；
  - IR\_TxDmaEn 设置为 ENABLE，即使能 TX 的 GDMA 传输；
  - IR\_TxWaterLevel 设置为 15；
- 使能 IR 外设发送功能；

```
void driver_ir_init(uint32_t vFreq)
{
    RCC_PeriphClockCmd(APBPeriph_IR, APBPeriph_IR_CLOCK, ENABLE);

    IR_InitTypeDef IR_InitStruct;
    IR_StructInit(&IR_InitStruct);

    IR_InitStruct.IR_Freq          = vFreq;
    IR_InitStruct.IR_DutyCycle     = 2;
    IR_InitStruct.IR_Mode          = IR_MODE_TX;
    IR_InitStruct.IR_TxInverse     = IR_TX_DATA_NORMAL;
    IR_InitStruct.IR_TxFIFOThrLevel = IR_TX_FIFO_THR_LEVEL;
    IR_InitStruct.IR_TxDmaEn       = ENABLE;
    IR_InitStruct.IR_TxWaterLevel   = 15;
    IR_Init(&IR_InitStruct);
    .....
}
```

执行 driver\_ir\_gdma\_init 函数，对 GDMA 外设进行初始化：

- 使用 GDMA 通道 1；
- GDMA 的传输方向为内存到外设传输； GDMA\_SourceAddr 设置为 GDMA\_Send\_Buf； GDMA\_DestinationAddr 设 IR 的 TX\_FIFO；
- 使能 GDMA 通道 1 总传输完成中断 (GDMA\_INT\_Transfer)。

执行 GDMA\_Cmd 函数：使能 GDMA 传输；

```
void driver_ir_gdma_init(void)
{
    GDMA_InitStruct.GDMA_ChannelNum      = IO_TEST_GDMA_CHANNEL_MUM;
    GDMA_InitStruct.GDMA_DIR             = GDMA_DIR_MemoryToPeripheral;
    .....
    GDMA_InitStruct.GDMA_SourceAddr      = (uint32_t) ( GDMA_Send_Buf );
    GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t) (&IR->TX_FIFO);
    .....
    GDMA_INTConfig(IO_TEST_GDMA_CHANNEL_MUM, GDMA_INT_Transfer, ENABLE) ;
    .....
}
```

```
    GDMA_Cmd(IO_TEST_GDMA_CHANNEL_MUM, ENABLE);  
}
```

执行 IR\_Cmd 函数：使能 IR 外设发送功能；

```
void ir_demo(void)  
{  
    .....  
    IR_Cmd(IR_MODE_TX, ENABLE);  
}
```

当 GDMA 搬运数据完成时，触发 GDMA\_INT\_Transfer 中断，进入 GDMA 中断处理函数 ADC\_GDMA\_Channel\_Handler：

- 使能 GDMA 通道 1 总传输完成中断 (GDMA\_INT\_Transfer)；
- 失能 GDMA 通道 1 传输；
- 打印 GDMA 传输完成信息；
- 清除 GDMA 通道 1 GDMA\_INT\_Transfer 中断挂起位。

```
void IO_TEST_GDMA_Channel_Handler (void)  
{  
    GDMA_INTConfig(IO_TEST_GDMA_CHANNEL_MUM, GDMA_INT_Transfer, DISABLE);  
    GDMA_Cmd(IO_TEST_GDMA_CHANNEL_MUM, DISABLE);  
    DBG_DIRECT("IO_TEST_GDMA_Channel_Handler\r\n");  
    GDMA_ClearINTPendingBit(IO_TEST_GDMA_CHANNEL_MUM, GDMA_INT_Transfer);  
}
```

在 DebugAnalyser 工具上，打印 GDMA 搬运数据完成信息；使用红外遥控编码分析仪接收 IR 发送的数据，并观察发送数据是否符合预期。

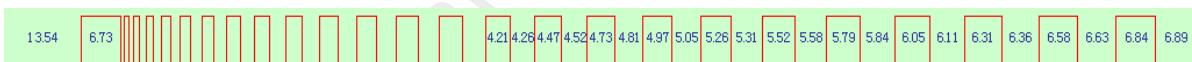


图 11-3 IR-GDMA 发送数据编码

## 11.6 红外 GDMA 接收

使用 IR 外设用 GDMA 搬运接收数据，实现 RX 功能。使用 IR 遥控器向 IR 收发模块发送 IR 数据，通过 GDMA 中断接收 IR 数据，在 DebugAnalyser 工具上，打印 GDMA 搬运 IR 数据信息。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ IR\ Rx+GDMA。

### 11.6.1 硬件设计

硬件连接：P2\_5 -> IR 收发模块的接收端。

EVB 外接 IR 收发模块，连接 P2\_5 和 IR 收发模块的接收端，连接 VCC 和 IR 收发模块的 VCC，连接 GND 和 IR 收发模块的 GND。

在 EVB 上，使用跳线连接 P2\_6 和 LED0。

## 11.6.2 软件流程

配置 PAD、PINMUX，设置 P2\_6 为 PWM 功能；初始化 PWM 外设，初始状态：使用 TIM7，实现输出周期为 0.04s、占空比为 50% 的 PWM 功能(详情参考：TIM ->PWM)。

使能 TIM7，开启 PWM 外设。

```
void ir_demo (void)
{
    board_pwm_init();
    driver_pwm_init();
    TIM_Cmd(PWM_TIMER_NUM, ENABLE);
}
```

配置 PAD、PINMUX：设置 P2\_5 为 IRDA\_RX 功能；

执行 driver\_ir\_init 函数，对 IR 外设进行初始化：

- 设置 IR 接收频率为 40MHz，IR 模式为 IR 接收模式，启动方式为 IR 自动接收模式，数据接收触发方式为下降沿触发；
- 设置 IR 接收 FIFO 阈值为 30，当 IR 接收 FIFO 满时丢弃最新数据，过滤低于 50ns 的杂波数据；
- 设置触发接收计数器阈值中断的电平类型为高电平，IR\_RxCntThr 设置为 0xEEEE08；
- IR\_RxDmaEn 设置为 ENABLE，即使能 RX 的 GDMA 传输；
- IR\_TxWaterLevel 设置为 4；

使能 IR 外设接收功能，清除 IR 接收 FIFO。

```
void driver_ir_init(void)
{
    .....
    IR_InitStruct.IR_Freq          = 40000000; /* IR carrier frequency is 38KHz */
    IR_InitStruct.IR_Mode          = IR_MODE_RX; /* IR receiveing mode */
    IR_InitStruct.IR_RxStartMode   = IR_RX_AUTO_MODE;
    IR_InitStruct.IR_RxFIFOThrLevel = IR_RX_FIFO_THR_LEVEL;
    IR_InitStruct.IR_RXFIFOFullCtrl = IR_RX_FIFO_FULL_DISCARD_NEWEST;
    IR_InitStruct.IR_RxFilterTime  = IR_RX_FILTER_TIME_50ns;
    IR_InitStruct.IR_RxTriggerMode = IR_RX_FALL_EDGE;
    IR_InitStruct.IR_RxCntThrType = IR_RX_Count_High_Level;
```

```
IR_InitStruct.IR_RxCntThr      = 0xEEEE08;  
IR_InitStruct.IR_RxDmaEn       = ENABLE;  
IR_InitStruct.IR_RxWaterLevel   = 4;  
IR_Init(&IR_InitStruct);  
IR_Cmd(IR_MODE_RX, ENABLE);  
IR_ClearRxFIFO();  
}
```

执行 driver\_ir\_gdma\_init 函数，对 GDMA 外设进行初始化：

- 使用 GDMA 通道 1；
- GDMA 的传输方向为外设到内存传输；GDMA\_SourceAddr 设置为 IR 的 RX\_FIFO；GDMA\_DestinationAddr 设置为 GDMA\_Recv\_Buf；
- 使能 GDMA 通道 1 总传输完成中断（GDMA\_INT\_Transfer）。

执行 GDMA\_Cmd 函数：使能 GDMA 传输；

```
void driver_ir_gdma_init(void)  
{  
    GDMA_InitStruct.GDMA_ChannelNum      = IO_TEST_GDMA_CHANNEL_MUM;  
    GDMA_InitStruct.GDMA_DIR              = GDMA_DIR_PeripheralToMemory  
    .....  
    GDMA_InitStruct.GDMA_SourceAddr       = (uint32_t)(&IR->RX_FIFO);  
    GDMA_InitStruct.GDMA_DestinationAddr  = (uint32_t)(GDMA_Recv_Buf);  
    .....  
    GDMA_INTConfig(IO_TEST_GDMA_CHANNEL_MUM, GDMA_INT_Transfer, ENABLE);  
    .....  
    GDMA_Cmd(IO_TEST_GDMA_CHANNEL_MUM, ENABLE);  
}
```

当 GDMA 搬运数据完成时，触发 GDMA\_INT\_Transfer 中断，进入 GDMA 中断处理函数 IO\_TEST\_GDMA\_Channel\_Handler：

- 失能 GDMA 通道 1 总传输完成中断（GDMA\_INT\_Transfer）；
- 失能 GDMA 通道 1 传输；
- 保存 GDMA 搬运数据长度；
- 执行 io\_handle\_gdma\_msg 函数，处理 GDMA 搬运数据；
- 清除 GDMA 通道 1 GDMA\_INT\_Transfer 中断挂起位。

```
void IO_TEST_GDMA_Channel_Handler (void)  
{  
    GDMA_INTConfig(IO_TEST_GDMA_CHANNEL_MUM, GDMA_INT_Transfer, DISABLE);
```

```
GDMA_Cmd(IO_TEST_GDMA_CHANNEL_MUM, DISABLE);
IR_GDMA_Rev_Data_Len = IO_TEST_GDMA_TRANSFER_SIZE;
io_handle_gdma_msg();
GDMA_ClearINTPendingBit(IO_TEST_GDMA_CHANNEL_MUM, GDMA_INT_Transfer);
}
```

在 io\_handle\_gdma\_msg 中，打印传输数据长度以及数据信息。

```
void io_handle_gdma_msg (void)
{
    DBG_DIRECT("io_handle_gdma_msg:      IR_GDMA_Rev_Data_Len      =      %d      \r\n",
IR_GDMA_Rev_Data_Len);
    for (uint32_t i = 0; i < IR_GDMA_Rev_Data_Len; i++)
    {
        DBG_DIRECT("io_handle_gdma_msg:      GDMA_Recv_Buf[%d]      =      0x%x      \r\n", i,
GDMA_Recv_Buf[i]);
    }
}
```

使用 IR 遥控器向 IR 收发模块发送 IR 数据。在 DebugAnalyser 工具上，打印通过 GDMA 接收到的 IR 数据以及其长度信息。

## 12 集成电路内置音频总线(I2S)

专用硬件设备：逻辑分析仪

专用测试软件：DSview V1.1.2

### 12.1 音频数据传输

使用 I2S 输出数据，并用逻辑分析仪观察波形。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\I2S\Tx\_polling。

#### 12.1.1 硬件设计

硬件连接：P3\_2 -> Pin\_frame, P3\_3 -> Pin\_clock, P4\_0 -> Pin\_data。

EVB 外接逻辑分析仪，连接 P3\_2 和 Pin\_frame, P3\_3 和 Pin\_clock, P4\_0 和 Pin\_data。

#### 12.1.2 软件流程

引脚定义：

```
#define I2S_LRCK_PIN          P3_2
#define I2S_BCLK_PIN            P3_3
#define I2S_DATA_TX_PIN         P4_0
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、无内部上拉、输出使能、输出低；

配置 PINMUX：分配引脚分别为 I2S\_BCLK\_PINMUX、I2S\_LRCK\_PINMUX、I2S\_DATA\_TX\_PINMUX 功能。

```
void board_i2s_init(void)
{
    Pad_Config(I2S_BCLK_PIN,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
    PAD_OUT_ENABLE, PAD_OUT_LOW);

    Pad_Config(I2S_LRCK_PIN,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
    PAD_OUT_ENABLE, PAD_OUT_LOW);

    Pad_Config(I2S_DATA_TX_PIN, PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
    PAD_OUT_ENABLE, PAD_OUT_LOW);

    Pinmux_Config(I2S_BCLK_PIN, I2S_BCLK_PINMUX);
    Pinmux_Config(I2S_LRCK_PIN, I2S_LRCK_PINMUX);
    Pinmux_Config(I2S_DATA_TX_PIN, I2S_DATA_TX_PINMUX);
}
```

使能 I2S0 时钟；

初始化 I2S0 外设：

- I2S\_ClockSource 设置为 I2S\_CLK\_40M，即 I2S 时钟源为 40M；
- I2S\_BClockMi 设置为 0x271，I2S\_BClockNi 设置为 0x10，即 BCLK = I2S\_ClockSource \* (I2S\_BClockNi / I2S\_BClockMi) = 1024K，LRCK = BCLK/64 = 16K；
- I2S\_DeviceMode 设置为 I2S\_DeviceMode\_Master，即主设备模式；
- I2S\_ChannelType 设置为 I2S\_Channel\_stereo，即传输通道类型为双声道输出；
- I2S\_DataWidth 设置为 I2S\_Width\_16Bits，即数据宽度为 16bits 数据宽度；
- I2S\_DataFormat 设置为 I2S\_Mode，即数据格式为 I2S 格式；
- I2S\_DMACmd 设置为 I2S\_DMA\_DISABLE，即失能 DMA 数据传输功能；

使能 I2S0 发送模式。

```
void driver_i2s_init(void)
{
    RCC_PeriphClockCmd(APB_I2S, APB_I2S_CLOCK, ENABLE);

    I2S_InitTypeDef I2S_InitStruct;
    I2S_StructInit(&I2S_InitStruct);

    I2S_InitStruct.I2S_ClockSource      = I2S_CLK_40M;
    I2S_InitStruct.I2S_BClockMi        = 0x271; /* <!LRCK = 16K */
    I2S_InitStruct.I2S_BClockNi        = 0x10; /* <!BCLK = 1024K */
    I2S_InitStruct.I2S_DeviceMode     = I2S_DeviceMode_Master;
    I2S_InitStruct.I2S_ChannelType    = I2S_Channel_stereo;
    I2S_InitStruct.I2S_DataWidth       = I2S_Width_16Bits;
    I2S_InitStruct.I2S_DataFormat      = I2S_Mode;
    I2S_InitStruct.I2S_DMACmd         = I2S_DMA_DISABLE;

    I2S_Init(I2S_NUM, &I2S_InitStruct);
    I2S_Cmd(I2S_NUM, I2S_MODE_TX, ENABLE);
}
```

定义发送数据，循环发送数据。

```
void i2s_senddata(void)
{
    uint32_t i = 0x12348800;
    while (1)
    {
        if (I2S_GetTxFIFOFreeLen(I2S_NUM))
        {
```

```
/* 16bit format, lower half word send first! */  
I2S_SendData(I2S_NUM, i++);  
}  
}  
}
```

配置逻辑分析仪，检测 I2S 输出，观察波形。

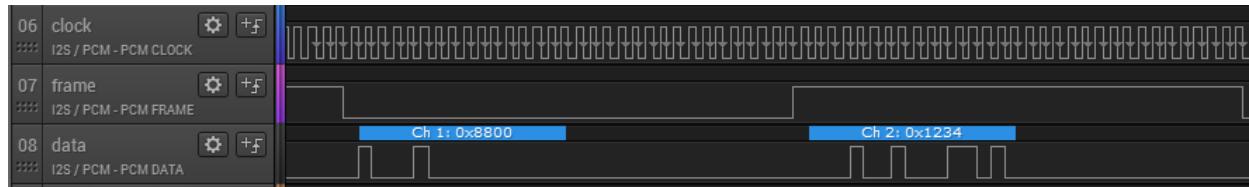


图 12-1 I2S 输出波形

## 13 编解码器(CODEC)

专用硬件设备：FT232 usb 转串口、AMIC、DMIC、PDM 播放器

专用测试软件：串口调试工具、PCM\_Data\_Parser、Cool Edit Pro

### 13.1 模拟麦克

实现 AMIC 采集模拟语音经 CODEC 编码的功能。AMIC 采集模拟语音数据，经 CODEC 编码，送到 I2S0 接收 FIFO，利用 GDMA 将数据搬运到 UART0，PC 端收到 UART 传输的数据，使用 PCM\_Data\_Parser 解析数据，使用 Cool Edit Pro 播放录制的语音。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ CODEC\ AMIC。

#### 13.1.1 硬件设计

硬件连接：H0 -> MICBIAS， P2\_6 -> MIC\_P， P2\_7 -> MIC\_N；同时： P2\_4 -> RX， P2\_5 -> TX。

EVB 外接 AMIC 模块，连接 H0 和 MICBIAS，GND 和 GND，P2\_6 和 MIC\_P，P2\_7 和 MIC\_N；EVB 外接 FT232 模块，连接 P2\_4 和 RX，P2\_5 和 TX。

#### 13.1.2 软件流程

配置 PAD、PINMUX，设置 P2\_4、P2\_5 为 UART0 功能；

配置 PAD：设置引脚，SW 模式、PowerOn、无内部上拉、输出使能或失能、输出高或低；

```
void board_codec_init(void)
{
    Pad_Config(H_0, PAD_SW_MODE, PAD_NOT_PWRON, PAD_PULL_NONE, PAD_OUT_ENABLE,
PAD_OUT_HIGH);

    Pad_Config(P2_6, PAD_SW_MODE, PAD_NOT_PWRON, PAD_PULL_NONE, PAD_OUT_DISABLE,
PAD_OUT_HIGH);

    Pad_Config(P2_7, PAD_SW_MODE, PAD_NOT_PWRON, PAD_PULL_NONE, PAD_OUT_DISABLE,
PAD_OUT_LOW);
}
```

初始化 UART 外设：

➤ 设置波特率为 3000000，使能 GDMA 发送功能等；

初始化 I2S 外设：

➤ 设置 I2S 时钟源为 40M，位时钟为 16K，主设备模式，单声道输出，I2S 格式等；

初始化 GDMA 外设：

- 设置为 GDMA 通道 0;
- 设置传输方向为外设到外设传输, 源端地址为 I2S0 的 RX\_DR, 目的端地址为 UART 的 RB\_THR;
- 使能总传输完成中断 (GDMA\_INT\_Transfer) 等;

使能 CODEC 时钟;

初始化 CODEC 外设:

- CODEC\_MicType 设置为 CODEC\_AMIC, 即使用模拟麦克风;
- CODEC\_MICBstMode 设置为 MICBST\_Mode\_Differential, 即差分模式;
- CODEC\_SampleRate 设置为 SAMPLE\_RATE\_16KHz, 即采样频率为 16KHz;
- CODEC\_I2SFormat 设置为 CODEC\_I2S\_DataFormat\_I2S, 即 I2S 格式;
- CODEC\_I2SDataWidth 设置为 CODEC\_I2S\_DataWidth\_16Bits, 即数据宽度为 16bit;

```
void driver_codec_init(void)
{
    RCC_PерiphClockCmd(APBPeriph_CODEC, APBPeriph_CODEC_CLOCK, ENABLE);
    CODEC_InitTypeDef CODEC_InitStruct;
    CODEC_StructInit(&CODEC_InitStruct);
    CODEC_InitStruct.CODEC_MicType      = CODEC_AMIC;
    CODEC_InitStruct.CODEC_MICBstMode   = MICBST_Mode_Differential;
    CODEC_InitStruct.CODEC_SampleRate   = SAMPLE_RATE_16KHz;
    CODEC_InitStruct.CODEC_I2SFormat    = CODEC_I2S_DataFormat_I2S;
    CODEC_InitStruct.CODEC_I2SDataWidth = CODEC_I2S_DataWidth_16Bits;
    CODEC_Init(CODEC, &CODEC_InitStruct);
}
```

使能 I2S 发送模式和接收模式; 使能 GDMA 通道 0。

```
void codec_demo(void)
{
    .....
    I2S_Cmd(I2S0, I2S_MODE_TX | I2S_MODE_RX, ENABLE);
    GDMA_Cmd(GDMA_Channel_AMIC_NUM, ENABLE);
}
```

AMIC 采集到模拟语音数据, 经 CODEC 编码, 送到 I2S0 接收 FIFO, 利用 GDMA 将数据搬运到 UART0;  
当数据传输完成时, 触发 GDMA\_INT\_Transfer 中断, 进入中断处理函数 GDMA\_Channel\_AMIC\_Handler:

- 重新设置源端地址和目的端地址; 重新设置 GDMA 传输数据大小;
- 清除 GDMA 通道 0 GDMA\_INT\_Transfer 中断挂起位, 使能 GDMA 通道 0。

```
void GDMA_Channel_AMIC_Handler(void)
{
```

```
GDMA_SetSourceAddress(GDMA_Channel_AMIC, (uint32_t)(&(I2S0->RX_DR)));
GDMA_SetDestinationAddress(GDMA_Channel_AMIC, (uint32_t)(&(UART->RB_THR)));

GDMA_SetBufferSize(GDMA_Channel_AMIC, AUDIO_FRAME_SIZE / 4);

GDMA_ClearINTPendingBit(GDMA_Channel_NUM, GDMA_INT_Transfer);
GDMA_Cmd(GDMA_Channel_NUM, ENABLE);
}
```

将语音数据导入到 PC 端：

- 使用 UART 工具接收数据。选择正确的端口，将波特率设置为 3M，打开端口；开始录音，S-Bee2 进行语音采集、语音编码、发送编码数据；录音完成，将编码数据存储到 txt 文件中；
- 数据中会包含多余的空格，需要进一步处理。用 PCM\_Data\_Parser.exe 处理 txt 文件，输出 audio\_data.data 即是编码后的语音数据；
- 使用 Cool Edit Pro 查看解码后的语音波形，或播放语音。

## 13.2 数字麦克

实现 DMIC 采集数字语音经 CODEC 编码的功能。DMIC 采集数字语音数据，经 CODEC 编码，送到 I2S0 接收 FIFO，利用 GDMA 将数据搬运到 UART0，PC 端收到 UART 传输的数据，使用 PCM\_Data\_Parser 解析数据，使用 Cool Edit Pro 播放录制的语音。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\CODEC\DMIC。

### 13.2.1 硬件设计

硬件连接：P3\_2 -> CLK，P3\_3 -> DATA；同时：P2\_4 -> RX，P2\_5 -> TX。

EVB 外接 DMIC 模块，连接 P3\_2 和 CLK，P3\_3 和 DATA，GND 和 GND，VCC 和 VCC；EVB 外接 FT232 模块，连接 P2\_4 和 RX，P2\_5 和 TX。

### 13.2.2 软件流程

配置 PAD、PINMUX，设置 P2\_4、P2\_5 为 UART0 功能；

引脚定义：

```
#define DMIC_MSBC_CLK_PIN          P3_2
#define DMIC_MSBC_DAT_PIN           P3_3
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、无内部上拉、输出失能、输出低；

配置 PINMUX：分配引脚分别为为 DMIC1\_CLK、DMIC1\_DAT 功能。

```
void board_codec_init(void)
```

```
{  
    Pad_Config(DMIC_MSBC_CLK_PIN, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_NONE,  
    PAD_OUT_DISABLE, PAD_OUT_LOW);  
  
    Pad_Config(DMIC_MSBC_DAT_PIN, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_NONE,  
    PAD_OUT_DISABLE, PAD_OUT_LOW);  
  
    Pinmux_Config(DMIC_MSBC_CLK_PIN, DMIC1_CLK);  
    Pinmux_Config(DMIC_MSBC_DAT_PIN, DMIC1_DAT);  
}
```

初始化 UART 外设:

- 设置波特率为 3000000, 使能 GDMA 发送功能等;

初始化 I2S 外设:

- 设置 I2S 时钟源为 40M, 位时钟为 16K, 主设备模式, 单声道输出, I2S 格式等;

初始化 GDMA 外设:

- 设置为 GDMA 通道 0;
- 设置传输方向为外设到外设传输, 源端地址为 I2S0 的 RX\_DR, 目的端地址为 UART 的 RBTHR;
- 使能总传输完成中断 (GDMA\_INT\_Transfer) 等;

使能 CODEC 时钟;

初始化 CODEC 外设:

- CODEC\_MicType 设置为 CODEC\_DMIC, 即使用数字麦克风;
- CODEC\_DmicClock 设置为 DMIC\_Clock\_2MHz, 即设置 DMIC 的时钟频率为 2M;
- CODEC\_DmicDataLatch 设置为 DMIC\_Rising\_Latch, 即上升沿锁存;
- CODEC\_SampleRate 设置为 SAMPLE\_RATE\_16KHz, 即采样频率为 16KHz;
- CODEC\_I2SFormat 设置为 CODEC\_I2S\_DataFormat\_I2S, 即 I2S 格式;
- CODEC\_I2SDataWidth 设置为 CODEC\_I2S\_DataWidth\_16Bits, 即数据宽度为 16bit;

```
void driver_codec_init(void)  
{  
    RCC_PeriphClockCmd(APBPeriph_CODEC, APBPeriph_CODEC_CLOCK, ENABLE);  
  
    CODEC_InitTypeDef CODEC_InitStruct;  
    CODEC_StructInit(&CODEC_InitStruct);  
    CODEC_InitStruct.CODEC_MicType = CODEC_DMIC;  
    CODEC_InitStruct.CODEC_DmicClock = DMIC_Clock_2MHz;  
    CODEC_InitStruct.CODEC_DmicDataLatch = DMIC_Rising_Latch;  
    CODEC_InitStruct.CODEC_SampleRate = SAMPLE_RATE_16KHz;  
    CODEC_InitStruct.CODEC_I2SFormat = CODEC_I2S_DataFormat_I2S;  
    CODEC_InitStruct.CODEC_I2SDataWidth = CODEC_I2S_DataWidth_16Bits;
```

```
CODEC_Init(CODEC, &CODEC_InitStruct);
DBG_DIRECT("Use dmic test!");
}
```

使能 I2S 发送模式和接受模式；使能 GDMA 通道 0。

```
void codec_demo(void)
{
    .....
    I2S_Cmd(I2S0, I2S_MODE_TX | I2S_MODE_RX, ENABLE);
    GDMA_Cmd(GDMA_Channel_DMIC_NUM, ENABLE);
}
```

DMIC 采集到数字语音数据，经 CODEC 编码，送到 I2S0 接收 FIFO，利用 GDMA 将数据搬运到 UART0；当数据传输完成时，触发 GDMA\_INT\_Transfer 中断，进入中断处理函数 GDMA\_Channel\_DMIC\_Handler：

- 重新设置源端地址和目的端地址；重新设置 GDMA 传输数据大小；
- 清除 GDMA 通道 0 传输完成中断挂起位，使能 GDMA 通道 0。

```
void GDMA_Channel_DMIC_Handler(void)
{
    GDMA_SetSourceAddress(GDMA_Channel_DMIC, (uint32_t)(&(I2S0->RX_DR)));
    GDMA_SetDestinationAddress(GDMA_Channel_DMIC, (uint32_t)(&(UART->RB_THR)));

    GDMA_SetBufferSize(GDMA_Channel_DMIC, AUDIO_FRAME_SIZE / 4);

    GDMA_ClearINTPendingBit(GDMA_Channel_DMIC_NUM, GDMA_INT_Transfer);
    GDMA_Cmd(GDMA_Channel_DMIC_NUM, ENABLE);
}
```

将语音数据导入到 PC 端：

- 使用 UART 工具接收数据。选择正确的端口，将波特率设置为 3M，打开端口；开始录音，S-Bee2 进行语音采集、语音编码、发送编码数据；录音完成，将编码数据存储到 txt 文件中；
- 数据中会包含多余的空格，需要进一步处理。用 PCM\_Data\_Parser.exe 处理 txt 文件，输出 audio\_data.data 即是编码后的语音数据；
- 使用 Cool Edit Pro 查看解码后的语音波形，或播放语音。

### 13.3 脉冲密度调制接口（PDM）

实现 AMIC 采集模拟语音经 CODEC 编码，使用 PDM 接口输出语音的功能。AMIC 采集模拟语音数据，经 CODEC 编码，送到 I2S0 接收 FIFO，利用 GDMA 将数据搬运到 I2S0 发送 FIFO，使用 PDM 接口外接 PDM 播放器，播放录制的语音。

工程目录: \HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ CODEC\ PDM.

### 13.3.1 硬件设计

硬件连接: P3\_2 -> CLK, P3\_3 -> DATA; 同时: H0 -> MICBIAS, P2\_6 -> MIC\_P, P2\_7 -> MIC\_N。

EVB 外接 PDM 播放器, 连接 P3\_2 和 CLK, P3\_3 和 DATA; EVB 外接 AMIC 模块, 连接 H0 和 MICBIAS, GND 和 GND, P2\_6 和 MIC\_P, P2\_7 和 MIC\_N。

### 13.3.2 软件流程

引脚定义:

```
#define PDM_CLK_PIN          P3_2  
#define PDM_DAT_PIN          P3_3
```

配置 PAD: 设置 AMIC 引脚、SW 模式等; 设置 PDM 引脚, PINMUX 模式等;

配置 PINMUX: 分配 PDM 引脚分别为为 LRC\_I\_PCM、BCLK\_I\_PCM 功能

```
void board_codec_init(void)  
{  
    .....  
    Pad_Config(PDM_CLK_PIN,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);  
    Pad_Config(PDM_DAT_PIN,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);  
    Pinmux_Config(PDM_CLK_PIN, LRC_I_PCM);  
    Pinmux_Config(PDM_DAT_PIN, BCLK_I_PCM);  
}
```

初始化 I2S 外设:

➤ 设置 I2S 时钟源为 40M, 位时钟为 16K, 主设备模式, 单声道输出, I2S 格式等;

初始化 GDMA 外设:

➤ 设置为 GDMA 通道 0;  
➤ 设置传输方向为外设到外设传输, 源端地址为 I2S0 的 RX\_DR, 目的端地址为 I2S0 的 TX\_DR;  
➤ 使能总传输完成中断 (GDMA\_INT\_Transfer) 等;

使能 CODEC 时钟;

初始化 CODEC 外设:

➤ CODEC\_MicType 设置为 CODEC\_AMIC, 即使用模拟麦克风;  
➤ CODEC\_MICBstMode 设置为 MICBST\_Mode\_Differential, 即差分模式;  
➤ CODEC\_SampleRate 设置为 SAMPLE\_RATE\_16KHz, 即采样频率为 16KHz;

- CODEC\_I2SFormat 设置为 CODEC\_I2S\_DataFormat\_I2S, 即 I2S 格式;
- CODEC\_I2SDataWidth 设置为 CODEC\_I2S\_DataWidth\_16Bits, 即数据宽度为 16bit;
- CODEC\_DaMute 设置为 DAC\_UuMute, 即使能 DAC 输出;
- CODEC\_DaGain 设置为 0xFF, 即设置 DAC 音量;
- CODEC\_DacZeroDetTimeout 设置为 DAC\_Zero\_DetTimeout\_1024\_16\_Sample, 即 1024\*16 采样次数;

```
void driver_codec_init(void)
{
    RCC_PeriphClockCmd(APBPeriph_CODEC, APBPeriph_CODEC_CLOCK, ENABLE);

    CODEC_InitTypeDef CODEC_InitStruct;
    CODEC_StructInit(&CODEC_InitStruct);

    CODEC_InitStruct.CODEC_MicType          = CODEC_AMIC;
    CODEC_InitStruct.CODEC_MICBstMode       = MICBST_Mode_Differential;
    CODEC_InitStruct.CODEC_SampleRate       = SAMPLE_RATE_16KHz;
    CODEC_InitStruct.CODEC_I2SFormat        = CODEC_I2S_DataFormat_I2S;
    CODEC_InitStruct.CODEC_I2SDataWidth     = CODEC_I2S_DataWidth_16Bits;
    CODEC_InitStruct.CODEC_DaMute          = DAC_UuMute;
    CODEC_InitStruct.CODEC_DaGain          = 0xFF;
    CODEC_InitStruct.CODEC_DacZeroDetTimeout = DAC_Zero_DetTimeout_1024_16_Sample;

    CODEC_Init(CODEC, &CODEC_InitStruct);
}
```

使能 I2S 发送模式和接受模式; 使能 GDMA 通道 0。

```
void codec_demo(void)
{
    .....
    I2S_Cmd(I2S0, I2S_MODE_TX | I2S_MODE_RX, ENABLE);
    GDMA_Cmd(GDMA_Channel_AMIC_NUM, ENABLE);
}
```

AMIC 采集到模拟语音数据, 经 CODEC 编码, 送到 I2S0 接收 FIFO, 利用 GDMA 将数据搬运到 I2S0 发送 FIFO; 当数据传输完成时, 触发 GDMA\_INT\_Transfer 中断, 进入中断处理函数 GDMA\_Channel\_AMIC\_Handler:

- 重新设置源端地址和目的端地址; 重新设置 GDMA 传输数据大小;
- 清除 GDMA 通道 0 传输完成中断挂起位, 使能 GDMA 通道 0。

```
void GDMA_Channel_AMIC_Handler(void)
{
```

```
GDMA_SetSourceAddress(GDMA_Channel_AMIC, (uint32_t)(&(I2S0->RX_DR)));
GDMA_SetDestinationAddress(GDMA_Channel_AMIC, (uint32_t)(&(I2S0->TX_DR)));

GDMA_SetBufferSize(GDMA_Channel_AMIC, AUDIO_FRAME_SIZE / 4);

GDMA_ClearINTPendingBit(GDMA_Channel_NUM, GDMA_INT_Transfer);
GDMA_Cmd(GDMA_Channel_NUM, ENABLE);

}
```

PDM 接口外接 PDM 播放器，播放录制的语音。

## 14 三线 SPI(SPI-3WIRE)

专用硬件设备：AN3205 模块

### 14.1 三线 SPI 通信

使用 SPI3WIRE 与 AN3205 进行数据传输。获取 AN3205 ID，判断 ID 是否正确，在 DebugAnalyser 工具上，打印相应信息。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\SPI3WIRE\Polling。

#### 14.1.1 硬件设计

硬件连接：P2\_2 -> SCLK， P2\_3 -> SDIO。

EVB 外接 AN3205 模块，连接 P2\_2 和 SCLK，P2\_3 和 SDIO，同时连接 VDD 和模块的 VDD，GND 和模块的 VSS。

#### 14.1.2 软件流程

引脚定义：

```
#define SPI_3WIRE_CLK_PIN      P2_2  
#define SPI_3WIRE_DATA_PIN     P2_3
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、无内部上拉、输出使能、输出高；

配置 PINMUX：分配引脚分别为 SPI2W\_CLK、SPI2W\_DATA 功能。

```
void board_3wire_spi_init(void)  
{  
    Pad_Config(SPI_3WIRE_CLK_PIN, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_NONE,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);  
    Pad_Config(SPI_3WIRE_DATA_PIN, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_NONE,  
    PAD_OUT_ENABLE, PAD_OUT_HIGH);  
    Pinmux_Config(SPI_3WIRE_CLK_PIN, SPI2W_CLK);  
    Pinmux_Config(SPI_3WIRE_DATA_PIN, SPI2W_DATA);  
}
```

使能 SPI3WIRE 时钟；

初始化 SPI3WIRE 外设：

- SPI3WIRE\_SysClock 设置为 20000000，即系统时钟为 20M；
- SPI3WIRE\_Speed 设置为 800000，即 SPI3WIRE 的输出时钟为 800K；

- SPI3WIRE\_Mode 设置为 SPI3WIRE\_2WIRE\_MODE, 即工作模式为两线 SPI;
- SPI3WIRE\_ReadDelay 设置为 0x3, 即读取数据时的延时时间为: (SPI3WIRE\_ReadDelay+1)/(2\*SPI3WIRE\_Speed) = 2.5us;
- SPI3WIRE\_OutputDelay 设置为 SPI3WIRE\_OE\_DELAY\_NONE, 即输出不产生延迟;
- SPI3WIRE\_ExtMode 设置为 SPI3WIRE\_NORMAL\_MODE, 即正常模式;

```
void driver_3wire_spi_init(void)
{
    RCC_PeriphClockCmd(APBPeriph_SPI2W, APBPeriph_SPI2W_CLOCK, ENABLE);
    SPI3WIRE_InitTypeDef SPI3WIRE_InitStruct;
    SPI3WIRE_StructInit(&SPI3WIRE_InitStruct);
    SPI3WIRE_InitStruct.SPI3WIRE_SysClock      = 20000000;
    SPI3WIRE_InitStruct.SPI3WIRE_Speed        = 800000;
    SPI3WIRE_InitStruct.SPI3WIRE_Mode         = SPI3WIRE_2WIRE_MODE;
    SPI3WIRE_InitStruct.SPI3WIRE_ReadDelay    = 0x3;
    SPI3WIRE_InitStruct.SPI3WIRE_OutputDelay  = SPI3WIRE_OE_DELAY_NONE;
    SPI3WIRE_InitStruct.SPI3WIRE_ExtMode       = SPI3WIRE_NORMAL_MODE;
    SPI3WIRE_Init(&SPI3WIRE_InitStruct);
}
```

设置 resync 信号的持续时间；使能 resync 信号输出；循环判断 Resync 信号正在输出标志位状态，SET 时继续判断；RESET 时失能 resync 信号输出；

使能 SPI3WIRE；

执行 mouse\_reset 函数，复位产品（对地址 06H 的 bit7 置 1）；

执行 mouse\_getproductid 函数，读产品 ID 到 id 数组，（地址 20H 的数据对应 ID 的 high byte[11:4]，地址 21H 的数据对应 ID 的 lower byte[0:3]）；

判断产品 ID 是否为 0x5820，正确则打印 pass 信息，否则打印 fail 信息。

```
void spi3wire_demo(void)
{
    /* Send resync time. Resync signal time = 2*1/(2*SPI3WIRE_Speed) = 1.25us */
    SPI3WIRE_SetResyncTime(2);
    SPI3WIRE_ResyncSignalCmd(ENABLE);
    while (SPI3WIRE_GetFlagStatus(SPI3WIRE_FLAG_RESYNC_BUSY) == SET);
    SPI3WIRE_ResyncSignalCmd(DISABLE);

    /* Enable SPI3WIRE to normal communication */
    SPI3WIRE_Cmd(ENABLE);
```

```
/* Reset mouse and Read mouse product ID */

mouse_reset();

mouse_getproductid(&id[0]);


if ((0x58 == id[0]) && (0x20 == (id[1] & 0xF0)))
{
    /* Notes: DBG_DIRECT is only used in debug demo, do not use in app project.*/
    DBG_DIRECT("SPI3WIRE test pass!");

}
else
{
    /* Notes: DBG_DIRECT is only used in debug demo, do not use in app project.*/
    DBG_DIRECT("SPI3WIRE test failed!");
}

}
```

在 DebugAnalyser 工具上，打印产品 ID 信息。

# 15 正交解调(QDEC)

专用硬件设备：滚轮编码器

## 15.1 相位检测

使用 QDEC，实现检测转动传感器的运动状态的功能。滚动 Mouse 滚轮，在 DebugAnalyser 工具上，打印滚轮滚动信息。

Q-decode 用于检测如图 15-1 转动传感装置的运动状态。转动装置运动时会输出两路正交信号 PHA 和 PHB，Q-decode 通过检测 PHA 和 PHB 的相位变化来判断转动的方向、速度。相位分为：00 相位，01 相位，11 相位，10 相位。

如图 15-2 所示，正常情况相位状态变化有 2 种：00->10->11->01 和 01->11->10->00，PHA 和 PHB 每次只应有一个信号变化，如果 2 个信号同时变化，则认为状态出错。

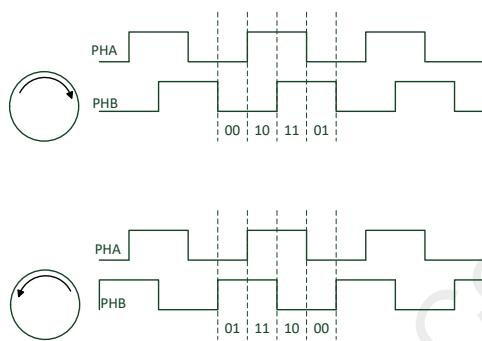


图 15-1 QDEC 连接图

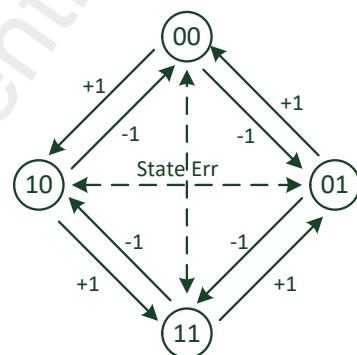


图 15-2 QDEC 相位转移图

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\QDEC\QDEC。

### 15.1.1 硬件设计

硬件连接：P2\_2 -> PhaseA, P2\_3 -> PhaseB。

EVB 外接 Mouse 模块，连接 P2\_2 和 PhaseA, P2\_3 和 PhaseB, VCC 和 VCC, GND 和 GND。

### 15.1.2 软件流程

引脚定义：

```
#define QDEC_Y_PHA_PIN          P2_2 /* phase A */
#define QDEC_Y_PHB_PIN          P2_3 /* phase B */
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、内部上拉、输出失能、输出低；

配置 PINMUX：分配引脚分别为 qdec\_phase\_a\_y、qdec\_phase\_b\_y 功能。

```
void board_qdec_init(void)
{
    Pad_Config(QDEC_Y_PHA_PIN,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_UP,
PAD_OUT_DISABLE, PAD_OUT_LOW);

    Pad_Config(QDEC_Y_PHB_PIN,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_UP,
PAD_OUT_DISABLE, PAD_OUT_LOW);

    Pinmux_Config(QDEC_Y_PHA_PIN, qdec_phase_a_y);
    Pinmux_Config(QDEC_Y_PHB_PIN, qdec_phase_b_y);
}
```

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_qdec\_init 函数，对 QDEC 外设进行初始化：

- axisConfigY 设置为 ENABLE，即使能 Y 轴功能；
  - debounceEnableY 设置为 Debounce\_Enable，即使能去抖动功能；
- 使能 QDEC Y 轴新数据产生中断 (QDEC\_Y\_INT\_NEW\_DATA)。

```
void driver_qdec_init(void)
{
    .....
    QDEC_InitTypeDef QDEC_InitStruct;
    QDEC_StructInit(&QDEC_InitStruct);
    QDEC_InitStruct.axisConfigY      = ENABLE;
    QDEC_InitStruct.debounceEnableY = Debounce_Enable;

    QDEC_INTConfig(QDEC, QDEC_Y_INT_NEW_DATA, ENABLE);
    .....
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 QDEC\_Cmd 函数，使能 QDECY 轴通道。

```
QDEC_Cmd(QDEC, QDEC_AXIS_Y, ENABLE);
```

当 QDEC Y 轴检测到新数据时，触发 QDEC\_Y\_INT\_NEW\_DATA 中断，进入中断处理函数 Qdecode\_Handler：

- 屏蔽 QDEC\_Y\_INT\_NEW\_DATA 中断；

- 读 Y 轴运动方向、计数值；
- 定义消息类型 IO\_MSG\_TYPE\_QDECODE，保存 Y 轴数据。发送 msg 到 task。
- 清除 QDEC\_Y\_INT\_NEW\_DATA 中断挂起位；取消屏蔽 QDEC\_Y\_INT\_NEW\_DATA 中断。

```
void Qdecode_Handler(void)
{
    if(QDEC_GetFlagState(QDEC, QDEC_FLAG_NEW_CT_STATUS_Y) == SET)
    {
        QDEC_INTMask(QDEC, QDEC_Y_CT_INT_MASK, ENABLE);
        Y_Axis_Data.AxisDirection = QDEC.GetAxisDirection(QDEC, QDEC_AXIS_Y);
        Y_Axis_Data.AxisCount = QDEC.GetAxisCount(QDEC, QDEC_AXIS_Y);

        T_IO_MSG int_qdec_msg;
        int_qdec_msg.type = IO_MSG_TYPE_QDECODE;
        int_qdec_msg.u.buf = (void *)&Y_Axis_Data;
        if(false == app_send_msg_to_apptask(&int_qdec_msg))
        .....
        QDEC_ClearINTPendingBit(QDEC, QDEC_CLR_NEW_CT_Y);
        QDEC_INTMask(QDEC, QDEC_Y_CT_INT_MASK, DISABLE);
    }
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_QDECODE，执行 io\_handle\_qdec\_msg：

- 打印 Y 轴数据，即 Mouse 滚轮滚动的数据。

```
void io_handle_qdec_msg(T_IO_MSG *io_qdec_msg)
{
    QDEC_Data_TypeDef *p_buf = io_qdec_msg->u.buf;
    APP_PRINT_INFO2("[io_qdec]io_handle_qdec_msg: Y_Axis_Direction = %d,Y_Axis_Count = %d",
                    p_buf->AxisDirection, p_buf->AxisCount);
}
```

滚动 Mouse 滚轮，在 DebugAnalyser 工具上，打印滚轮滚动信息。

# 16 8080 并行接口(IF8080)

专用硬件设备：2.4 寸 LCD 液晶模块（XSJ140HHAB2401）

点亮背灯需要子板 R15 接入一个  $0\Omega$  电阻。

## 16.1 自动 GDMA 刷屏模式

循环对整屏（320\*320）写红色、绿色。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\I8080\AutoMode。

### 16.1.1 硬件设计

硬件连接：P0\_4->D0, P0\_5->D1, P0\_6->D2, P0\_7->D3, P4\_0->D4, P4\_1->D5, P4\_2->D6, P4\_3->D7; P0\_0->CS, P1\_5->DCX, P1\_6->RD, P9\_2->WR, P2\_0->RST, P1\_2->BL。

EVB 外接 LCD 液晶模块。

### 16.1.2 软件流程

初始化写入屏幕数据。

```
void data_picture_init(void)
{
    .....
}
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、无内部上拉、输出使能、输出低；

配置 PINMUX：分配 D0-D7 等引脚为 IDLE\_MODE 功能；

配置 PAD、PINMUX，设置 LCD\_BL 为 PWM 输出；。

```
void board_lcd_init(void)
{
    Pad_Config(LCD_8080_D0,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

    Pad_Config(LCD_8080_D1,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

    Pad_Config(LCD_8080_D2,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

    Pad_Config(LCD_8080_D3,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);
```

```

Pad_Config(LCD_8080_D4,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

Pad_Config(LCD_8080_D5,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

Pad_Config(LCD_8080_D6,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

Pad_Config(LCD_8080_D7,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

Pad_Config(LCD_8080_CS,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

Pad_Config(LCD_8080_DCX,   PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

Pad_Config(LCD_8080_RD,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

Pad_Config(LCD_8080_WR,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

Pinmux_Config(LCD_8080_D0, IDLE_MODE); // D0
Pinmux_Config(LCD_8080_D1, IDLE_MODE); // D1
Pinmux_Config(LCD_8080_D2, IDLE_MODE); // D2
Pinmux_Config(LCD_8080_D3, IDLE_MODE); // D3
Pinmux_Config(LCD_8080_D4, IDLE_MODE); // D4
Pinmux_Config(LCD_8080_D5,, IDLE_MODE); // D5
Pinmux_Config(LCD_8080_D6,, IDLE_MODE); // D6
Pinmux_Config(LCD_8080_D7,, IDLE_MODE); // D7
Pinmux_Config(LCD_8080_CS, IDLE_MODE); // CS
Pinmux_Config(LCD_8080_DCX, IDLE_MODE); // DCX
Pinmux_Config(LCD_8080_RD, IDLE_MODE); // RD
Pinmux_Config(LCD_8080_WR, IDLE_MODE); // WR

Pad_Config(LCD_8080_BL,    PAD_PINMUX_MODE,    PAD_IS_PWRON,    PAD_PULL_NONE,
PAD_OUT_ENABLE, PAD_OUT_LOW);

Pinmux_Config(LCD_8080_BL, timer_pwm4);
}

```

使能 LCD 时钟；

配置 LCD 引脚组为 LCD\_PinGroup\_2，即 LCD 引脚定义如下： CS(P0\_0), DCX(P1\_5), WR(P0\_2),

RD(P1\_6), D0(P0\_4), D1(P0\_5), D2(P0\_6), D3(P0\_7), D4(P4\_0), D5(P4\_1), D6(P4\_2), D7(P4\_3);

初始化 TIM 外设:

初始化 TIMER 外设:

- TIM\_PWM\_En 设置为 PWM\_ENABLE, 即 PWM 模式;
- TIM\_Mode 设置为 TIM\_Mode\_UserDefine, 即用户定义模式;
- 设置 TIM\_PWM\_High\_Count 和 TIM\_PWM\_Low\_Count, 用户可以根据需求分配高电平周期值和低电平周期值;
- TIM\_SOURCE\_DIV 设置为 TIM\_CLOCK\_DIVIDER\_4, 即配置时钟分频大小;

使能 TIM2;

初始化 LCD 外设:

- LCD\_ClockDiv 设置为 LCD\_CLOCK\_DIV\_2, 即二分频;
- LCD\_Mode 设置为 LCD\_MODE\_MANUAL, 即手动模式;
- LCD\_GuardTimeCmd 设置为 LCD\_GUARD\_TIME\_DISABLE, 即关闭保护时间功能;
- LCD\_GuardTime 设置为 LCD\_GUARD\_TIME\_1T, 即 1 倍时钟时间;
- LCD\_8BitSwap 设置为 LCD\_8BitSwap\_ENABLE, 先发送低字节再发送高字节;
- LCD\_16BitSwap 设置为 LCD\_16BitSwap\_DISABLE, 即先发送高 16 位数据再发送低 16 位数据;
- LCD\_TxThr 设置为 10, 即触发 LCD 中断阈值为 10;
- LCD\_TxDMA\_Cmd 设置为 LCD\_TX\_DMA\_ENABLE, 即使能 GDMA 功能与内部 FIFO 控制;
- IF8080\_VsyncCmd 设置为 IF8080\_VSYNC\_DISABLE, 即失能 VSYNC 传输。

```
void driver_lcd_init(void)
{
    RCC_PeriphClockCmd(APBPeriph_TIMER, APBPeriph_TIMER_CLOCK, ENABLE);
    IF8080_PinGroupConfig(IF8080_PinGroup_2);

    TIM_TimeBaseInitTypeDef TIM_InitStruct;
    TIM_StructInit(&TIM_InitStruct);
    TIM_InitStruct.TIM_PWM_En = PWM_ENABLE;
    TIM_InitStruct.TIM_Period = 10 - 1 ;
    TIM_InitStruct.TIM_PWM_High_Count = 900 - 1 ;
    TIM_InitStruct.TIM_PWM_Low_Count = 100 - 1 ;
    TIM_InitStruct.TIM_Mode = TIM_Mode_UserDefine;
    TIM_InitStruct.TIM_SOURCE_DIV = TIM_CLOCK_DIVIDER_4;
    TIM_TimeBaseInit(BL_PWM_TIM, &TIM_InitStruct);
    TIM_Cmd(BL_PWM_TIM, ENABLE);
}
```

```
RCC_PeriphClockCmd(APBPeriph_IF8080, APBPeriph_IF8080_CLOCK, DISABLE);
RCC_PeriphClockCmd(APBPeriph_IF8080, APBPeriph_IF8080_CLOCK, ENABLE);

IF8080_PinGroupConfig(IF8080_PinGroup_2);
IF8080_InitTypeDef IF8080_InitStruct;
IF8080_StructInit(&IF8080_InitStruct);

IF8080_InitStruct.LCD_ClockDiv      = IF8080_CLOCK_DIV_2;
IF8080_InitStruct.LCD_Mode         = IF8080_MODE_MANUAL;
IF8080_InitStruct.LCD_GuardTimeCmd = IF8080_GUARD_TIME_DISABLE;
IF8080_InitStruct.LCD_GuardTime    = IF8080_GUARD_TIME_1T;
IF8080_InitStruct.LCD_8BitSwap     = IF8080_8BitSwap_ENABLE;
IF8080_InitStruct.LCD_16BitSwap    = IF8080_16BitSwap_DISABLE;
IF8080_InitStruct.LCD_TxThr        = 10;
IF8080_InitStruct.LCD_TxDMACmd    = IF8080_TX_DMA_DISABLE;
IF8080_InitStruct.IF8080_TxDMACmd = IF8080_TX_DMA_ENABLE;
IF8080_InitStruct.IF8080_VsyncCmd  = IF8080_VSYNC_DISABLE;
IF8080_InitStruct.IF8080_VsyncPolarity = IF8080_VSYNC_POLARITY_FALLING;
IF8080_Init(&LCD_InitStruct);

}
```

初始化 LCD\_RST 引脚：

- 复位 LCD\_8080\_RST 引脚： PINMUX 模式、PowerOn、无内部上拉、输出使能、输出低；
- 延迟 50ms；
- 置位 LCD\_8080\_RST 引脚： PINMUX 模式、PowerOn、无内部上拉、输出使能、输出高；
- 延迟 50ms。

```
void lcd_reset_init(void)
{
    lcd_set_reset(true);
    platform_delay_ms(50);
    lcd_set_reset(false);
    platform_delay_ms(50);
}
```

初始化 LCD 屏幕。

```
void lcd_st7796_init (void)
{
    st7796_write_cmd(0x11);
```

```
platform_delay_ms(120);
st7796_write_cmd(0x36);
st7796_write_data(0x48);
.....
st7796_write_cmd(0x21);
}
```

开启 LCD 背光。

```
void lcd_st7796_power_on(void)
{
    st7796_write_cmd(0x11);
    st7796_write_cmd(0x29);
    lcd_set_backlight(100);
}
```

用 GDMA 搬运数据的方式循环对屏幕区域（320\*320）写红色，绿色：

- 设置 IF8080 为手动模式；
- 设置 LCD 屏尺寸参数；
- 设置 IF8080 为自动模式；
- 初始化 GDMA 外设，设置搬运数据为 IF8080\_Color\_Group1，即对屏幕区域写红色；
- 开启自动模式写入数据；
- 当 GDMA 中断标志位置位后，跳出循环。

```
void lcd_auto_write_by_gdma_demo (void)
{
    .....
    while (1)
    {
        /* Enable Manual mode */
        IF8080_SwitchMode(IF8080_MODE_MANUAL);
        WriteBlock(0, X_SIZE - 1, 0, Y_SIZE - 1);
        /* Auto mode operation */
        IF8080_SwitchMode(IF8080_MODE_AUTO);
        driver_gdma_init((uint32_t)(IF8080_Color_Group1), (uint32_t)(IF8080_Color_Group1));
        lcd_auto_write(0x2C, (uint32_t)(PICTURE_FRAME_SIZE));
        while (GDMA_GetTransferINTStatus(IF8080_GDMA_Channel_NUM) == RESET)
        {
            __nop();
        }
    }
}
```

```

.....
    __nop();
}
}
}
```

其中 `driver_gdma_init` 初始化 GDMA 外设：

- 设置为 GDMA 通道 0；
- 设置传输方向为 memory 到外设传输，源端地址为 `IF8080_LLI_REG1_GDMA_BASE`，目的端地址为 IF8080 的 FIFO；
- 使能 Multi-block 传输：设置 `LLI_TRANSFER` 模式，设置传输 LLI 类型结构体首地址；
- 配置 block 传输后 LLI 结构体中源地址、目的地址、链表指针、控制寄存器；
- 使能 GDMA 总传输完成中断（`GDMA_INT_Transfer`）。

```

void driver_gdma_init(uint32_t addr)
{
    .....
    GDMA_InitStruct.GDMA_ChannelNum      = IF8080_GDMA_Channel_NUM;
    GDMA_InitStruct.GDMA_DIR             = GDMA_DIR_MemoryToPeripheral;
    GDMA_InitStruct.GDMA_SourceAddr     = IF8080_LLI_REG1_GDMA_BASE;;
    GDMA_InitStruct.GDMA_DestinationAddr = (uint32_t)(&(IF8080->FIFO));
    GDMA_InitStruct.GDMA_Multi_Block_En   = 1;
    GDMA_InitStruct.GDMA_Multi_Block_Mode = LLI_TRANSFER;
    GDMA_InitStruct.GDMA_Multi_Block_Struct = (uint32_t)IF8080_LLI_REG1_GDMA_BASE;
    .....
    GDMA_Config_LLIConfig(g1_addr, g2_addr, &GDMA_InitStruct);
    GDMA_INTConfig(IF8080_GDMA_Channel_NUM, GDMA_INT_Transfer, ENABLE);
    .....
}
```

当 GDMA 搬运数据完成后，触发 GDMA 中断，进入中断处理函数 `GDMA0_Channel0_Handler`：

- 清除 GDMA 通道 0 `GDMA_INT_Transfer` 中断挂起位；
- 打印 GDMA 搬运数据完成信息；
- 延迟 500ms；
- 判断 `GDMA_FLAG` 值是否为 0，若其值为 0，用 GDMA 搬运 `IF8080_Color_Group2` 数据，置位 `GDMA_FLAG`，对屏幕区域写入绿色；
- 若 `GDMA_FLAG` 值为 1，用 GDMA 搬运 `IF8080_Color_Group1` 数据，复位 `GDMA_FLAG`，对屏幕区域写入红色；

- 开启 LCD 自动写入功能。
- 使能 GDMA 总传输完成中断 (GDMA\_INT\_Transfer)。

```
void GDMA0_Channel0_Handler (void)
{
    GDMA_ClearINTPendingBit(IF8080_GDMA_Channel_NUM, GDMA_INT_Transfer);
    DBG_DIRECT("GDMA0_Channel0_Handler!");

    delay(500);

    GDMA_ClearAllTypeINT(IF8080_GDMA_Channel_NUM);
    IF8080_SwitchMode(IF8080_MODE_MANUAL);
    WriteBlock(0, X_SIZE - 1, 0, Y_SIZE - 1);
    /* Auto mode operation */
    IF8080_SwitchMode(IF8080_MODE_AUTO);
    if(GDMA_FLAG ==0)
    {
        driver_gdma_init((uint32_t)(IF8080_Color_Group2), (uint32_t)(IF8080_Color_Group2));
        GDMA_FLAG =1;
    }
    else
    {
        driver_gdma_init((uint32_t)(IF8080_Color_Group1), (uint32_t)(IF8080_Color_Group1));
        GDMA_FLAG =0;
    }
    lcd_auto_write(0x2C, (uint32_t)(PICTURE_FRAME_SIZE));}
```

LCD 液晶屏滚动显示红色、绿色。

# 17 实时时钟(RTC)

## 17.1 滴答定时器

使用 RTC 实现 TICK 功能，在 DebugAnalyser 工具上，定时打印信息。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ RTC\ Tick。

### 17.1.1 软件流程

RTC 外设初始化：

- 复位 RTC 外设；
- RTC 的分频系数设置为 (3200-1)，RTC 时钟频率为 10Hz (32K/3200)；
- 取消屏蔽 RTC 滴答中断；使能 RTC 滴答中断；复位 RTC 计数值；启动 RTC 外设。

```
void driver_rtc_init(void)
{
    RTC_DeInit();
    RTC_SetPrescaler(RTC_PRESCALER_VALUE);

    RTC_MaskINTConfig(RTC_INT_TICK, DISABLE);
    RTC_INTConfig(RTC_INT_TICK, ENABLE);
    .....
    RTC_ResetCounter();
    RTC_Cmd(ENABLE);
}
```

设定时间到，触发中断，进入中断处理函数 RTC\_Handler：

判断 RTC 滴答中断状态为 SET 则：

- 打印信息；
- 清除滴答中断。

```
void RTC_Handler(void)
{
    if (RTC_GetINTStatus(RTC_INT_TICK) == SET)
    {
        DBG_DIRECT("[main]RTC_Handler: RTC_INT_TICK");
        RTC_ClearTickINT();
    }
}
```

{

在 DebugAnalyser 工具上，每 0.1s 打印信息。

## 17.2 计数溢出

使用 RTC 实现 Overflow 功能，RTC 计数器溢出，在 DebugAnalyser 工具上，打印信息。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ RTC\ Overflow。

### 17.2.1 软件流程

RTC 外设初始化：

- 复位 RTC 外设；
- RTC 的分频系数设置为 (0)，RTC 时钟频率为 1/32000Hz；
- 取消屏蔽 RTC 计数器溢出中断；复位 RTC 计数值；启动 RTC 外设。

```
void driver_rtc_init(void)
{
    RTC_DeInit();
    RTC_SetPrescaler(RTC_PRESCALER_VALUE);

    RTC_MaskINTConfig(RTC_INT_OVF, DISABLE);
    .....
    RTC_ResetCounter();
    RTC_Cmd(ENABLE);
}
```

当计数器溢出时（计数大于  $2^{24}=16777216$ ， $16777216/32000 \approx 524s$ ），触发中断，进入中断处理函数 RTC\_Handler：

判断 RTC 计数器溢出中断状态为 SET 则：

- 打印信息；
- 并清除计数器溢出中断。

```
void RTC_Handler(void)
{
    if (RTC_GetINTStatus(RTC_INT_OVF) == SET)
    {
        DBG_DIRECT("[main]RTC_Handler: RTC_INT_OVF");
        RTC_ClearOverFlowINT();
    }
}
```

{

在 DebugAnalyser 工具上，每 524s 打印信息。

## 17.3 定时比较器（闹钟）

使用 RTC 实现 Comparator 功能，RTC 计数到，在 DebugAnalyser 工具上，打印信息。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ RTC\ Comparator。

### 17.3.1 软件流程

RTC 外设初始化：

- 复位 RTC 外设；
- 设置 RTC 的分频系数，RTC 时钟频率为 10Hz；
- 设置比较器通道 1 的比较值为 10；
- 取消屏蔽 RTC 比较器通道 1 计数中断；使能 RTC 比较器通道 1 计数中断；复位 RTC 计数值；启动 RTC 外设。

```
void driver_rtc_init(void)
{
    RTC_DeInit();
    RTC_SetPrescaler(RTC_PRESCALER_VALUE);
    RTC_SetComp(RTC_COMP_INDEX, RTC_COMP_VALUE);

    RTC_MaskINTConfig(RTC_COMP_INDEX_INT, DISABLE);
    RTC_INTConfig(RTC_COMP_INDEX_INT, ENABLE);
    .....
    RTC_ResetCounter();
    RTC_Cmd(ENABLE);
}
```

比较器设定时间（0.1s×10）到，触发中断，进入中断处理函数 RTC\_Handler。

判断 RTC 比较器通道 1 计数中断状态为 SET 则：

- 打印信息；
- 重新设置比较值；
- 清除比较器通道 1 计数中断；

```
void RTC_Handler(void)
{
    if (RTC_GetINTStatus(RTC_COMP_INDEX_INT) == SET)
```

```
{  
    DBG_DIRECT("[main]RTC_Handler: RTC_INT_CMP_NUM");  
    DBG_DIRECT("[main]RTC_Handler: RTC counter current value = %d", RTC_GetCounter());  
    RTC_SetCompValue(RTC_COMP_INDEX, RTC_GetCounter() + RTC_COMP_VALUE);  
    RTC_ClearCompINT(RTC_COMP_INDEX);  
}  
}
```

在 DebugAnalyser 工具上，每 1s 时，打印信息。

## 17.4 分频比较器

使用 RTC 实现 Rescaler + Comparator 功能，RTC 分频比较值以及 RTC 计数都满足中断条件时，在 DebugAnalyser 工具上，打印信息。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ RTC\ Pre\_comp3。

### 17.4.1 软件流程

1. 如果开启 GDMA\_INT\_Block 中断：

RTC 外设初始化：

- 复位 RTC 外设；
- 设置 RTC 的分频系数，RTC 时钟频率为 10Hz；
- 设置 RTC 的分频比较值为 320 (RTC 分频比较值<=分频系数)；
- 设置比较器通道 3 的比较值为 10；
- 取消屏蔽 RTC 分频比较器通道 3 中断；使能 RTC 分频比较器通道 3 中断；复位 RTC 计数值；启动 RTC 外设。

```
void driver_rtc_init(void)  
{  
    RTC_DeInit();  
    RTC_SetPrescaler(RTC_PRESCALER_VALUE);  
    RTC_SetPreCompValue(RTC_PRECOMP_VALUE);  
    RTC_SetCompValue(RTC_COMP3, RTC_COMP3_VALUE);  
  
    RTC_MaskINTConfig(RTC_INT_PRE_COMP3, DISABLE);  
    RTC_INTConfig(RTC_INT_PRE_COMP3, ENABLE);  
    .....  
    RTC_ResetCounter();
```

```
    RTC_Cmd(ENABLE);  
}
```

RTC 分频比较器值到达过设定值,且比较器设定时间 ( $0.1s \times 10$ ) 到, 触发中断, 进入中断处理函数 **RTC\_Handler**:

判断 RTC 分频比较器通道 3 中断状态为 SET 则:

- 打印信息;
- 重新设置比较值;
- 清除分频比较器通道 3 计数中断。

```
void RTC_Handler(void)  
{  
    if (RTC_GetINTStatus(RTC_INT_PRE_COMP3) == SET)  
    {  
        DBG_DIRECT("RTC_INT_PRE_COMP3");  
        RTC_SetCompValue(RTC_COMP3, counter + RTC_COMP3_VALUE);  
        RTC_ClearCompINT(RTC_INT_PRE_COMP3);  
    }  
}
```

2. 如果未开启 GDMA\_INT\_Block 中断:

RTC 外设初始化:

- 复位 RTC 外设;
- 设置 RTC 的分频系数, RTC 时钟频率为 10Hz;
- 设置 RTC 的分频比较值为 320 (RTC 分频比较值<=分频系数);
- 取消屏蔽 RTC 分频比较器中断; 使能 RTC 分频比较器中断; 复位 RTC 计数值; 启动 RTC 外设。

```
void driver_rtc_init(void)  
{  
    RTC_DeInit();  
    RTC_SetPrescaler(RTC_PRESCALER_VALUE);  
    RTC_SetPreCompValue(RTC_PRECOMP_VALUE);  
  
    RTC_MaskINTConfig(RTC_INT_PRE_COMP, DISABLE);  
    RTC_INTConfig(RTC_INT_PRE_COMP, ENABLE);  
    .....  
    RTC_ResetCounter();  
    RTC_Cmd(ENABLE);  
}
```

RTC 分频比较器值到达过设定值,且设定时间(0.1s)到,触发中断,进入中断处理函数 RTC\_Handler:  
判断 RTC 分频比较器中断状态为 SET 则:

- 打印信息;
- 清除分频比较器中断。

```
void RTC_Handler(void)
{
    if (RTC_GetINTStatus(RTC_INT_PRE_COMP) == SET)
    {
        DBG_DIRECT("RTC_INT_PRE_COMP");
        RTC_ClearCompINT(RTC_INT_PRE_COMP);
    }
}
```

在 DebugAnalyser 工具上, 设定时间到, 打印中断信息。

## 18 低功耗比较器(LPC)

专用硬件设备：直流稳压源

### 18.1 电压检测

验证 LPC 电压检测功能，P2\_5（可用于 LPC 电压检测的引脚：P2\_X、Vbat，其中 X 为 0~7）检测输入电压，当电压低于设定阈值（阈值范围：80mV~3200mV）时触发 LPC 中断，在 DebugAnalyser 工具上，打印信息。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\LPC\VoltageDetection。

#### 18.1.1 硬件设计

硬件连接：P2\_5 -> 直流稳压源。

P2\_5 外接直流稳压源，调节直流稳压源的电压值，P2\_5 检测电压。

#### 18.1.2 软件流程

引脚定义：

```
#define LPC_CAPTURE_PIN P2_5
```

配置 PAD：设置引脚、PINMUX 模式、PowerOn、无内部上拉、输出失能、输出高；

```
void board_lpc_init(void)
{
    Pad_Config(LPC_CAPTURE_PIN, PAD_PINMUX_MODE, PAD_IS_PWRON, PAD_PULL_NONE,
    PAD_OUT_DISABLE, PAD_OUT_HIGH);
    Pinmux_Config(LPC_CAPTURE_PIN, IDLE_MODE);
}
```

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_lpc\_init 函数，对 LPC 外设进行初始化：

- LPC\_Channel 设置为 LPC\_CAPTURE\_CHANNEL，即 P2\_5 引脚；
- LPC\_Edge 设置为 LPC\_VOLTAGE\_DETECT\_EDGE，即 LPC 比较器输出极性为低于设置的电压阈值 (LPC\_Vin\_Below\_Vth)；
- LPC\_Threshold 设置为 LPC\_VOLTAGE\_DETECT\_THRESHOLD，即电压阈值为 2000mV；使能 LPC 功能。

```
void driver_lpc_init(void)
{
    LPC_InitTypeDef LPC_InitStruct;
    LPC_StructInit(&LPC_InitStruct);
    LPC_InitStruct.LPC_Channel = LPC_CAPTURE_CHANNEL;
    LPC_InitStruct.LPC_Edge = LPC_VOLTAGE_DETECT_EDGE;
    LPC_InitStruct.LPC_Threshold = LPC_VOLTAGE_DETECT_THRESHOLD;
    LPC_Init(&LPC_InitStruct);
    LPC_Cmd(ENABLE);
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 nvic\_lpc\_config 函数：

- 使能电压符合 LPC 比较器输出极性中断 (LPC\_INT\_LPCOMP\_NV)。

```
void nvic_lpc_config(void)
{
    .....
    /* Enable voltage detection interrupt.If Vin<Vth, cause this interrupt */
    LPC_INTConfig(LPC_INT_LPCOMP_NV, ENABLE);
}
```

调节直流稳压源电压值，当 P2\_5 检测电压低于 2000mV 时，触发 LPC\_INT\_VOLTAGE\_COMP 中断，

进入中断处理函数 LPCOMP\_Handler:

- 失能 LPC\_INT\_LPCOMP\_NV 中断；
- 定义消息类型 IO\_MSG\_TYPE\_BAT\_LPC，发送 msg 给 task。

```
void LPCOMP_Handler(void)
{
    LPC_INTConfig(LPC_INT_VOLTAGE_COMP, DISABLE);
    T_IO_MSG int_lpc_msg;
    int_lpc_msg.type = IO_MSG_TYPE_BAT_LPC;
    if (false == app_send_msg_to_apptask(&int_lpc_msg))
    .....
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg)函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_BAT\_LPC：打印信息。

```
void app_handle_io_msg(T_IO_MSG io_msg)
```

```
{  
    .....  
  
    case IO_MSG_TYPE_BAT_LPC:  
        APP_PRINT_INFO0("[app] app_handle_io_msg: LPC low voltage detection msg.");  
}
```

调节直流稳压源，当电压低于 2000mV 时，在 DebugAnalyser 工具上，打印信息。

## 18.2 电压比较计数器

验证 LPC 比较器功能，P2\_4 检测输入电压，当检测电压高于阈值发生 10 次（计数器范围：0x0~0xffff）时，触发 RTC 中断，在 DebugAnalyser 工具上，打印信息。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ LPC\ Comparator。

### 18.2.1 硬件设计

硬件连接：P2\_4 -> 直流稳压源。

P2\_4 外接直流稳压源，调节直流稳压源的电压值，P2\_4 检测电压。

### 18.2.2 软件流程

配置 PAD、PINMUX，配置 P2\_4，用于检测模拟量；

初始化 LPC 外设：

- 设置 LPC\_Edge，LPC 比较器输出极性为高于设置的电压阈值（LPC\_Vin\_Over\_Vth）；
- 设置 LPC\_Threshold，LPC 比较器的电压阈值为 2400mV；

复位 LPC 计数器；

设置 LPC 比较器值为 LPC\_COMP\_VALUE（10），即 LPC 比较器计数 10 次产生中断；

使能 LPC 计数器；

使能 LPC 计数器计数等于比较器数据的中断（LPC\_INT\_COUNT\_COMP）。

```
void driver_lpc_init(void)  
{  
    .....  
  
    LPC_InitStruct.LPC_Edge      = LPC_VOLTAGE_DETECT_EDGE;  
    LPC_InitStruct.LPC_Threshold = LPC_VOLTAGE_DETECT_THRESHOLD;  
    .....  
  
    LPC_ResetCounter();  
    LPC_SetCompValue(LPC_COMP_VALUE);  
    LPC_CounterCmd(ENABLE);
```

```
LPC_INTConfig(LPC_INT_LPCOMP_CNT, ENABLE);  
.....  
}
```

当 P2\_4 检测电压高于 2400mV 发生 10 次时，触发 LPC\_INT\_LPCOMP\_CNT 中断，进入中断处理函数 RTC\_Handler：

- 打印信息；
- 重新设置比较值；
- 清除 LPC\_INT\_COUNT\_COMP 中断挂起位。

```
void RTC_Handler(void)  
{  
    /* LPC counter comparator interrupt */  
    if (LPC_GetINTStatus(LPC_INT_LPCOMP_CNT) == SET)  
    {  
        /* Notes: DBG_DIRECT is only used in debug demo, do not use in app project. */  
        DBG_DIRECT("LPC_INT_LPCOMP_CNT");  
        LPC_WriteComparator(LPC_ReadComparator() + LPC_COMP_VALUE);  
        LPC_ClearINTPendingBit(LPC_INT_LPCOMP_CNT);  
    }  
}
```

反复调节直流稳压源，当 P2\_4 检测电压高于 3200mV 发生 10 次时，在 DebugAnalyser 工具上，打印信息。

## 18.3 电压检测唤醒低功耗状态（DLPS）

验证 LPC 中断唤醒深度低功耗功能，P2\_5（可用于 LPC 电压检测的引脚：P2\_X、Vbat，其中 X 为 0~7）检测输入电压，当电压高于设定阈值时，系统自动进入 DLPS；当电压低于设定阈值（阈值范围：80mV~3200mV）时触发 LPC 中断，唤醒系统，退出 DLPS，在 DebugAnalyser 工具上，打印电压检测信息。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ LPC\ DLPS。

### 18.3.1 硬件设计

硬件连接：P2\_5 -> 直流稳压源。

P2\_5 外接直流稳压源，调节直流稳压源的电压值，P2\_5 检测电压。

### 18.3.2 软件流程

开启 DLPS 功能。

```
#define DLPS_EN           1  
#define USE_LPC_DLPS      1  
#define USE_USER_DEFINE_DLPS_EXIT_CB 1  
#define USE_USER_DEFINE_DLPS_ENTER_CB 1
```

初始化 LPC 全局数据。

```
void global_data_lpc_init(void)  
{  
    IO_LPC_DLPS_Enter_Allowed = true;  
}
```

配置 PAD、PINMUX 模式：设置 P2.5 为电压检测功能(详情参考：LPC-> VoltageDetection)。

执行 pwr\_mgr\_init 函数，初始化电源管理：

- 注册用户进入 DLPS 回调函数 app\_enter\_dlps\_config，注册用户退出 DLPS 回调函数 app\_exit\_dlps\_config；
- 注册硬件控制回调函数 DLPS\_IO\_EnterDlpsCb 和 DLPS\_IO\_ExitDlpsCb，进入 DLPS 会保存 CPU、PINMUX、Peripheral 等，退出 DLPS 会恢复 CPU、PINMUX、Peripheral 等；
- 设置 DLPS 模式。

```
void pwr_mgr_init(void)  
{  
#if DLPS_EN  
    if (false == dlps_check_cb_reg(app_dlps_check_cb))  
        .....  
    DLPS_IORegUserDlpsEnterCb(app_enter_dlps_config);  
    DLPS_IORegUserDlpsExitCb(app_exit_dlps_config);  
    DLPS_IORegister();  
    lps_mode_set(LPM_DLPS_MODE);  
    .....  
#endif  
}
```

在 app\_enter\_dlps\_config 中，执行 io\_lpc\_dlps\_enter 函数，打印进入 DLPS 信息。

```
void io_uart_dlps_enter(void)  
{  
    DBG_DIRECT("DLPS ENTER");
```

{

在 app\_exit\_dlps\_config 中，执行 io\_lpc\_dlps\_exit 函数，打印退出 DLPS 信息。

```
void io_uart_dlps_exit(void)
{
    DBG_DIRECT("DLPS EXIT");
}
```

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_lpc\_init 函数，对 LPC 外设进行初始化：

- LPC\_Channel 设置为 LPC\_CAPTURE\_CHANNEL，即 P2\_5 引脚；
- LPC\_Edge 设置为 LPC\_VOLTAGE\_DETECT\_EDGE，即 LPC 比较器输出极性为低于设置的电压阈值 (LPC\_Vin\_Below\_Vth)；
- LPC\_Threshold 设置为 LPC\_VOLTAGE\_DETECT\_THRESHOLD，即电压阈值为 2000mV；使能 LPC 功能。

```
void driver_lpc_init(void)
{
    LPC_InitTypeDef LPC_InitStruct;
    LPC_InitStructInit(&LPC_InitStruct);
    LPC_InitStruct.LPC_Channel = LPC_CAPTURE_CHANNEL;
    LPC_InitStruct.LPC_Edge = LPC_VOLTAGE_DETECT_EDGE;
    LPC_InitStruct.LPC_Threshold = LPC_VOLTAGE_DETECT_THRESHOLD;
    LPC_InitStruct(&LPC_InitStruct);    LPC_Cmd(ENABLE);
}
```

开始任务调度。

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 nvic\_lpc\_config 函数：

- 使能 LPC\_INT\_LPCOMP\_AON 中断；
- 使能电压符合 LPC 比较器输出极性中断 (LPC\_INT\_VOLTAGE\_COMP)；
- 使能 RTC 系统唤醒功能。

```
void nvic_lpc_config(void)
{
    .....
    /* Enable voltage detection interrupt.If Vin<Vth, cause this interrupt */
    LPC_INTConfig(LPC_INT_LPCOMP_AON, ENABLE);
    LPC_INTConfig(LPC_INT_LPCOMP_NV, ENABLE);
```

```
    RTC_SystemWakeupConfig(ENABLE);  
}
```

调节直流稳压源电压值，当 P2\_5 检测电压低于 2000mV 时，触发 LPC\_INT\_VOLTAGE\_COMP 中断，进入中断处理函数 LPCOMP\_Handler：

判断 LPC\_INT\_LPCOMP\_AON 状态为 SET 时：

- 清除 LPC\_INT\_LPCOMP\_AON 中断挂起位
  - 定义消息类型 IO\_MSG\_TYPE\_BAT\_LPC，发送 msg 给 task；
- 失能 LPC\_INT\_VOLTAGE\_COMP 中断。

```
void LPCOMP_Handler(void)  
{  
    if (LPC_GetINTStatus(LPC_INT_LPCOMP_AON) == SET)  
    {  
        LPC_ClearINTPendingBit(LPC_INT_LPCOMP_AON);  
        T_IO_MSG int_lpc_msg;  
        int_lpc_msg.type = IO_MSG_TYPE_BAT_LPC;  
        .....  
    }  
    LPC_INTConfig(LPC_INT_LPCOMP_NV, DISABLE);  
}
```

app\_main\_task 循环检测 msg queue。当有 msg 时，执行 app\_handle\_io\_msg(io\_msg) 函数。

在 app\_handle\_io\_msg 函数中，判断消息类型为 IO\_MSG\_TYPE\_BAT\_LPC，打印电压检测信息，执行 io\_handle\_uart\_msg 函数，执行 io\_uart\_handle\_msg：

判断类型为 IO\_MSG\_TYPE\_BAT\_LPC 时：

- 打印电压检测完成信息；
- 使能 LPC\_INT\_VOLTAGE\_COMP 中断
- IO\_LPC\_DLPS\_Enter\_Allowed 标志设置为 TRUE。

```
void io_lpc_handle_msg(T_IO_MSG *io_lpc_msg)  
{  
    .....  
    if (IO_MSG_TYPE_BAT_LPC == type)  
    {  
        APP_PRINT_INFO0("io_handle_lpc_msg: VoltageDetection done ");  
        LPC_INTConfig(LPC_INT_LPCOMP_NV, ENABLE);  
        IO_LPC_DLPS_Enter_Allowed = true;  
    }
```

```
    }
```

```
}
```

调节直流稳压源，当接入 P2-5 引脚的电压高于 2000mV，会自动进入 DLPS；当电压低于 2000mV 时，会退出 DLPS，并在 DebugAnalyser 工具上循环打印电压检测信息。

# 19 看门狗(WDG)

## 19.1 看门狗按键

验证 WDG 功能。看门狗定时 10s，使用按键（KEY0）触发中断进行喂狗，在 DebugAnalyser 工具上打印喂狗，重启看门狗信息。

工程目录：\HoneyComb\sdk\board\evb\_stack\_img\io\_sample\WDG\Reset。

### 19.1.1 软件流程

配置 PAD、PINMUX，设置 P4\_0（KEY0）为 GPIO 输入；初始化 GPIO 外设：使能 GPIO 中断模式，下降沿触发中断，使能 GPIO 中断去抖动，去抖时间为 10ms；取消屏蔽 GPIO 外部中断；使能 GPIO 中断。app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,  
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_wdg\_init 函数，对 WDG 外设进行初始化：

- 使能 WDG 时钟；
- 配置 WDG 定时器，定时时间为： $(2^{(11+1)-1} * ((77+1)/32000)) = 9.98s$ ，定时时间到未喂狗则复位全部（可根据需求，配置复位等级）；
- 使能 WDG 外设。

```
void driver_wdg_init(void)  
{  
    WDG_ClockEnable();  
    WDG_Config(77, 11, RESET_ALL);  
    WDG_Enable();  
}
```

开始任务调度。

```
os_sched_start();
```

看门狗定时 10s 到，仍没有按按键喂狗，则系统重启；

看门狗定时 10s 内，有按键按下，进入 GPIO 中断，发送 msg 给 task；task 检测到 msg，app 层解析 msg，执行 wdg\_feed 函数，实现喂狗功能。

GPIO 触发中断，发送 msg，检测 msg，解析 msg 的详情参考：GPIO -> Input key。

```
void wdg_feed(void)  
{  
    WDG_Restart();
```

{

看门狗定时 10s 到，仍没有按键按下时，系统重启；

看门狗定时 10s 内，有按键按下，喂狗。同时在 DebugAnalyser 工具上，打印喂狗，重启看门狗信息。

## 19.2 看门狗定时器

验证 WDG 功能。看门狗定时 10s，使用定时器 (TIMER) 定时 5s 触发中断进行喂狗，在 DebugAnalyser 工具上打印喂狗，重启看门狗信息，并重新设置定时器 (TIMER) 定时时间为 11s，门狗定时 10s 到，重启系统。

工程目录：\HoneyComb\ sdk\ board\ evb\_stack\_img\ io\_sample\ WDG\ Reset\_timer.

### 19.2.1 软件流程

app\_task 初始化。

```
os_task_create(&app_task_handle,"app",app_main_task,0,APP_TASK_STACK_SIZE,  
APP_TASK_PRIORITY);
```

在 app\_main\_task 中执行 driver\_init 函数，执行 driver\_timer\_init 和 driver\_wdg\_init 函数：

在 driver\_timer\_init 函数中对 TIM6 外设进行初始化：TIM\_Period 设置为((5000000)\*40-1)，即 5s 时间，并开启 TIM6 中断；

在 driver\_wdg\_init 函数中对 WDG 外设进行初始化：

- 使能 WDG 时钟；
- 配置 WDG 定时器，定时时间为： $(2^{(11+1)-1} * ((77+1)/32000)) = 9.98s$ ，定时时间到未喂狗则复位全部（可根据需求，配置复位等级）；
- 使能 WDG 外设。

```
void driver_wdg_init(void)  
{  
    WDG_ClockEnable();  
    WDG_Config(77, 11, RESET_ALL);  
    WDG_Enable();  
}
```

开始任务调度。

```
os_sched_start();
```

在 stack 准备好时，执行 app\_handle\_dev\_state\_evt 函数，执行 timer\_cmd(ENABLE) 函数，开启定时器。

定时器定时 5s，进入 TIM 中断，更改定时周期为 11s，发送 msg 给 task；task 检测到 msg，app 层解析 msg，执行 wdg\_feed 函数，实现喂狗功能。

TIM 触发中断，发送 msg，检测 msg，解析 msg 的详情参考：TIM -> Timer\_interrupt。

```
void wdg_feed(void)
{
    WDG_Restart();
}
```

定时器开启 5s 后进入 TIMER 中断，喂狗，更改定时器定时周期为 11s，同时在 DebugAnalyser 工具上，打印喂狗，重启看门狗信息，看门口定时 10s 到，未触发 TIMER 中断，重启系统。

## 参考文献

- [1] RTL8762D Evaluation Board User Manual CN\_V1.0
- [2] RTL8762D SDK User Guide CN
- [3] RTL8762D MP Tool User Guide CN
- [4] DebugAnalyzer User Guide
- [5] RTL8762C ADC 应用实例