Machine Translated by Google

I2C Device Application Notes

RT-THREAD Document Center

Copyright @2019 Shanghai Ruisaide Electronic Technology Co., Ltd.

WWW.RT-THREAD.ORG

Friday 28th September, 2018

Machine Translated by Google

Table of contents

Machine Translated by Google

This application note uses the MPU6050 6-axis sensor that drives the I2C interface as an example to explain how to develop applications using the I2C device driver interface. It also explains the RT-Thread I2C device driver framework and related functions in detail.

1 Purpose and structure of this paper

1.1 Purpose and Background of this Paper

The I2C (or i2c, IIC, iic) bus is a simple, bidirectional two-wire (clock
The I2C synchronous serial bus (SCL, SDA) uses only two wires to transmit information between connected devices and is one of the most widely used communication interfaces in semiconductor chips. RT-Thread introduces the I2C device driver framework, which provides two low-level hardware interfaces: GPIO emulation and hardware controller.

1.2 Structure of this paper

This article first describes the basics of the RT-Thread I2C device driver framework, then describes the I2C device driver interface in detail, uses the I2C device driver interface to write a driver for the MPU6050, and provides a code example verified on the Zhengdian Atom STM32F4 Explorer development board.

2 Introduction to the I2C Device Driver Framework

When developing projects with an MCU, the I2C bus is often required. Generally, the MCU comes with an I2C controller (hardware I2C), but you can also use the MCU's two GPIOs to write your own program to simulate the I2C bus protocol and achieve the same functionality.

RT-Thread provides an I/O device management framework that divides I/O device management into three layers: the application layer, the I/O device management layer, and the underlying driver layer. The I/O device management framework provides upper-layer applications with a unified device operation interface and I2C device driver interface, while providing lower-layer applications with the underlying driver interface. Applications access underlying devices through standard interfaces provided by the I/O device module. Changes to the underlying device do not affect upper-layer applications. This approach ensures high application portability, allowing applications to be easily ported from one MCU to another.

This article takes the 6-axis inertial sensor MPU6050 as an example, uses the GPIO provided by the RT-Thread I2C device driver framework to simulate the I2C controller, and explains how applications use the I2C device driver interface to access I2C devices.

Machine Translated by Google

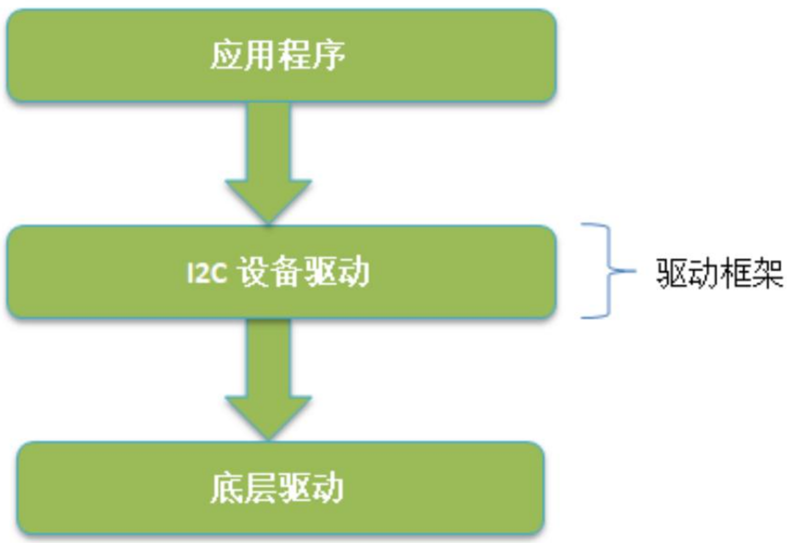

Figure 1: *RT-Thread I2C* Device driver framework

3. Run the I2C device driver sample code

3.1 Sample Code Software and Hardware Platform

1. Zhengdian Atom STM32F4 Explorer Development Board 2. GY-521 MPU-6050 module
3. MDK5
4. RT-Thread Source Code
5. I2C Example Code

The MCU of the Atom Explorer STM32F4 development board is STM32F407ZGT6. This example uses the USB serial port. The USART1 port sends data and supplies power, and SEGGER JLINK is used to connect to JTAG debugging.

The GY521 module used in this experiment is a 6-axis inertial sensor module with an onboard MPU6050. We used the development board's PD6 (SCL) and PD7 (SDA) pins as analog I2C pins. Using DuPont wires, we connected the GY521 module's SCL pin to PD6, SDA pin to PD7, GND pin to the development board's GND pin, and VCC pin to 3.3V.

Machine Translated by Google

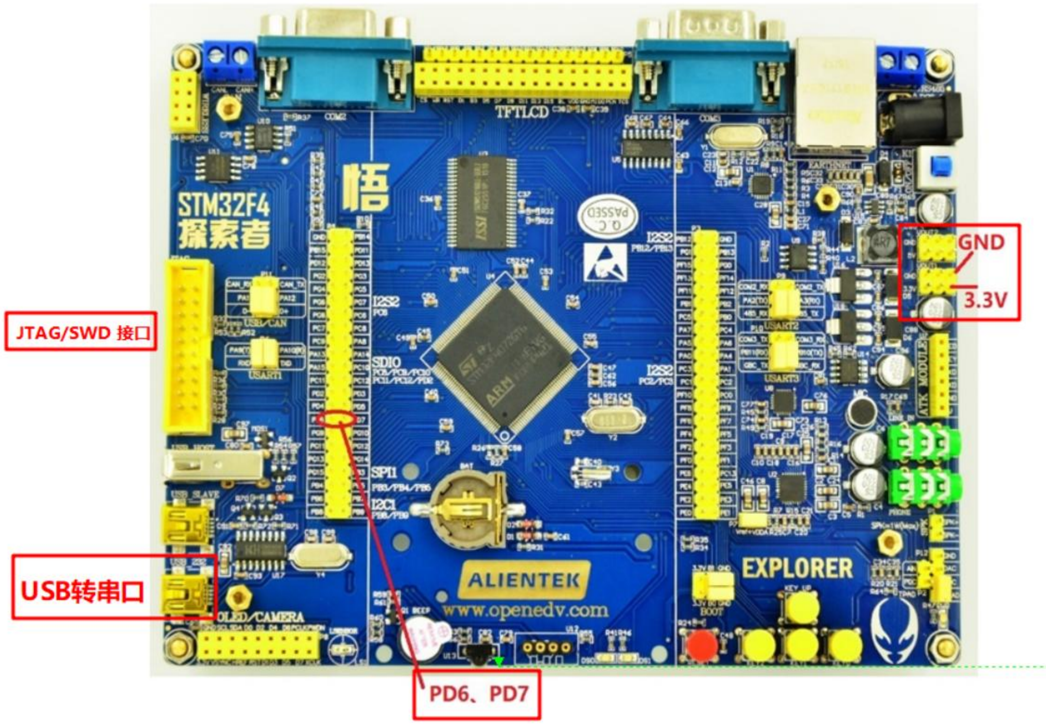

Figure 2: Zhengdian Atom Development Board

Figure 3: *GY521* Model Fast

This article is based on the Zhengdian Atom STM32F4 Explorer development board and provides the underlying I2C driver (GPIO simulation mode)

The adding method and specific application example code of I2C device (taking driving MPU6050 as an example), including register reading and writing

Machine Translated by Google

Due to the universality of RT-Thread upper-layer application API, these codes are not limited to specific hardware platforms.

Platform, users can easily port it to other platforms.

3.2 Enable **I2C** device driver

1. Use the env tool Enter the rt-thread\bsp\stm32f4xx-HAL directory in the command line and enter menuconfig

   command to enter the configuration interface.

2. Configure the shell to use serial port 1: Select Using UART1 and go to RT-Thread Kernel —> Kernel Device

   In the Object menu, change the device name for console to uart1.

3. Go to RT-Thread Components -> Device Drivers menu and select Using I2C device drivers.

   This example uses GPIO to simulate I2C, so you also need to turn on Use GPIO to simulate I2C.

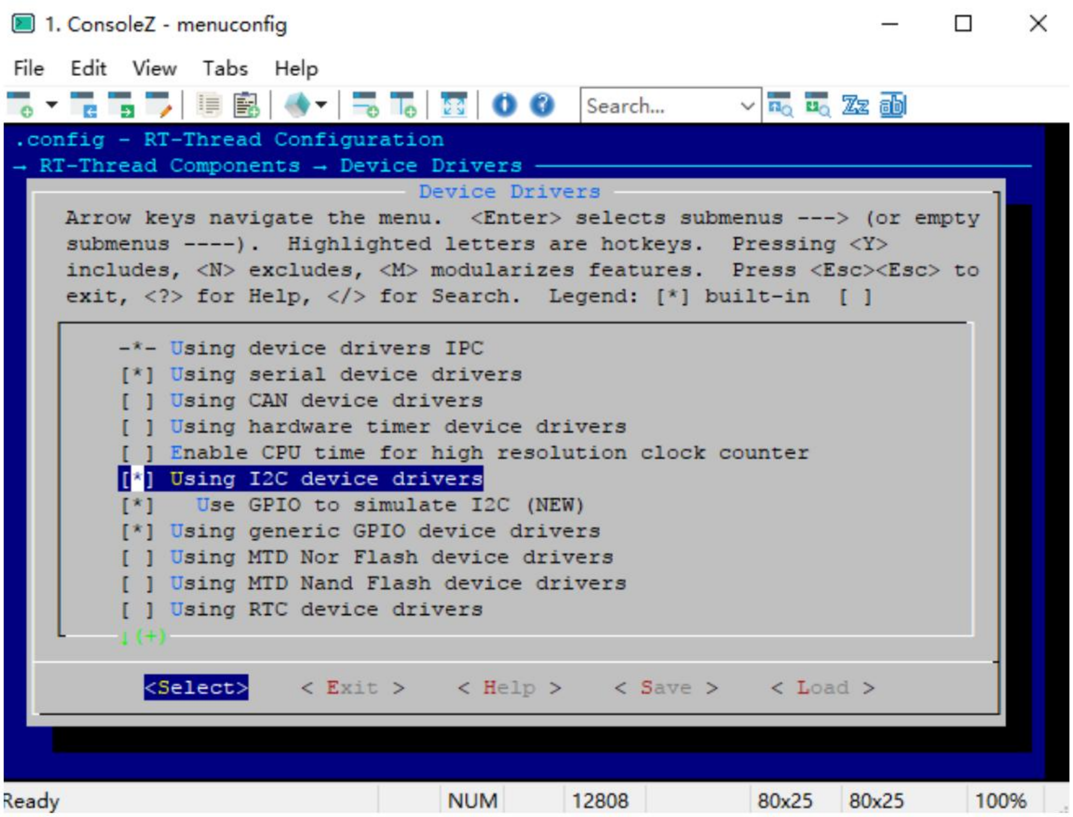

Figure 4: use *menuconfig* Open *i2c*

4. Exit the menuconfig configuration interface and save the configuration. Enter scons --target=mdk5 -s in the env command line

   The command generates an mdk5 project, and the new project is named project. Use MDK5 to open the project and modify the MCU model.

   For STM32F407ZGTx, change the debug option to J-LINK.

Machine Translated by Google

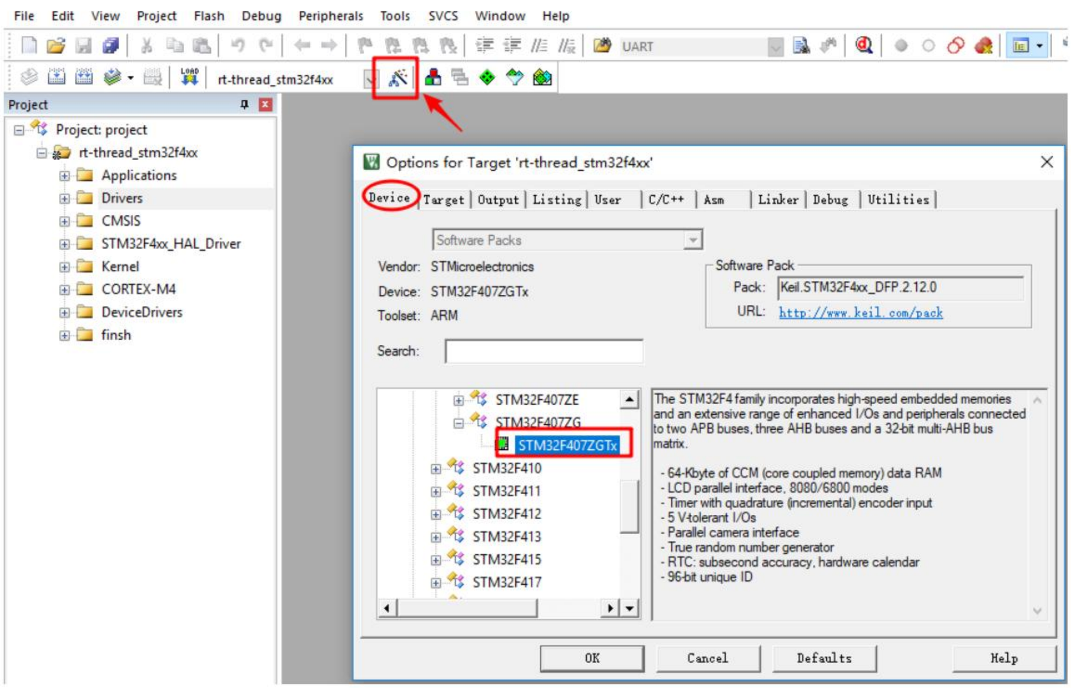

Figure 5: Revise *MCU*

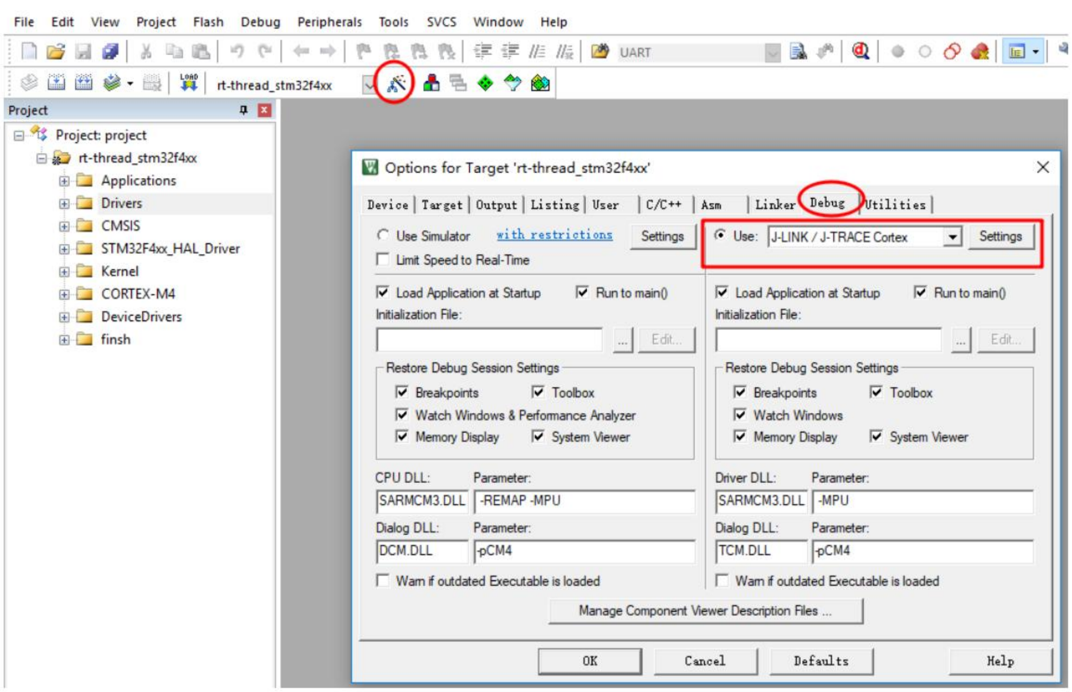

Figure 6: Modify debugging options

5. After compiling the project, download the program to the development board and run it. In the terminal PuTTY (open the corresponding port and configure the baud rate to 115200) Enter the list_device command to see the device named i2c2, the device type is I2C Bus,

The I2C device driver has been added successfully. As shown in the figure:

Machine Translated by Google

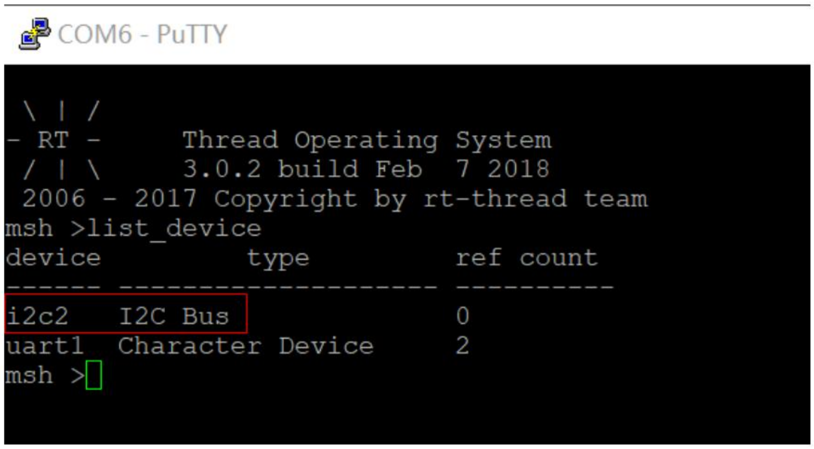

Figure 7: use *list_device* Command View *i2c* bus

3.3 Running the Example Code

Copy the main.c file in the I2C sample code to the \rt-thread\bsp\stm32f4xx-HAL\applications directory.

Record and replace the original main.c. Copy drv_mpu6050.c and drv_mpu6050.h to \rt-thread\bsp\stm32f4xx -HAL\drivers directory, and add them to the corresponding groups in the project. As shown in the figure:

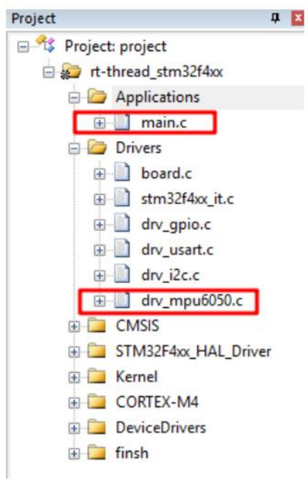

Figure 8: Adding a driver

This example uses GPIO PD6 as SCL, GPIO PD7 as SDA, and the I2C bus name is i2c2.

Modify the following parameters in drv_i2c.c file as needed to adapt to your own board, and ensure that the parameters defined in drv_mpu6050.c are

The macro MPU6050_I2C_BUS_NAME is the same as the macro I2C_BUS_NAME in drv_i2c.c .

To change the default driver port GPIOB of drv_i2c.c to GPIOD, as shown in the figure below:

Machine Translated by Google

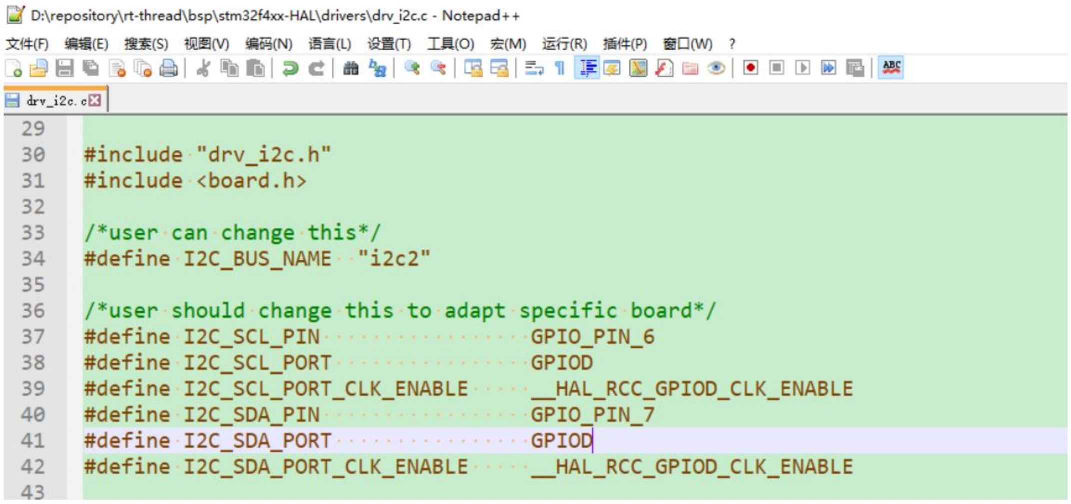

Figure 9: *drv_i2c.c* in *i2c* Board Configuration

Connect the MPU6050 module and the development board, compile the project and download the program to the development board, reset the MCU, and then PuTTY will print out the MPU6050 sensor data it reads, including temperature, three-axis acceleration, and three-axis angular velocity:

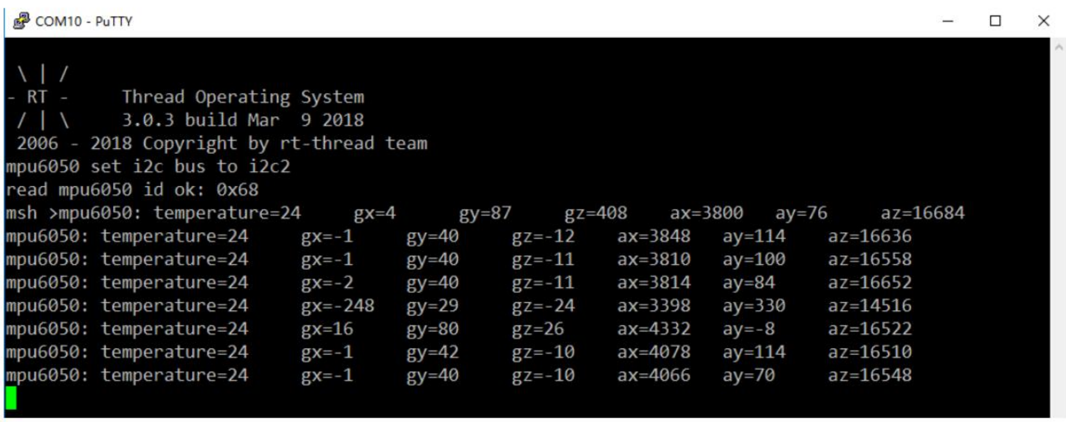

Figure 10: Terminal printing information

4 Detailed explanation of I2C device driver interface

Following the steps above, I believe readers can quickly run the RT-ThreadI2C device driver. How to develop applications using I2C device driver interface?

RT-Thread I2C device driver currently only supports host mode. To use RT-Thread I2C device driver, you need to use Use the menuconfig tool to enable the macros RT_USING_DEVICE and RT_USING_I2C if you want to use GPIO To simulate I2C, you also need to enable the macro RT_USING_I2C_BITOPS.

The general process of using I2C device driver is as follows:

1. Users can enter the list_device command in the msh shell to view the existing I2C devices and determine the I2C device name say.

Machine Translated by Google

2. Use rt_i2c_bus_device_find() or rt_device_find() to find the device and pass in the I2C device name to get the I2C bus device handle.

3. Use rt_i2c_transfer() to send and receive data. If the host only sends data, use rt_i2c_master_send(). If the host only receives data, use rt_i2c_master_recv().

Next, this chapter will explain in detail the use of the I2C device driver interface.

4.1 Finding Devices

If an application wants to use an I2C device that has been managed by the operating system, it needs to call the device search function. Only after the I2C device is found can information be transmitted to the device.

Function prototype: **struct** rt_i2c_bus_device *rt_i2c_bus_device_find(const **char** * bus_name)

parameter	describe
bus_name	I2C device name

Function returns: If the I2C device exists, it returns the I2C device handle, otherwise it returns RT_NULL.

The source code for searching the device in mpu6050_hw_init() in the underlying driver drv_mpu6050.c of this article's sample code is as follows:

```
#define MPU6050_I2CBUS_NAME "i2c2" /* I2C device name, must be added to drv_i2c.c
        The registered I2C device name is consistent with */
static struct rt_i2c_bus_device *mpu6050_i2c_bus; /* I2C device handle*/
... ...
... ...

int mpu6050_hw_init(void) {

    rt_uint8_t res;

    mpu6050_i2c_bus = rt_i2c_bus_device_find(MPU6050_I2CBUS_NAME); /* Find I2C device*/


    if (mpu6050_i2c_bus == RT_NULL) {

        MPUDEBUG("can't find mpu6050 %s device\r\n",MPU6050_I2CBUS_NAME); return
        -RT_ERROR;
    }

... ...
```

Machine Translated by Google


```
... ...
}
```

4.2 Data Transmission

The core API of RT-Thread I2C device driver is rt_i2c_transfer(), which transmits messages in a chained manner. Through the message chain, it can be called once to complete multiple data transmission and reception. This function can be used to send data, Can also be used to receive data.

Function prototype:

```
rt_size_t rt_i2c_transfer(struct rt_i2c_bus_device *bus,
                          struct rt_i2c_msg          msgs[],
                          rt_uint32_t                num)
```

parameter	describe
bus	I2C bus device handle
msgs[]	I2C message array
num	The number of message arrays

Function returns: the number of successfully transmitted message arrays

The message array msgs[] type is

```
struct rt_i2c_msg
{
        rt_uint16_t addr;          // Slave address
        rt_uint16_t flags; // Flags, read, write, etc.
        rt_uint16_t len; // read          // Number of bytes of read and write data
        rt_uint8_t *buf;        and write data pointer
}
```

addr slave address supports 7-bit and 10-bit binary addresses (flags |= RT_I2C_ADDR_10BIT). The slave addresses used by the I2C device driver interface of RT-Thread are all addresses that do not contain read and write bits. Change flags.

The optional flag value is the macro defined in the i2c.h file. The value of sending data is RT_I2C_WR, and the value of receiving data is The value RT_I2C_RD can be combined with other macros using the bitwise operation “|” as needed.

Machine Translated by Google


```
#define RT_I2C_WR                   0x0000
#define RT_I2C_RD (1u << 0)
#define RT_I2C_ADDR_10BIT (1u << 2) /* this is a ten bit chip address */
#define RT_I2C_NO_START (1u << 4)
#define RT_I2C_IGNORE_NACK (1u << 5)
#define RT_I2C_NO_READ_ACK (1u << 6) /* when I2C reading, we do not ACK
        */
```

4.2.1. Sending Data

Users can call the I2C device driver interface rt_i2c_master_send() or rt_i2c_transfer() to send

Data. The function call relationship is as follows:

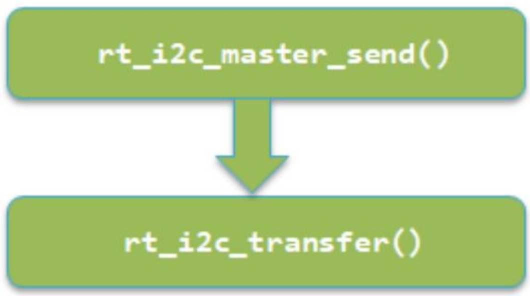

Figure **11:** Send data function call relationship

The mpu6050_write_reg() function in drv_mpu6050.c is used by the MCU to write data to the mpu6050 register.

There are two implementations of this function, which respectively call the I2C device driver interface rt_i2c_transfer() and

rt_i2c_master_send() implementation.

The MPU6050 datasheet used in this example mentions that the 7-bit slave address is 110100X, where X is the AD0 of the chip.

The pin determines that the AD0 of the GY521 module is connected to GND, so the address of the MPU6050 as a slave is 1101000.

The hexadecimal form is 0x68. To write a register of MPU6050, the host first sends the slave address MPU6050_ADDR,

The read/write flag R/W is RT_I2C_WR (0 for write, 1 for read), and then the host sends the slave register address reg and

datadata.

1) Use rt_i2c_transfer() to send data

The underlying driver drv_mpu6050.c of the sample code in this article sends the data source code as follows:

```
#define MPU6050_ADDR                        0X68

// Write a single register of mpu6050
//reg: register address
```

Machine Translated by Google


```
//data: data //
Return value: 0, normal / -1, error code rt_err_t
mpu6050_write_reg(rt_uint8_t reg, rt_uint8_t data) {

        struct rt_i2c_msg msgs;
        rt_uint8_t buf[2] = {reg, data};

        msgs.addr = MPU6050_ADDR;              /* Slave address*/ /
        msgs.flags = RT_I2C_WR;                * Write flag*/ /*
        msgs.buf = buf; msgs.len               Send data pointer*/
        = 2;

        if (rt_i2c_transfer(mpu6050_i2c_bus, &msgs, 1) == 1) {

                return RT_EOK;
        }
        else
        {
                return -RT_ERROR;
        }
}
```

Take the example code in this article where rt_i2c_transfer() is called to send data. The hexadecimal value of the slave MPU6050 address is 0X68, the hexadecimal value of the register address reg is 0X6B, and the hexadecimal value of the data sent is 0X80. The sample waveform is shown in the figure below. The first data sent is 0XD0. The upper 7 bits of the first data are the slave address, and the lowest bit is the read/write bit (value is 0). Therefore, the first data is: 0X68 << 1|0 = 0XD0, and then the register address 0X6B and data 0X80 are sent in sequence.

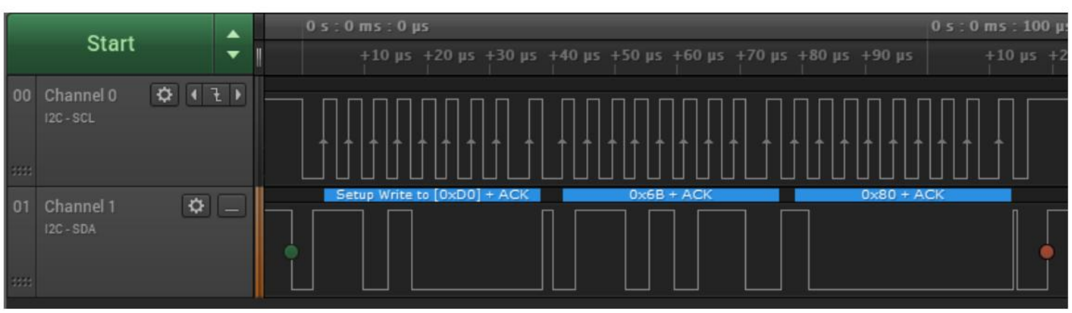

Figure **12:** *I2C* Transmitted data waveform example

2) Use rt_i2c_master_send() to send data

Function prototype:

```
rt_size_t rt_i2c_master_send(struct rt_i2c_bus_device *bus, rt_uint16_t addr,
```

Machine Translated by Google


```
                              rt_uint16_t              flags,
                              const rt_uint8_t         *buf,
                              rt_uint32_t              count)
```

parameter	describe
bus	I2C bus device handle
addr	Slave address, excluding read/write bits
flags	Flag, read and write flag is write. Only supports 10-bit address selection Select RT_I2C_ADDR_10BIT
buf	Pointer to the data to be sent
count	Number of data bytes sent

Function returns: the number of data bytes successfully sent.

This function is a simple encapsulation of rt_i2c_transfer() .

The underlying driver drv_mpu6050.c of the sample code in this article sends the data source code as follows:

```
#define MPU6050_ADDR                0X68

// Write a single register of mpu6050
//reg: register address
//data: data
// Return value: 0, normal / -1, error code
rt_err_t mpu6050_write_reg(rt_uint8_t reg, rt_uint8_t data)
{
    rt_uint8_t buf[2];

    buf[0] = reg;
    buf[1] = data;

    if (rt_i2c_master_send(mpu6050_i2c_bus, MPU6050_ADDR, 0, buf ,2) ==
        2)
    {
        return RT_EOK;
    }
    else
    {
        return -RT_ERROR;
    }
}
```

Machine Translated by Google

4.2.2. Receiving Data

Users can call the I2C device driver interface rt_i2c_master_recv() or rt_i2c_transfer() to receive data. The function call relationship is as follows:

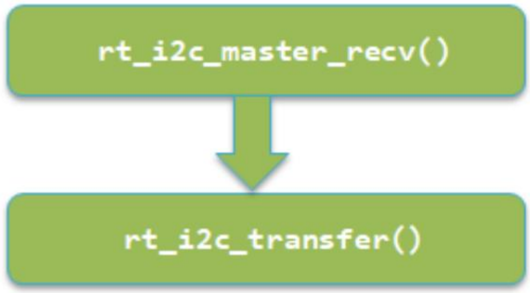

Figure 13: Receiving data function call relationship

The mpu6050_read_reg() function in the sample code drv_mpu6050.c in this article is used by the MCU to read data from the MPU6050 register. This function can also be implemented in two ways, calling the I2C device driver interface rt_i2c_transfer () and rt_i2c_master_recv() respectively .

To read a register of the MPU6050, the host first sends the slave address MPU6050_ADDR, the read/write flag R/W set to RT_I2C_WR (0 for write, 1 for read), and the slave register address reg before starting to read the device. Then, it sends the slave address MPU6050_ADDR, the read/write flag R/W set to RT_I2C_RD (0 for write, 1 for read), and saves the read data pointer.

1) Use rt_i2c_transfer() to receive data

The underlying driver drv_mpu6050.c of the sample code in this article receives the data source code as follows:

```
#define MPU6050_ADDR                                 0X68

// Read register data //reg:
register address to be read //len: number
of bytes of data to be read //buf: read
data storage area // Return value: 0,
normal / -1, error code rt_err_t
mpu6050_read_reg(rt_uint8_t reg, rt_uint8_t len, rt_uint8_t *buf
        )
{
        struct rt_i2c_msg msgs[2];

        msgs[0].addr = MPU6050_ADDR; msgs[0].flags      /* Slave address*/ /*
        = RT_I2C_WR;                                    Write flag*/
```

Machine Translated by Google


```
        msgs[0].buf = &reg;                     /* Slave register address */
        msgs[0].len = 1;                        /* Number of bytes of data sent */

        msgs[1].addr = MPU6050_ADDR;            /* Slave address */
        msgs[1].flags = RT_I2C_RD;              /* Read flags */
        msgs[1].buf = buf; msgs[1].len          /* Read data pointer */
        = len;                                  /* Number of bytes to read */

        if (rt_i2c_transfer(mpu6050_i2c_bus, msgs, 2) == 2)
        {
                return RT_EOK;
        }
        else
        {
                return -RT_ERROR;
        }
}
```

Take the example code in this article where rt_i2c_transfer() is called to receive data as an example. The slave MPU6050 address is The hexadecimal value is 0X68, and the register address reg is 0X75. The sample waveform is shown below. The first send The data is 0XD0, the first data has the upper 7 bits of the slave address, the lowest bit is the read/write bit (the value is 0), so the first A data value is: 0X68 << 1|0 = 0XD0, and then the register address 0X75 is sent. The data is 0XD1, the read/write bit is read (value is 1), the value is: 0X68 << 1 | 1 = 0XD1, and then the read data is received 0X68.

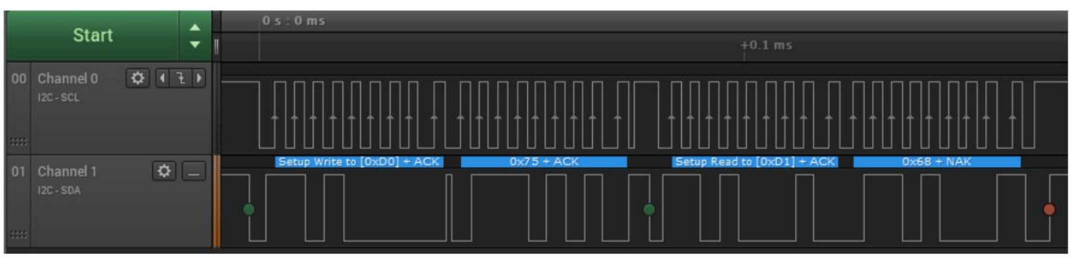

Figure **14:** *I2C* Transmitted data waveform example

2) Receive data using rt_i2c_master_recv()

Function prototype:

```
rt_size_t rt_i2c_master_recv(struct rt_i2c_bus_device *bus,
                             rt_uint16_t addr,
                             rt_uint16_t flags,
                             rt_uint8_t *buf,
                             rt_uint32_t                  count)
```

Machine Translated by Google

parameter	describe
bus	I2C bus device handle
addr	Slave address, excluding read/write bits
flags	Flag, read and write flag is read, only supports 10-bit address selection RT_I2C_ADDR_10BIT
buf	Accepts data pointer
count	Number of received data bytes

Function returns: the number of data bytes successfully received.

This function is a simple encapsulation of rt_i2c_transfer() and can only read data (receive data).

The underlying driver drv_mpu6050.c of the sample code in this article receives the data source code as follows:

```
#define MPU6050_ADDR                    0X68

// Read register data //
reg: register address to be read //len:
number of bytes of data to be read //
buf: read data storage area // Return
value: 0, normal / -1, error code rt_err_t
mpu6050_read_reg(rt_uint8_t reg, rt_uint8_t len, rt_uint8_t *buf
    )
{
    if (rt_i2c_master_send(mpu6050_i2c_bus, MPU6050_ADDR, 0, &reg, 1) ==
        1)
    {
        if (rt_i2c_master_recv(mpu6050_i2c_bus, MPU6050_ADDR, 0, buf, len
            ) == len)
        {
            return RT_EOK;
        }
        else
        {
            return -RT_ERROR;
        }
    }
    else
    {
        return -RT_ERROR;
    }
```

Machine Translated by Google


```
}
```

4.3 I2C Device Driver Application

Usually, the read-only registers of I2C interface chips are divided into two types: one is a single function register, and the other is a register with continuous addresses and similar functions. For example, the registers 0X3B, 0X3C, 0X3D, 0X3E, 0X3F, 0X40 stores the high 8 bits and low 8 bits of the three-axis acceleration X, Y, and Z axes in sequence.

The underlying driver drv_mpu6050.c in the sample code in this article uses the mpu6050_read_reg() function to read the 3-axis acceleration data of the MPU6050:

```
#define MPU_ACCEL_XOUTH_REG            0X3B        // Acceleration value, X-axis high 8-bit register

// Get acceleration value (raw
value) //gx,gy,gz: raw readings of gyroscope x,y,z axis (signed) //
Return value: 0, success / -1, error code rt_err_t
mpu6050_accelerometer_get(rt_int16_t *ax, rt_int16_t *ay,
        rt_int16_t *az)
{
        rt_uint8_t buf[6], ret;

        ret = mpu6050_read_reg(MPU_ACCEL_XOUTH_REG, 6, buf); if (ret == 0) {


                *ax = ((rt_uint16_t)buf[0] << 8) | buf[1]; *ay = ((rt_uint16_t)buf[2]
                << 8) | buf[3]; *az = ((rt_uint16_t)buf[4] << 8) | buf[5];


                return RT_EOK;
        }
        else
        {
                return -RT_ERROR;
        }
}
```