

3. 18 陀螺仪读取原始数据

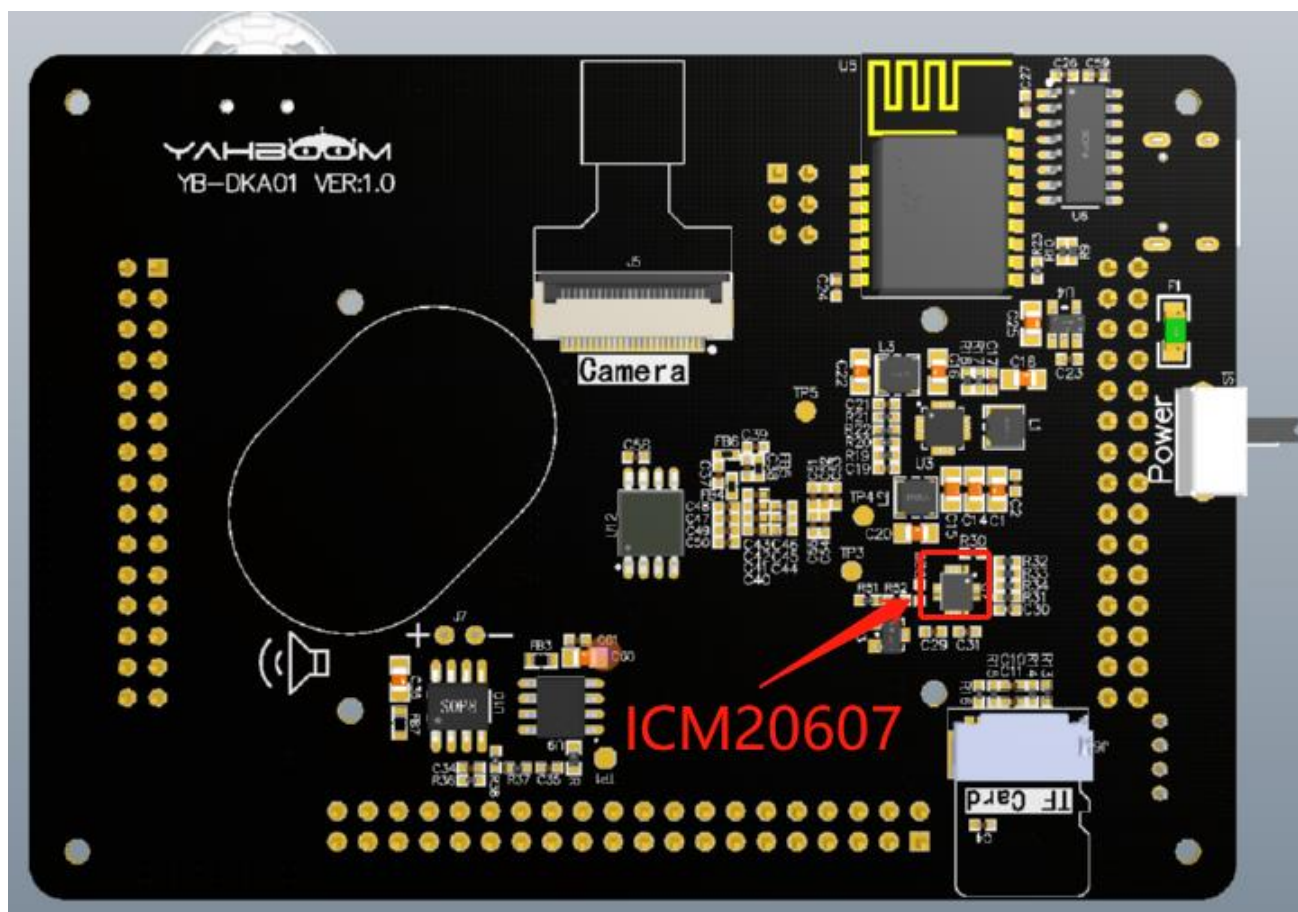
一、实验目的

本节课主要学习 K210 通过 I2C 读取 ICM20607 芯片的 X/Y/Z 轴原始数据。

二、实验准备

1. 实验元件

六轴姿态传感器 icm20607



2. 元件特性

ICM20607 是一个六轴运动跟踪设备，它结合了一个 3 轴陀螺仪和一个 3 轴加速度计，具有 1K 字节的 FIFO，可以降低串行总线接口上的流量，并通过允许系统处理器突发读写传感器数据，然后进入低功耗模式来降低功耗。ICM20607 的陀

螺仪可编程的范围为 ± 250 、 ± 500 、 ± 1000 和 ± 2000 度/秒，加速度计的全量程为 $\pm 2g$ 、 $\pm 4g$ 、 $\pm 8g$ 和 $\pm 16g$ 。ICM20607 包含芯片上 16 位 ADC，可编程数字滤波器，嵌入式温度传感器和可编程中断，支持 I2C 和 SPI 通讯，VDD 操作范围为 $1.71V \sim 3.45V$ 。

陀螺仪特性

三轴 MEMS 陀螺仪在 ICM-20607 包括以下特点：

- 数字输出 X、Y、Z 轴角速度传感器(陀螺仪)，用户可编程的全量程范围为 ± 250 、 ± 500 、 ± 1000 和 $\pm 2000^\circ$ /秒，集成 16 位 adc
- 数字可编程低通滤波器
- 低功耗陀螺仪操作
- 工厂标定的灵敏度标度因子
- 自测

加速度计的特性

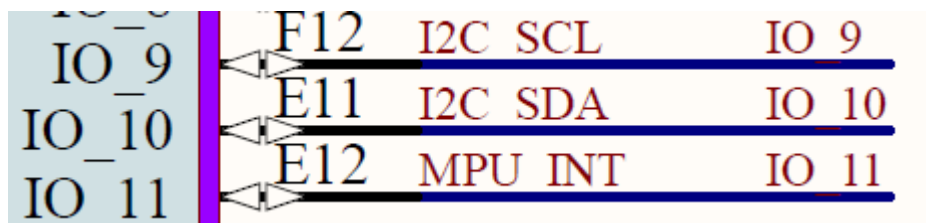
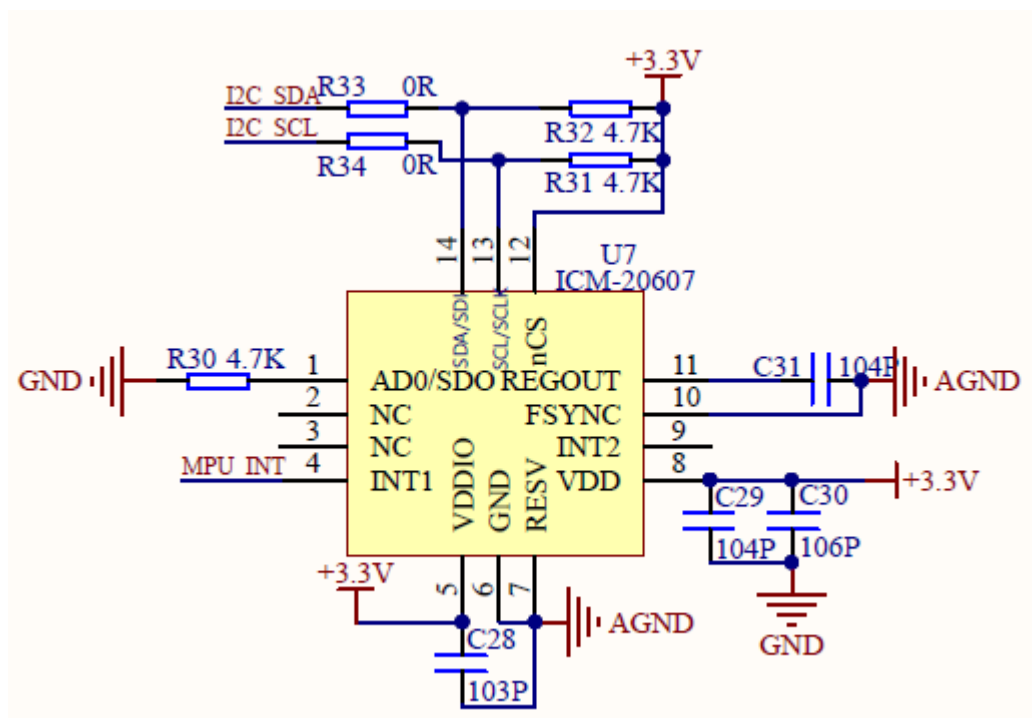
三轴 MEMS 加速度计在 ICM-20607 包括以下特点：

- 数字输出 X、Y、z 轴加速度计，可编程全量程范围为 $\pm 2g$ 、 $\pm 4g$ 、 $\pm 8g$ 、 $\pm 16g$ ，集成 16 位 adc
- 可编程中断
- Wake-on-motion
- 自测

3. 硬件连接

ICM20607 芯片使用 I2C 连接的方式，I2C_SCL 连接的是 IO9, I2C_SDA 连接的是

IO10, MPU_INT 连接的是 IO11。



4. SDK 中对应 API 功能

对应的头文件 i2c.h

I2C 总线用于和多个外部设备进行通信。多个外部设备可以共用一个 I2C 总线。

I2C 模块具有独立的 I2C 设备封装外设相关参数，自动处理多设备总线争用的功能。

K210 芯片集成电路总线有 3 个 I2C 总线接口，都可以作为 I2C 主机(MASTER)模式或从机 (SLAVE) 模式来使用。

I2C 接口支持标准模式 (0 到 100kb/s)，快速模式 (<=400kb/s)，7 位或

10 位寻址模式，批量传输模式，中断或轮询模式操作。

为用户提供以下接口：

- `i2c_init`: 初始化 I2C，配置从机地址、寄存器位宽度和 I2C 速率。
- `i2c_init_as_slave`: 配置 I²C 为从模式。
- `i2c_send_data`: I2C 写数据。
- `i2c_send_data_dma`: I2C 通过 DMA 写数据。
- `i2c_recv_data`: I2C 通过 CPU 读数据。
- `i2c_recv_data_dma`: I2C 通过 dma 读数据。
- `i2c_handle_data_dma`: I2C 使用 dma 传输数据。

三、实验原理

陀螺仪工作的原理：陀螺仪是基于角动量守恒的理论设计出来的一种用来传感与维持方向的装置。陀螺仪主要是由一个位于轴心且可旋转的转子构成，物体高速旋转时，角动量很大，旋转轴会一直稳定指向一个方向，所以陀螺仪的转动速度必须转的够快，不然会影响它的稳定性。

加速度计工作的原理：加速度计是一种惯性传感器，能够测量物体的加速力。

技术成熟的 MEMS 加速度计分为三种：压电式、容感式、热感式。压电式 MEMS 加速度计运用的是压电效应，在其内部有一个刚体支撑的质量块，有运动的情况下质量块会产生压力，刚体产生应变，把加速度转变成电信号输出。

容感式 MEMS 加速度计内部也存在一个质量块，从单个单元来看，它是标准的平板电容器。加速度的变化带动活动质量块的移动从而改变平板电容两极的间距和正对面积，通过测量电容变化量来计算加速度。

热感式 MEMS 加速度计内部没有任何质量块，它的中央有一个加热体，周边是温度传感器，里面是密闭的气腔，工作时在加热体的作用下，气体在内部形成一个热气团，热气团的比重和周围的冷气是有差异的，通过惯性热气团的移动形

成的热场变化让感应器感应到加速度值。

由于压电式 MEMS 加速度计内部有刚体支撑的存在,通常情况下,压电式 MEMS 加速度计只能感应到“动态”加速度,而不能感应到“静态”加速度,也就是我们所说的重力加速度。而容感式和热感式既能感应“动态”加速度,又能感应“静态”加速度。。

四、实验过程

1. 首先初始化 K210 的硬件引脚和软件功能使用的是 FPIOA 映射关系。

```
/******HARDWARE-PIN*****  
// 硬件Io口,与原理图对应  
  
#define PIN_ICM_SCL          (9)  
#define PIN_ICM_SDA          (10)  
#define PIN_ICM_INT          (11)  
  
/******SOICMWARE-GPIO*****  
// 软件GPIO口,与程序对应  
#define ICM_INT_GPIONUM      (2)  
  
/******FUNC-GPIO*****  
// GPIO口的功能,绑定到硬件Io口  
#define FUNC_ICM_INT         (FUNC_GPIOHS0 + ICM_INT_GPIONUM)  
#define FUNC_ICM_SCL         (FUNC_I2C0_SCLK)  
#define FUNC_ICM_SDA         (FUNC_I2C0_SDA)  
  
void hardware_init(void)  
{  
    /* I2C ICM20607 */  
    fpioa_set_function(PIN_ICM_SCL, FUNC_ICM_SCL);  
    fpioa_set_function(PIN_ICM_SDA, FUNC_ICM_SDA);  
}
```

2. 初始化 ICM20607 芯片,首先是先让设备复位,然后读取设备 ID 看看是否匹配,接下来是一系列的写入寄存器的操作,这里可以根据实际修改参数,详细的初始化流程及寄存器的作用可以查看硬件资料中的 icm20607 资料。

```

/* 初始化icm20607芯片 */
void icm20607_init(void)
{
    uint8_t val = 0x0, res = 1;
    i2c_hardware_init(ICM_ADDRESS); // 初始化
    msleep(10);

    icm_i2c_write(PWR_MGMT_1, 0x80); //复位设备
    msleep(100);
    icm20607_who_am_i();

    do
    { //等待复位成功
        icm_i2c_read(PWR_MGMT_1, &val, 1);
    } while(0x41 != val);

    icm_i2c_write(PWR_MGMT_1, 0x01); //时钟设置
    icm_i2c_write(PWR_MGMT_2, 0x00); //开启陀螺仪和加速度计
    icm_i2c_write(CONFIG, 0x01); //176HZ 1KHZ
    icm_i2c_write(SMPLRT_DIV, 0x07); //采样速率 SAMPLE_RATE = INTEN
    icm_i2c_write(GYRO_CONFIG, 0x18); //±2000 dps
    icm_i2c_write(ACCEL_CONFIG, 0x10); //±8g
    icm_i2c_write(ACCEL_CONFIG_2, 0x23); //Average 8 samples 44.8HZ
    return res;
}

```

3. 判断是否是 ICM20607 芯片，从 ICM20607 的 WHO_AM_I 寄存器读出值，然后对比是 ICM20607 的 ID 就行了。

```

/* 判断是否是icm20607芯片 */
void icm20607_who_am_i(void)
{
    uint8_t val, state = 1;
    do
    {
        icm_i2c_read(WHO_AM_I, &val, 1); // 读ICM20607的ID

        if (ICM20607_ID != val & state) // 当ID不对时，只报一次。
        {
            printf("WHO_AM_I=0x%02x\n", val);
            state = 0;
        }
    } while(ICM20607_ID != val);
    printf("WHO_AM_I=0x%02x\n", val);
}

```

4. 获取陀螺仪 X 轴的原始数据，直接从陀螺仪的 X 轴输出寄存器从读取两个数据，然后再按照高低位的方式把数据合成最终需要的数据。

```
/* 读取陀螺仪X轴原始数据 */
int16_t getRawGyroscopeX(void) {
    uint8_t val[2] = {0};
    icm_i2c_read(GYRO_XOUT_H, val, 2);
    return ((int16_t)val[0] << 8) + val[1];
}
```

5. 读取陀螺仪 Y 轴和 Z 轴的原始数据也是同样的步骤。

```
/* 读取陀螺仪Y轴原始数据 */
int16_t getRawGyroscopeY(void) {
    uint8_t val[2] = {0};
    icm_i2c_read(GYRO_YOUT_H, val, 2);
    return ((int16_t)val[0] << 8) + val[1];
}

/* 读取陀螺仪Z轴原始数据 */
int16_t getRawGyroscopeZ(void) {
    uint8_t val[2] = {0};
    icm_i2c_read(GYRO_ZOUT_H, val, 2);
    return ((int16_t)val[0] << 8) + val[1];
}
```

6. 利用同样的方法，也可以读取加速度计的 X/Y/Z 轴的原始数据。

```
/* 读取加速度计X轴原始数据 */
int16_t getRawAccelerationX(void) {
    uint8_t val[2] = {0};
    icm_i2c_read(ACCEL_XOUT_H, val, 2);
    return ((int16_t)val[0] << 8) + val[1];
}

/* 读取加速度计Y轴原始数据 */
int16_t getRawAccelerationY(void) {
    uint8_t val[2] = {0};
    icm_i2c_read(ACCEL_YOUT_H, val, 2);
    return ((int16_t)val[0] << 8) + val[1];
}

/* 读取加速度计Z轴原始数据 */
int16_t getRawAccelerationZ(void) {
    uint8_t val[2] = {0};
    icm_i2c_read(ACCEL_ZOUT_H, val, 2);
    return ((int16_t)val[0] << 8) + val[1];
}
```


7. 最后是把对应的数据打印出来，这里可以选择打印陀螺仪的数据还是加速度计的数据，默认是打印陀螺仪的数据，可以通过修改 GYRO_DATA 和 ACC_DATA 的值来改变显示的设备数据。

```
#define GYRO_DATA    1
#define ACC_DATA     0

while (1)
{
    #if GYRO_DATA
    val_gx = getRawGyroscopeX();
    val_gy = getRawGyroscopeY();
    val_gz = getRawGyroscopeZ();
    printf("gx=%d, gy=%d, gz=%d\n", val_gx, val_gy, val_gz);
    val_gx = val_gy = val_gz = 0;
    #elif ACC_DATA
    val_ax = getRawAccelerationX();
    val_ay = getRawAccelerationY();
    val_az = getRawAccelerationZ();
    printf("ax=%d, ay=%d, az=%d\n", val_ax, val_ay, val_az);
    val_ax = val_ay = val_az = 0;
    #else
    printf("Please set the GYRO_DATA or ACC_DATA to 1\n");
    #endif
    msleep(5);
}
```

8. 编译调试，烧录运行

把本课程资料中的 gyro 复制到 SDK 中的 src 目录下，然后进入 build 目录，运行以下命令编译。

```
cmake .. -DPROJ=gyro -G "MinGW Makefiles"
```

```
make
```

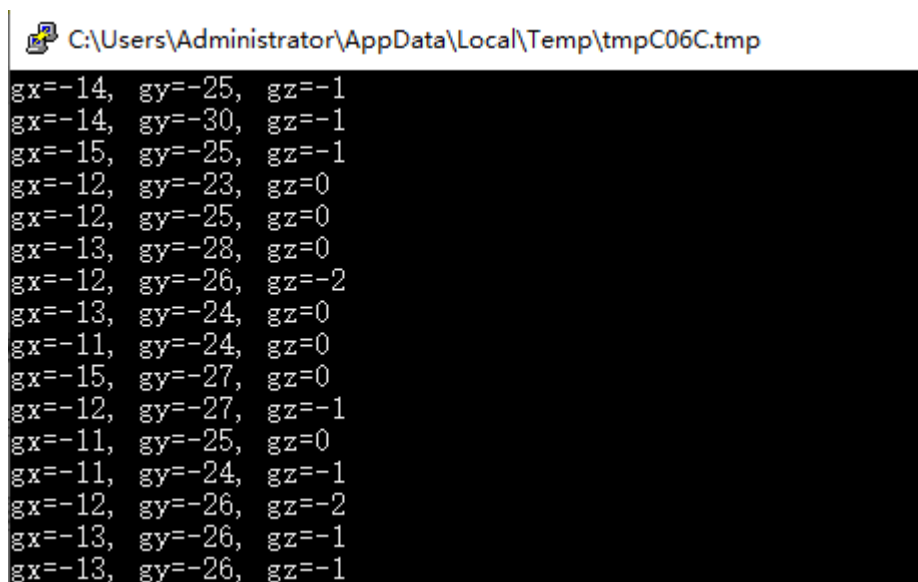
```
[ 93%] Building C object CMakeFiles/gyro.dir/src/gyro/main.c.obj
[ 95%] Linking C executable gyro
Generating .bin file ...
[100%] Built target gyro
PS C:\K210\SDK\kendryte-standalone-sdk-develop\build> █
```


编译完成后，在 build 文件夹下会生成 gyro.bin 文件。

使用 type-C 数据线连接电脑与 K210 开发板，打开 kflash，选择对应的设备，再将程序固件烧录到 K210 开发板上。

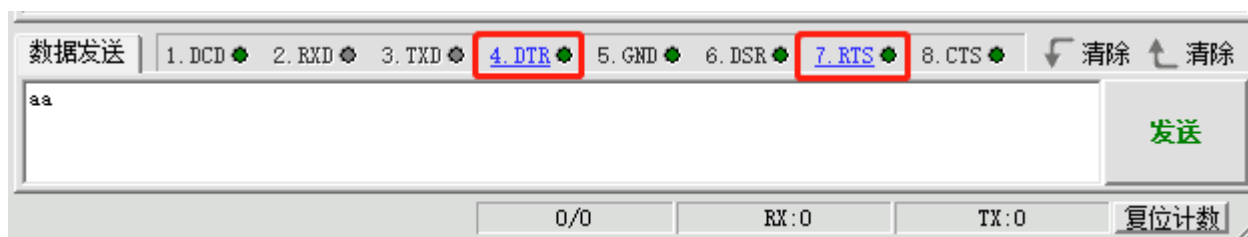
五、实验现象

烧录完成固件后，系统会弹出一个终端界面，如果没有弹出终端界面的可以打开串口助手显示调试内容。



```
C:\Users\Administrator\AppData\Local\Temp\tmpC06C.tmp
gx=-14, gy=-25, gz=-1
gx=-14, gy=-30, gz=-1
gx=-15, gy=-25, gz=-1
gx=-12, gy=-23, gz=0
gx=-12, gy=-25, gz=0
gx=-13, gy=-28, gz=0
gx=-12, gy=-26, gz=-2
gx=-13, gy=-24, gz=0
gx=-11, gy=-24, gz=0
gx=-15, gy=-27, gz=0
gx=-12, gy=-27, gz=-1
gx=-11, gy=-25, gz=0
gx=-11, gy=-24, gz=-1
gx=-12, gy=-26, gz=-2
gx=-13, gy=-26, gz=-1
gx=-13, gy=-26, gz=-1
```

打开电脑的串口助手，选择对应的 K210 开发板对应的串口号，波特率设置为 115200，然后点击打开串口助手。注意还需要设置一下串口助手的 DTR 和 RTS。在串口助手底部此时的 4. DTR 和 7. RTS 默认是红色的，点击 4. DTR 和 7. RTS，都设置为绿色，然后按一下 K210 开发板的复位键。



可以看到串口助手会显示当前的陀螺仪打印的数据，当我们上下左右摆动开

发板时，陀螺仪数据会跟着变化。



六、实验总结

1. ICM20607 芯片是一个六轴传感器芯片，里面包含一个三轴陀螺仪和一个三轴加速度计。
2. 陀螺仪和加速度计在使用前需要初始化，否则无法正常使用。
3. 本次 icm20607 使用的是 I2C 通讯的方式传输数据，也可以使用 SPI 传输数据的方式。

附：API 对应的头文件 i2c.h

i2c_init

描述

配置 I²C 器件从地址、寄存器位宽度和 I²C 速率。

函数原型

```
void i2c_init(i2c_device_number_t i2c_num, uint32_t slave_address, uint32_t address_width, uint32_t i2c_clk)
```

参数

参数名称	描述	输入输出
i2c_num	I ² C 号	输入
slave_address	I ² C 器件从地址	输入
address_width	I ² C 器件寄存器宽度(7 或 10)	输入
i2c_clk	I ² C 速率 (Hz)	输入

返回值

无。

i2c_init_as_slave

描述

配置 I²C 为从模式。

函数原型

```
void i2c_init_as_slave(i2c_device_number_t i2c_num, uint32_t slave_address, uint32_t address_width, const i2c_slave_handler_t *handler)
```

参数

参数名称	描述	输入输出
i2c_num	I ² C 号	输入
slave_address	I ² C 从模式的地址	输入
address_width	I ² C 器件寄存器宽度(7 或 10)	输入
handler	I ² C 从模式的中断处理函数	输入

返回值

无。

i2c_send_data

描述

写数据。

函数原型

```
int i2c_send_data(i2c_device_number_t i2c_num, const uint8_t *send_buf, size_t send_buf_len)
```

参数

参数名称	描述	输入输出
i2c_num	I ² C 号	输入
send_buf	待传输数据	输入
send_buf_len	待传输数据长度	输入

返回值

返回值 描述

0 成功

非 0 失败

i2c_send_data_dma

描述

通过 DMA 写数据。

函数原型

```
void i2c_send_data_dma(dmac_channel_number_t dma_channel_num,  
i2c_device_number_t i2c_num, const uint8_t *send_buf, size_t send_buf_len)
```

参数

参数名称	描述	输入输出
dma_channel_num	使用的 dma 通道号	输入
i2c_num	I ² C 号	输入

参数名称	描述	输入输出
send_buf	待传输数据	输入
send_buf_len	待传输数据长度	输入

返回值

无

i2c_recv_data

描述

通过 CPU 读数据。

函数原型

```
int i2c_recv_data(i2c_device_number_t i2c_num, const uint8_t *send_buf, size_t send_buf_len, uint8_t *receive_buf, size_t receive_buf_len)
```

参数

参数名称	描述	输入输出
i2c_num	I ² C 总线号	输入
send_buf	待传输数据，一般是 i2c 外设的寄存器，如果没有设置为 NULL	输入
send_buf_len	待传输数据长度，如果没有则写 0	输入
receive_buf	接收数据内存	输出
receive_buf_len	接收数据的长度	输入

返回值

返回值 描述

0 成功

非 0 失败

i2c_recv_data_dma

描述

通过 dma 读数据。

函数原型

```
void i2c_recv_data_dma(dmac_channel_number_t dma_send_channel_num, dmac_channel_number_t dma_receive_channel_num,
```

```
i2c_device_number_t i2c_num, const uint8_t *send_buf, size_t send_buf_len,  
uint8_t *receive_buf, size_t receive_buf_len)
```

参数

参数名称	描述	输入输出
dma_send_channel_num	发送数据使用的 dma 通道	输入
dma_receive_channel_num	接收数据使用的 dma 通道	输入
i2c_num	I ² C 总线号	输入
send_buf	待传输数据, 一般是 i2c 外设的寄存器, 如果没有设置为 NULL	输入
send_buf_len	待传输数据长度, 如果没有则写 0	输入
receive_buf	接收数据内存	输出
receive_buf_len	接收数据的长度	输入

返回值

无

i2c_handle_data_dma

描述

I²C 使用 dma 传输数据。

函数原型

```
void i2c_handle_data_dma(i2c_device_number_t i2c_num, i2c_data_t data,  
plic_interrupt_t *cb);
```

参数

参数名称	描述	输入输出
i2c_num	I ² C 总线号	输入
data	I ² C 数据相关的参数, 详见 i2c_data_t 说明	输入
cb	dma 中断回调函数, 如果设置为 NULL 则为阻塞模式, 直至传输完毕后退函数	输入

返回值

无

举例

```

/* i2c 外设地址是 0x32, 7 位地址, 速率 200K */
i2c_init(I2C_DEVICE_0, 0x32, 7, 200000);
uint8_t reg = 0;
uint8_t data_buf[2] = {0x00, 0x01}
data_buf[0] = reg;
/* 向 0 寄存器写 0x01 */
i2c_send_data(I2C_DEVICE_0, data_buf, 2);
i2c_send_data_dma(DMAC_CHANNEL0, I2C_DEVICE_0, data_buf, 4);
/* 从 0 寄存器读取 1 字节数据 */
i2c_receive_data(I2C_DEVICE_0, &reg, 1, data_buf, 1);
i2c_receive_data_dma(DMAC_CHANNEL0, DMAC_CHANNEL1, I2C_DEVICE_0, &reg, 1,
data_buf, 1);

```

数据类型

相关数据类型、数据结构定义如下：

- `i2c_device_number_t`: i2c 号。
- `i2c_slave_handler_t`: i2c 从模式的中断处理函数句柄
- `i2c_data_t`: 使用 dma 传输时数据相关的参数。
- `i2c_transfer_mode_t`: 使用 DMA 传输数据的模式，发送或接收。

`i2c_device_number_t`

描述

i2c 编号。

定义

```

typedef enum _i2c_device_number
{
    I2C_DEVICE_0,
    I2C_DEVICE_1,
    I2C_DEVICE_2,
    I2C_DEVICE_MAX,
} i2c_device_number_t;

```

`i2c_slave_handler_t`

描述

i2c 从模式的中断处理函数句柄。根据不同的中断状态执行相应的函数操作。

定义

```
typedef struct _i2c_slave_handler
{
    void(*on_receive)(uint32_t data);
    uint32_t(*on_transmit)();
    void(*on_event)(i2c_event_t event);
} i2c_slave_handler_t;
```

成员

成员名称	描述
I2C_DEVICE_0	I2C 0
I2C_DEVICE_1	I2C 1
I2C_DEVICE_2	I2C 2

i2c_data_t

描述

使用 dma 传输时数据相关的参数。

定义

```
typedef struct _i2c_data_t
{
    dmac_channel_number_t tx_channel;
    dmac_channel_number_t rx_channel;
    uint32_t *tx_buf;
    size_t tx_len;
    uint32_t *rx_buf;
    size_t rx_len;
    i2c_transfer_mode_t transfer_mode;
} i2c_data_t;
```

成员

成员名称	描述
tx_channel	发送时使用的 DMA 通道号
rx_channel	接收时使用的 DMA 通道号
tx_buf	发送的数据
tx_len	发送数据的长度
rx_buf	接收的数据

成员名称	描述
rx_len	接收数据长度
transfer_mode	传输模式，发送或接收

i2c_transfer_mode_t

描述

使用 DMA 传输数据的模式，发送或接收。

定义

```
typedef enum _i2c_transfer_mode
{
    I2C_SEND,
    I2C_RECEIVE,
} i2c_transfer_mode_t;
```

成员

成员名称	描述
I2C_SEND	发送
I2C_RECEIVE	接收