**一、硬件开发**

（一）思路整理

对于这个系统的整体设计，因为涉及到两个传感器以及ESP8266使用MQTT上报的部分，大概思路就是使用性能更好的STM32单片机做传感器数据采集，然后通过串口以一定格式和ESP8266通信，最后ESP8266 连接WIFI后通过MQTT协议进行上报，因为不需要太多的引脚，STM32选用STM32F103C8T6，ESP8266使用NODEMCU的板子。

（二）ESP8266部分

对于这个系统的整体设计，因为涉及到两个传感器以及ESP8266使用MQTT上报的部分，大概思路就是使用性能更好的STM32单片机做传感器数据采集，然后通过串口以一定格式和ESP8266通信，最后ESP8266 连接WIFI后通过MQTT协议进行上报，因为不需要太多的引脚，STM32选用STM32F103C8T6，ESP8266使用NODEMCU的板子。

void app\_main()

{

    char temperature[12] = "0";

    char pulse[12]       = "0";

    char spo2[12]         = "0";

    char ssid[32];

    char password[64];

    uint8\_t \*dat = (uint8\_t \*)malloc(BUF\_SIZE);

    int len,i = 1, j;

    ESP\_ERROR\_CHECK(nvs\_flash\_init());

    ESP\_LOGI(TAG, "ESP\_WIFI\_MODE\_STA");

    ESP\_LOGI(TAG, "[APP] Startup..");

    ESP\_LOGI(TAG, "[APP] Free memory: %d bytes", esp\_get\_free\_heap\_size());

    ESP\_LOGI(TAG, "[APP] IDF version: %s", esp\_get\_idf\_version());

    esp\_log\_level\_set("\*", ESP\_LOG\_INFO);

    esp\_log\_level\_set("MQTT\_CLIENT", ESP\_LOG\_VERBOSE);

    esp\_log\_level\_set("MQTT\_EXAMPLE", ESP\_LOG\_VERBOSE);

    esp\_log\_level\_set("TRANSPORT\_TCP", ESP\_LOG\_VERBOSE);

    esp\_log\_level\_set("TRANSPORT\_SSL", ESP\_LOG\_VERBOSE);

    esp\_log\_level\_set("TRANSPORT", ESP\_LOG\_VERBOSE);

    esp\_log\_level\_set("OUTBOX", ESP\_LOG\_VERBOSE);

    user\_gpio\_config();

    uart\_initialization();

    ESP\_LOGI(TAG, "[APP] Uart Init Finished.");

    if(gpio\_get\_level(GPIO\_NUM\_5) == 1){

        gpio\_set\_level(GPIO\_NUM\_2, 0);

        ESP\_LOGI(TAG, "[USER] Please Input Wifi ssid and password");

        ESP\_LOGI(TAG, "[USER] use this format: #ssid,password#\n");

        while (((len = uart\_read\_bytes(UART\_NUM\_0, dat, BUF\_SIZE,

                                    20 / portTICK\_RATE\_MS))) == 0 ||

            dat[0] != '#')

            ;

        gpio\_set\_level(GPIO\_NUM\_2, 1);

        while (dat[i] != ',') {

            ssid[i - 1] = (char)(dat[i]);

            i++;

        }

        ssid[i - 1] = '\0';

        j = i + 1;

        while (dat[j] != '#') {

            password[j - i - 1] = (char)(dat[j]);

            j++;

        }

        password[j - i - 1] = '\0';

        ESP\_LOGI(TAG,"[APP] Confirm. ssid:%s,password:%s\n", ssid, password);

        wifi\_init\_sta(ssid,

                    password);

        NVS\_store\_wifi(ssid, password);

    }else{

        gpio\_set\_level(GPIO\_NUM\_2, 1);

        NVS\_read\_wifi(ssid, password);

        ESP\_LOGI(TAG, "[APP] Confirm. ssid:%s,password:%s\n",

                 ssid, password);

        wifi\_init\_sta(ssid,

                      password);

    }

    ESP\_LOGI(TAG,"[APP] Wifi Init Finished.\n");

    mqtt\_app\_start();

    ESP\_LOGI(TAG,"[APP] MQTT Start.\n");

整体流程 ：系统设置->串口初始化->判断D1电平状态->读/写WIFI信息->初始化WIFI->连接WIFI

其中

if(gpio\_get\_level(GPIO\_NUM\_5) == 1){

        gpio\_set\_level(GPIO\_NUM\_2, 0);

        ESP\_LOGI(TAG, "[USER] Please Input Wifi ssid and password");

        ESP\_LOGI(TAG, "[USER] use this format: #ssid,password#\n");

        while (((len = uart\_read\_bytes(UART\_NUM\_0, dat, BUF\_SIZE,

                                    20 / portTICK\_RATE\_MS))) == 0 ||

            dat[0] != '#')

            ;

        gpio\_set\_level(GPIO\_NUM\_2, 1);

        while (dat[i] != ',') {

            ssid[i - 1] = (char)(dat[i]);

            i++;

        }

        ssid[i - 1] = '\0';

        j = i + 1;

        while (dat[j] != '#') {

            password[j - i - 1] = (char)(dat[j]);

            j++;

        }

        password[j - i - 1] = '\0';

        ESP\_LOGI(TAG,"[APP] Confirm. ssid:%s,password:%s\n", ssid, password);

        wifi\_init\_sta(ssid,

                    password);

        NVS\_store\_wifi(ssid, password);

用于将指定格式的WIFI数据解析并用于连接WIFI，然后使用NVS将其存储到掉电保护区。

主事件循环：

 while(1){

        len = uart\_read\_bytes(UART\_NUM\_0, dat, BUF\_SIZE, 20 / portTICK\_RATE\_MS);

        if(len > 0){

            ESP\_LOGI(TAG,"[APP] enter mqtt stream\n");

            if (strchr((const char \*)dat, '\n') != NULL) {

                dat[len] = '\0';

            }

            if (dat[0] == '{') {

                ESP\_LOGI(TAG,"[APP] Received data from sensor.\n");

                char publish\_buf[512] = "HelloWorld";

                user\_sensorArgsParse(temperature, pulse, spo2,

                                     (const char \*)dat);

                ESP\_LOGI(TAG, "[APP] Publish sensor data.\n");

                combine\_publish\_data((const char\*)temperature,(const char\*)pulse,(const char\*)spo2, publish\_buf);

                ESP\_LOGI(TAG,"[DEBUG] The published message:\n");

                ESP\_LOGI(TAG,"%s\n", publish\_buf);

                esp\_mqtt\_client\_publish(client, MQ\_MQTT\_TOPIC\_ATTRIBUTE\_PUBLISH, (const char\*)publish\_buf, 0, 0, 0);

            }

        }

        vTaskDelay(1000 / portTICK\_PERIOD\_MS);

        esp\_task\_wdt\_reset();

    }

整体思路： 从串口接收数据->判断是否满足格式要求->解析串口数据->拼接JSON->MQTT上报

NVS读写代码，类似于C语言的文件操作

/\*\*

 \* @brief 将WIFI的名称、密码存储掉电保护区�?

 \*

 \* @param ssid wifi名称

 \* @param password wifi密码

 \*/

void NVS\_store\_wifi(const char\* ssid , const char\* password){

    nvs\_handle handleNvs;

    if(nvs\_open("WIFI\_INFO", NVS\_READWRITE, &handleNvs) != ESP\_OK){

        ESP\_LOGE(TAG, "Open NVS Table fail.\n");

    }

*// �? WIFI ssid*

    if(nvs\_set\_str(handleNvs, "ssid", ssid) != ESP\_OK){

        ESP\_LOGE(TAG, "Save wifi ssid fail.\n");

    }else{

        ESP\_LOGI(TAG, "Save wifi ssid success. ssid: %s\n", ssid);

    }

*// �? WIFI 密码*

    if(nvs\_set\_str(handleNvs, "password", password) != ESP\_OK){

        ESP\_LOGE(TAG, "Save wifi password fail.\n");

    }else{

        ESP\_LOGI(TAG, "Save wifi password success. password: %s\n", password);

    }

    nvs\_commit(handleNvs);

    nvs\_close(handleNvs);

}

/\*\*

 \* @brief 从掉电保护区读取wifi数据

 \*

 \* @param ssid

 \* @param password

 \*/

void NVS\_read\_wifi(char\* ssid, char\* password){

    nvs\_handle handleNvs;

    char t\_ssid[32];

    char t\_password[64];

    if (nvs\_open("WIFI\_INFO", NVS\_READWRITE, &handleNvs) != ESP\_OK) {

        ESP\_LOGE(TAG, "Open NVS Table fail.\n");

    }

    uint32\_t len = sizeof(t\_ssid);

*// �? WIFI ssid*

    if(nvs\_get\_str(handleNvs, "ssid", t\_ssid, &len) != ESP\_OK){

        ESP\_LOGE(TAG, "Read Wifi ssid fail.\n");

    }else{

        ESP\_LOGI(TAG, "Read Wifi ssid success. ssid: %s\n", t\_ssid);

    }

    len = sizeof(t\_password);

*// �? WIFI 密码*

    if(nvs\_get\_str(handleNvs, "password", t\_password, &len) != ESP\_OK){

        ESP\_LOGE(TAG, "Read Wifi password fail.\n");

    }else{

        ESP\_LOGI(TAG, "Read Wifi password success. password: %s\n", t\_password);

    }

    strcpy(ssid, (const char \*)t\_ssid);

    strcpy(password, (const char \*)t\_password);

    nvs\_close(handleNvs);

}

解析串口数据：

*/\*  @brief 解析从串口传入的传感器数�? 格式 {A,B,C}*

*\**

*\*/*

void user\_sensorArgsParse(char\* temperature, char\* pulse,char\* spo2, const char\* sensor\_str){

    uint8\_t i = 1, j, k;

    while (sensor\_str[i] != ',') {

        pulse[i - 1] = sensor\_str[i];

        i++;

    }

    pulse[i - 1] = '\0';

    j = i + 1;

    while(sensor\_str[j] != ','){

        spo2[j - i - 1] = sensor\_str[j];

        j++;

    }

    spo2[j - i - 1] = '\0';

    k = j + 1;

    while (sensor\_str[k] != '}') {

        temperature[k - j - 1] = sensor\_str[k];

        k++;

    }

    temperature[k - j - 1] = '\0';

}

拼接JSON，使用 strcat 函数

void combine\_publish\_data(const char\* temperature, const char\* pulse,const char\* spo2 ,char \*publish\_buf){

    char temp[1024] = {0};

    strcpy(temp, "{");

    strcat(temp,"\"params\":{");

    strcat(temp,"\"Temp\":");

    strcat(temp, temperature);

    strcat(temp,",\"Pulse\":");

    strcat(temp, pulse);

    strcat(temp,",\"SPO2\":");

    strcat(temp, spo2);

    strcat(temp,"}}");

    strcpy(publish\_buf, (const char \*)temp);

}

Wifi、NVS、mqtt初始化以及事件抓取函数均参考官方SDK提供的Example：

MQTT初始化 需要配置HOST ，PORT

static void mqtt\_app\_start(void){

    esp\_mqtt\_client\_config\_t mcli\_config = {

        .host = CONFIG\_MQTT\_BORKER\_HOST,

        .port = 1883,

        .client\_id    = CONFIG\_MQTT\_CLIENT\_ID,

        .protocol\_ver = MQTT\_PROTOCOL\_V\_3\_1\_1,

        .keepalive = 60

    };

    client = esp\_mqtt\_client\_init(&mcli\_config);

    if(client == NULL){

        ESP\_LOGI(TAG,"[APP] MQTT client create failed.\r\n");

    }else{

        esp\_mqtt\_client\_register\_event(client, ESP\_EVENT\_ANY\_ID, mqtt\_event\_handler, client);

        esp\_mqtt\_client\_start(client);

    }

}

MQTT消息分发函数

static void mqtt\_event\_handler(void \*handler\_args, esp\_event\_base\_t base, int32\_t event\_id, void \*event\_data) {

    ESP\_LOGD(TAG, "Event dispatched from event loop base=%s, event\_id=%d", base, event\_id);

    mqtt\_event\_handler\_cb(event\_data);

}

该库使用消息分发以及捕获处理的思想，将MQTT通信中产生的各种事件以消息的形式反馈，而我们可以捕获这些消息做一些处理。

static esp\_err\_t mqtt\_event\_handler\_cb(esp\_mqtt\_event\_handle\_t event)

{

    esp\_mqtt\_client\_handle\_t client = event->client;

    int msg\_id;

*// your\_context\_t \*context = event->context;*

    switch (event->event\_id) {

        case MQTT\_EVENT\_CONNECTED:

            ESP\_LOGI(TAG, "MQTT\_EVENT\_CONNECTED");

            msg\_id = esp\_mqtt\_client\_publish(client, MQ\_MQTT\_TOPIC\_ATTRIBUTE\_PUBLISH, "device connect ok.", 0, 1, 0);

            ESP\_LOGI(TAG, "sent publish successful, msg\_id=%d", msg\_id);

            msg\_id = esp\_mqtt\_client\_subscribe(client, MQ\_MQTT\_TOPIC\_GET, 0);

            ESP\_LOGI(TAG, "sent subscribe successful, msg\_id=%d", msg\_id);

            msg\_id = esp\_mqtt\_client\_subscribe(client, MQ\_MQTT\_TOPIC\_GET, 1);

            ESP\_LOGI(TAG, "sent subscribe successful, msg\_id=%d", msg\_id);

            break;

        case MQTT\_EVENT\_DISCONNECTED:

            ESP\_LOGI(TAG, "MQTT\_EVENT\_DISCONNECTED");

            break;

        case MQTT\_EVENT\_SUBSCRIBED:

            ESP\_LOGI(TAG, "MQTT\_EVENT\_SUBSCRIBED, msg\_id=%d", event->msg\_id);

            msg\_id = esp\_mqtt\_client\_publish(client, MQ\_MQTT\_TOPIC\_ATTRIBUTE\_PUBLISH, "subcribe success.", 0, 0, 0);

            ESP\_LOGI(TAG, "sent publish successful, msg\_id=%d", msg\_id);

            break;

        case MQTT\_EVENT\_UNSUBSCRIBED:

            ESP\_LOGI(TAG, "MQTT\_EVENT\_UNSUBSCRIBED, msg\_id=%d", event->msg\_id);

            break;

        case MQTT\_EVENT\_PUBLISHED:

            ESP\_LOGI(TAG, "MQTT\_EVENT\_PUBLISHED, msg\_id=%d", event->msg\_id);

            break;

        case MQTT\_EVENT\_DATA:

            ESP\_LOGI(TAG, "MQTT\_EVENT\_DATA");

            ESP\_LOGI(TAG, "TOPIC=%.\*s\r\n", event->topic\_len, event->topic);

            ESP\_LOGI(TAG, "DATA=%.\*s\r\n", event->data\_len, event->data);

            break;

        case MQTT\_EVENT\_ERROR:

            ESP\_LOGI(TAG, "MQTT\_EVENT\_ERROR");

            break;

        default:

            ESP\_LOGI(TAG, "Other event id:%d", event->event\_id);

            break;

    }

    return ESP\_OK;

}

同样，WIFI连接中也会产生事件，也同样通过“消息”思想进行处理：

static void event\_handler(void\* arg, esp\_event\_base\_t event\_base,

                                int32\_t event\_id, void\* event\_data)

{

    if (event\_base == WIFI\_EVENT && event\_id == WIFI\_EVENT\_STA\_START) {

        esp\_wifi\_connect();

    } else if (event\_base == WIFI\_EVENT && event\_id == WIFI\_EVENT\_STA\_DISCONNECTED) {

        if (s\_retry\_num < EXAMPLE\_ESP\_MAXIMUM\_RETRY) {

            s\_retry\_num++;

            esp\_wifi\_connect();

            ESP\_LOGI(TAG, "retry to connect to the AP");

        } else {

            xEventGroupSetBits(s\_wifi\_event\_group, WIFI\_FAIL\_BIT);

        }

        ESP\_LOGI(TAG,"connect to the AP fail");

    } else if (event\_base == IP\_EVENT && event\_id == IP\_EVENT\_STA\_GOT\_IP) {

        ip\_event\_got\_ip\_t\* event = (ip\_event\_got\_ip\_t\*) event\_data;

        ESP\_LOGI(TAG, "got ip:%s",

                 ip4addr\_ntoa(&event->ip\_info.ip));

        s\_retry\_num = 0;

        xEventGroupSetBits(s\_wifi\_event\_group, WIFI\_CONNECTED\_BIT);

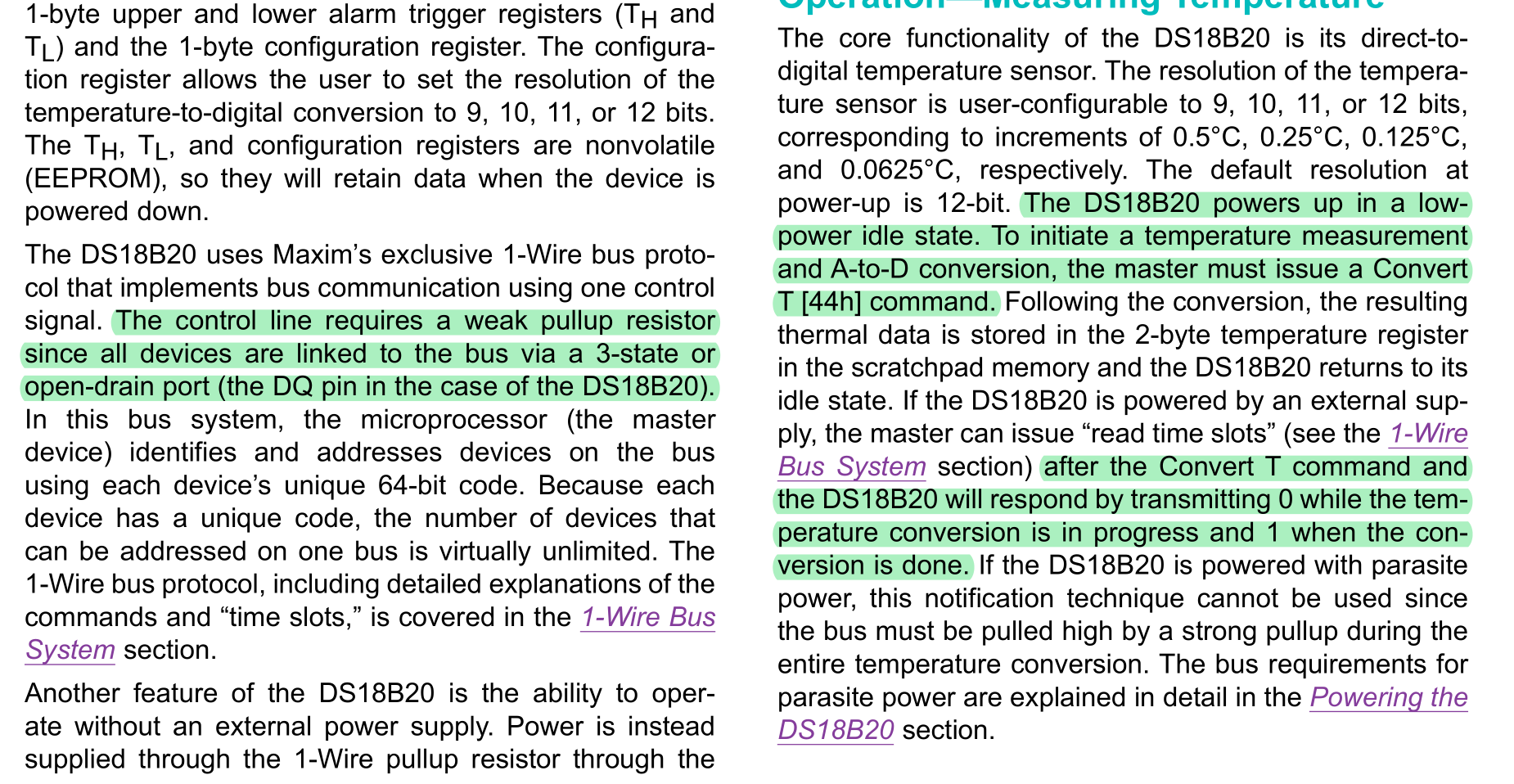
    }

}

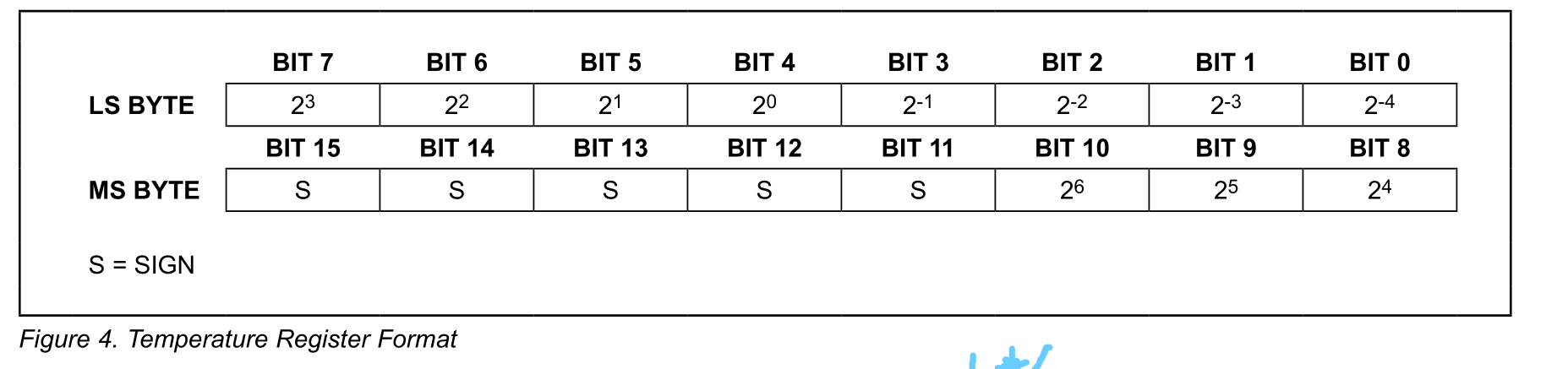
（三）STM32 部分：

传感器采集参数需要有心率、血氧以及温度三个参数。温度模块我选用简单易上手且网络资料丰富的DS18B20， 它是一个单线通信的模块。

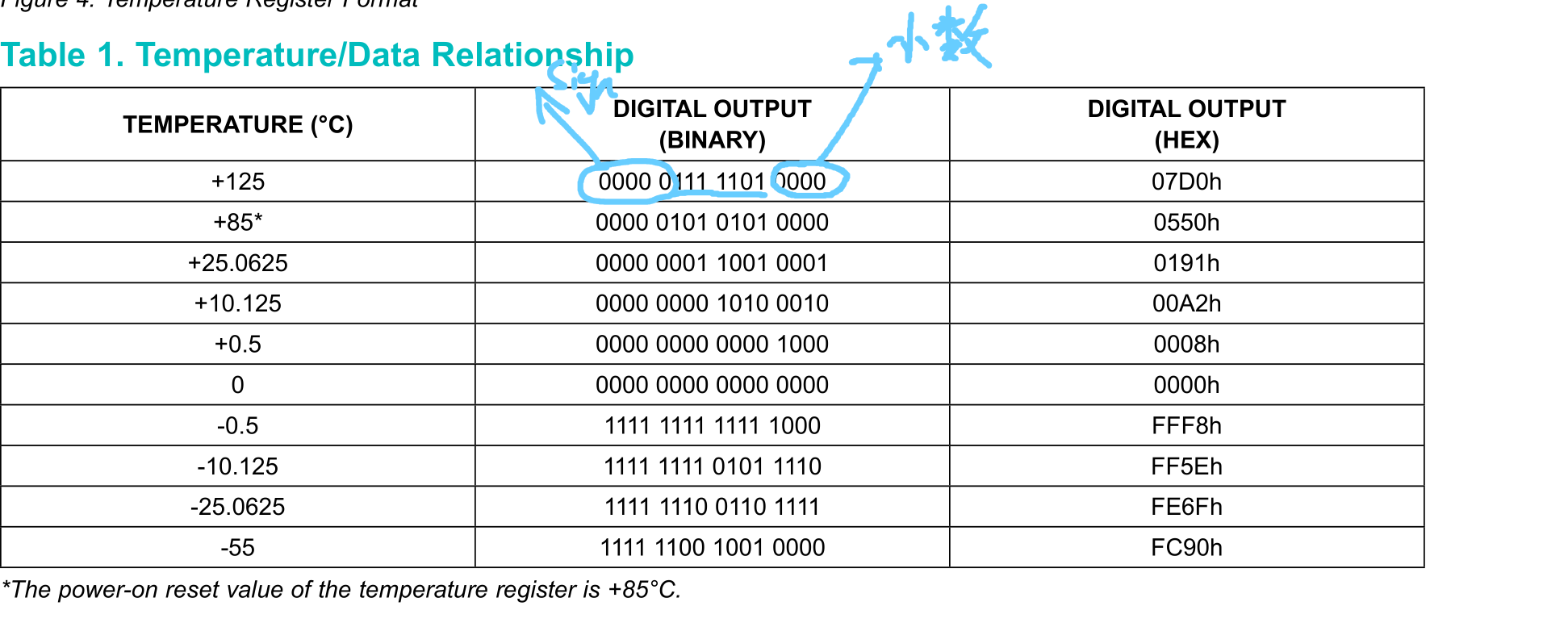
DS18B20的驱动部分按照芯片手册的时序以及命令介绍编写



DS18B20的存储格式：



可以看出它占用了两个字节，也就是读的时候需要读两次。而低字节中的低4位存储的是小数部分的数据。



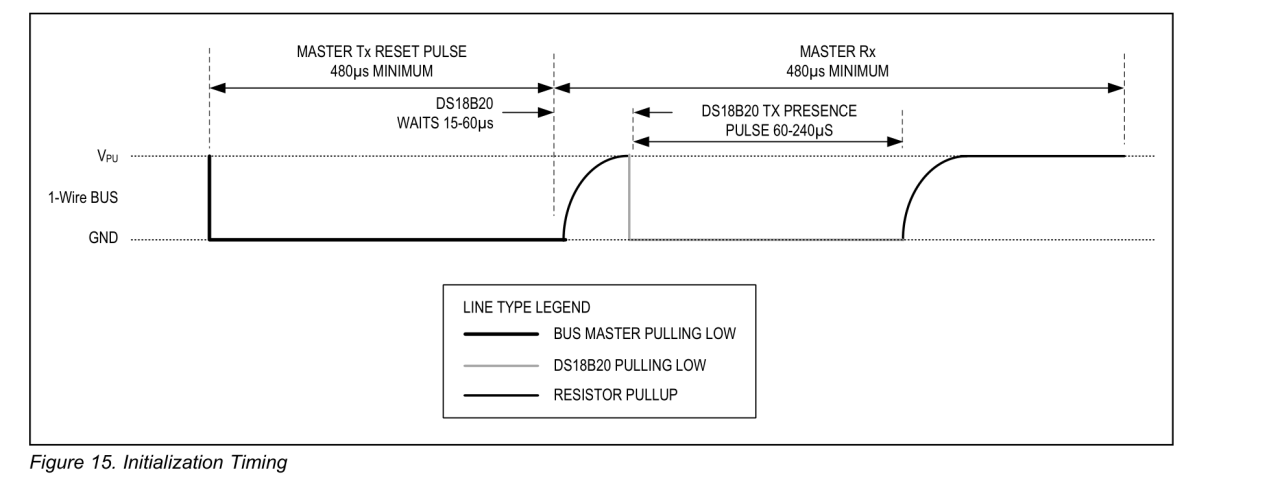
DS18B20的命令部分



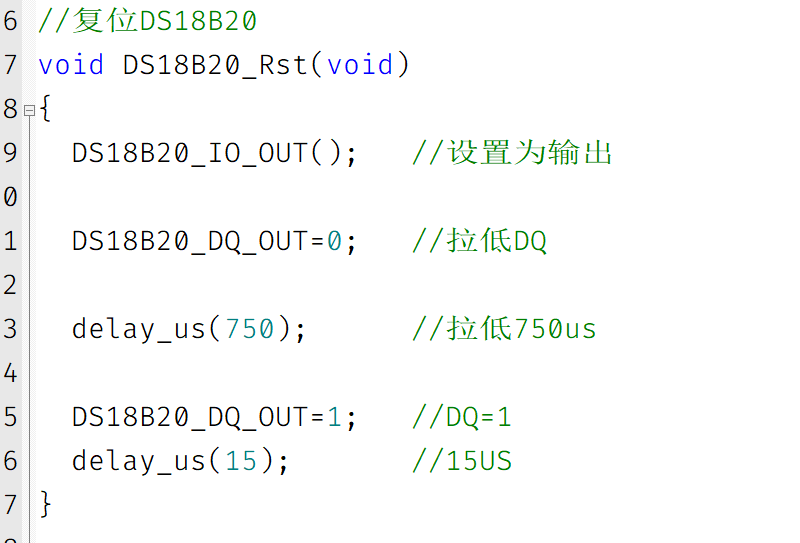


既然要写命令，肯定需要有能够往DS18B20中写的函数，此时我们需要参考DS18B20的时序图来编写“写”命令。

DS18B20的初始化时序图



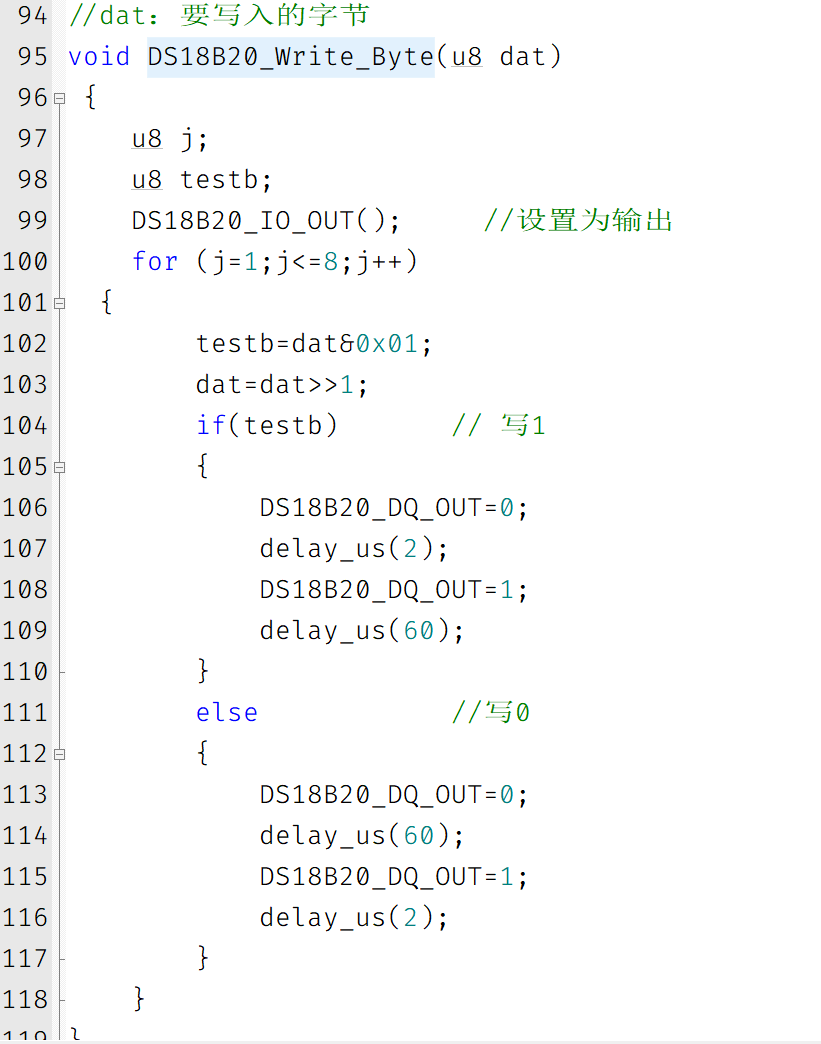
对应代码如下：



写和读时序

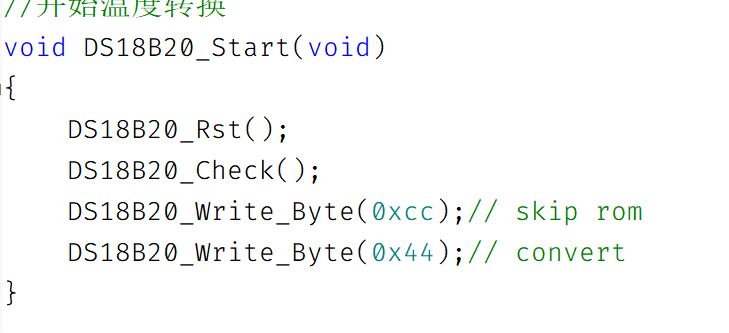


对应代码：

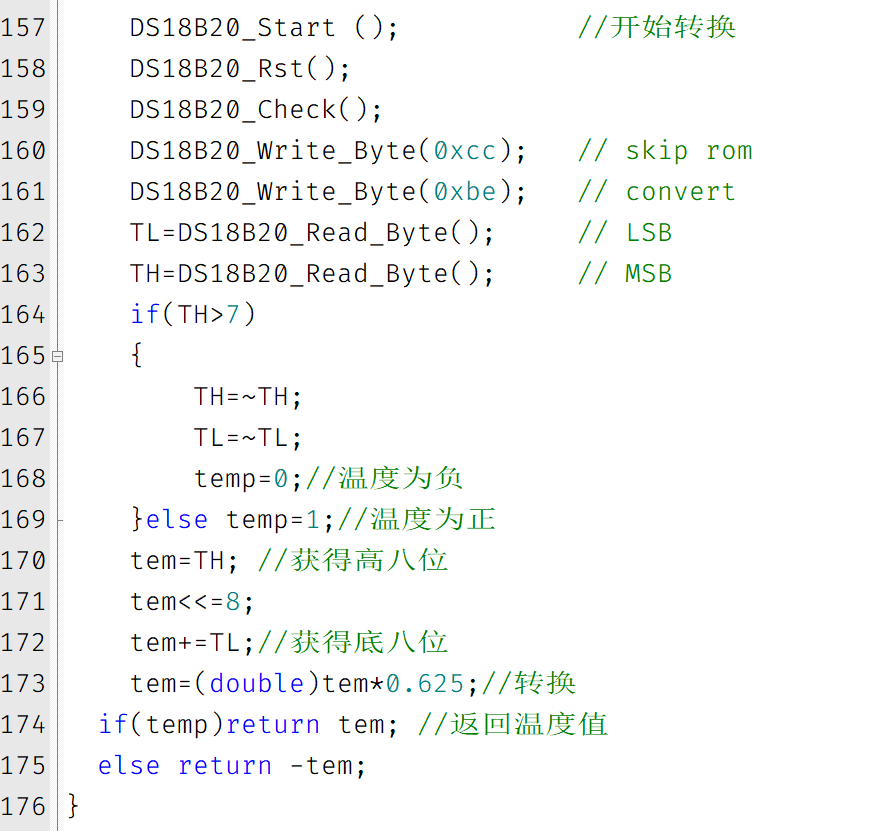


有了这两个函数后，就可以进行DS18B20的读写操作了，这时候可以将命令作为数据写入DS18B20。DS18B20的开始启动需要 复位-> 跳过ROM -> 开始转换

跳过ROM的命令为 0xCC , 开始转换的命令为 0x44：



此时DS18B20会读取温度并开始转换，但温度还只是存储在它的寄存器中，我们同样需要命令将其读出来。



其中 0xbe 即向DS18B20发送准备读它的寄存器。

第一次读取是读取的温度数据的低8位，再读一次则返回高8位，再根据之前的存储表格即可算出温度数据。

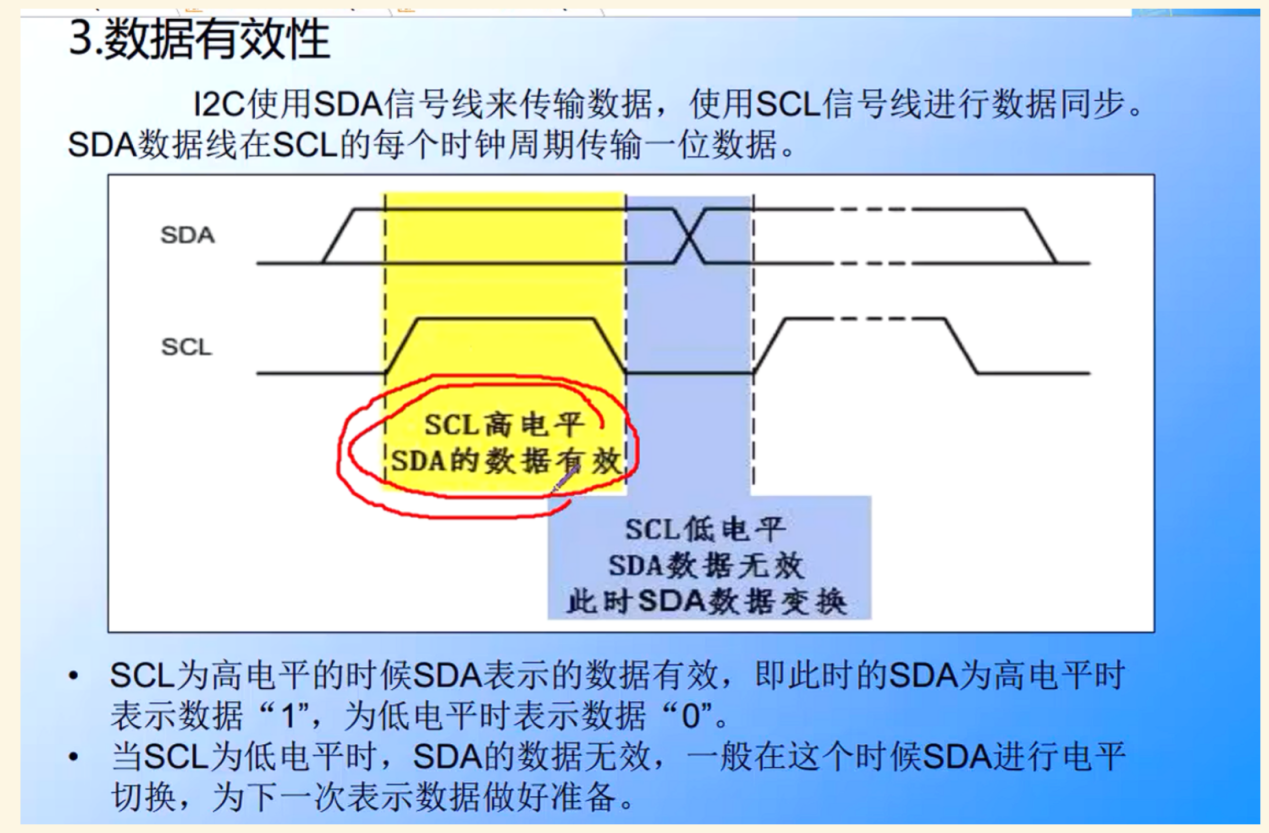
接下来就是我认为最为麻烦的心率、血氧的读取，选用的芯片是MAX30100，它是使用I2C协议进行通信。起初我以为它很简单，但是深入了解后发现它极为复杂，它并不能像DS18B20那样能够直接通过I2C读取转换的数据，而只能读取到通过红外光和红色LED照射返回的AD值，且这个值是一个变化的(交流的值)，我们要计算心率，必须对这个值以一定采样率、进行一定数量样本的采样并分析计算，才能得到结果。

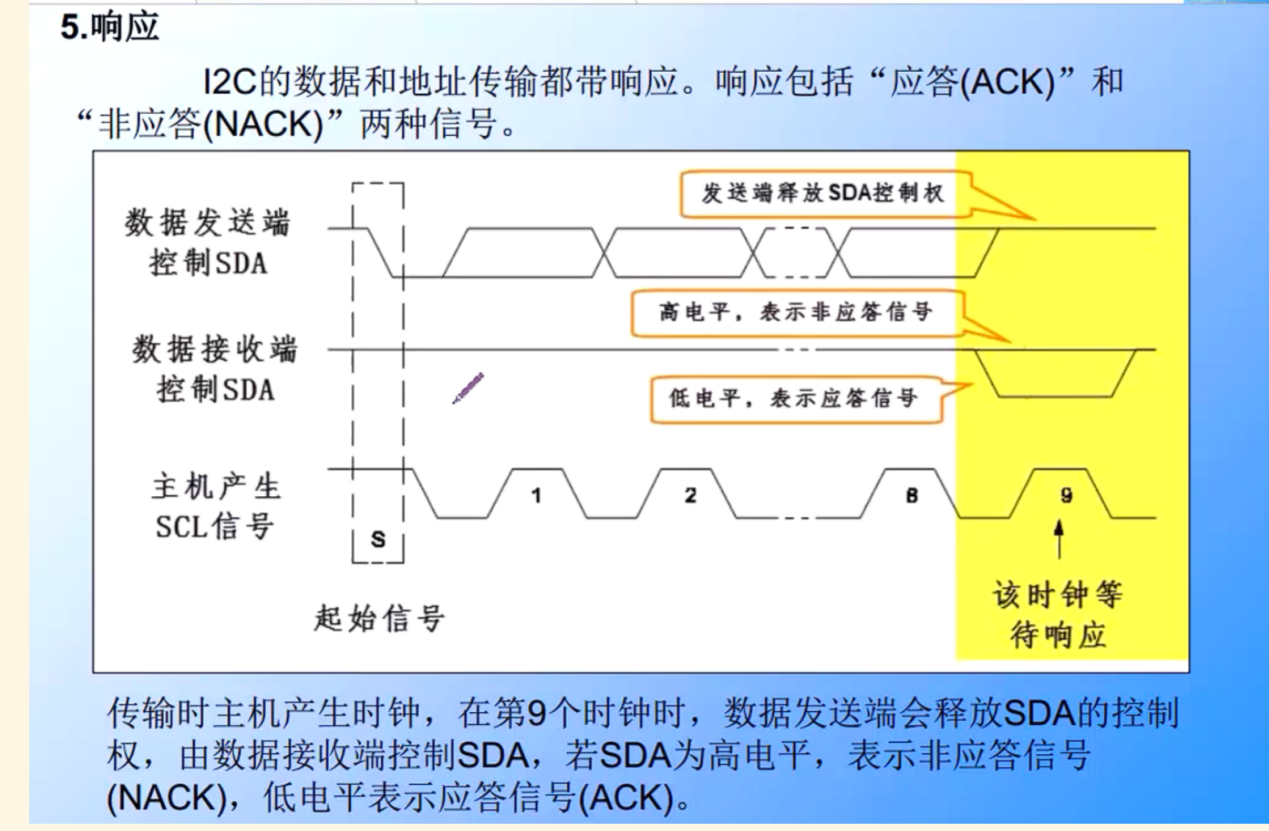
但是这个模块自带的滤波系统只能滤除50/60HZ的干扰信号，而人体的接触不良、外界的因素也会对模块本身造成大量的干扰，必须使用软件的滤波算法进行处理。

首先是模块的驱动部分，该模块的觉得大部分的工作模式以及参数设置，都需要软件编写驱动来进行设置，我们还是从最开始的读写开始。

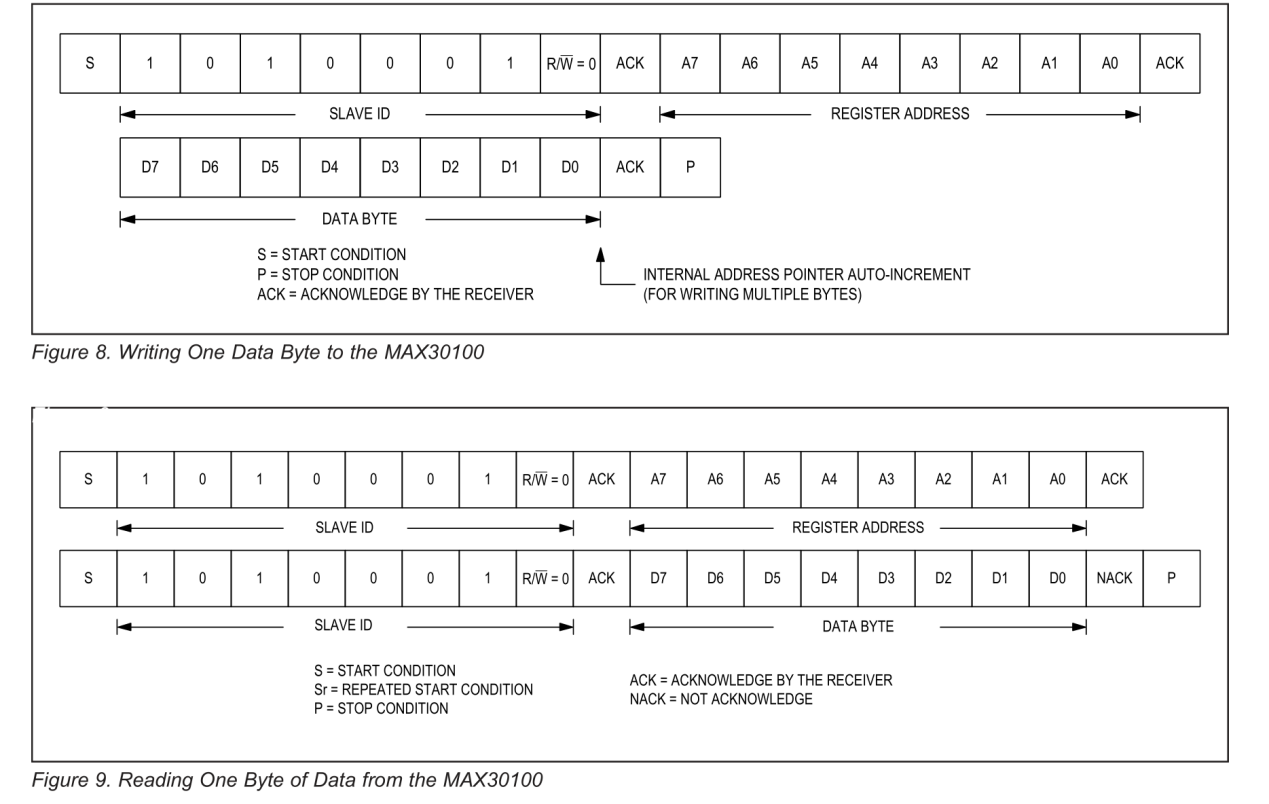
首先必须要一套I2C的通信协议驱动，包含 起始信号、结束信号、写字节信号、读字节信号、应答和非应答信号。







在按照这些时序写完I2C的驱动后，接下来才能进入MAX30100的驱动编写，首先还是从最基本的读、写开始。



与前面DS18B20的时序图不同的是，这里给出的是一个I2C的通信信号图，写出的代码如下：

uint8\_t max30100\_Bus\_Write(uint8\_t Register\_Address, uint8\_t Word\_Data)

{

/\* 采用串行EEPROM随即读取指令序列，连续读取若干字节 \*/

/\* 第1步：发起I2C总线启动信号 \*/

I2C\_Start();

/\* 第2步：发起控制字节，高7bit是地址，bit0是读写控制位，0表示写，1表示读 \*/

I2C\_SendByte(max30100\_WR\_address | I2C\_WR); /\* 此处是写指令 \*/

/\* 第3步：发送ACK \*/

if (I2C\_WaitAck() != 0)

{

goto cmd\_fail; /\* EEPROM器件无应答 \*/

}

/\* 第4步：发送字节地址 \*/

I2C\_SendByte(Register\_Address);

if (I2C\_WaitAck() != 0)

{

goto cmd\_fail; /\* EEPROM器件无应答 \*/

}

/\* 第5步：开始写入数据 \*/

I2C\_SendByte(Word\_Data);

/\* 第6步：发送ACK \*/

if (I2C\_WaitAck() != 0)

{

goto cmd\_fail; /\* EEPROM器件无应答 \*/

}

/\* 发送I2C总线停止信号 \*/

I2C\_Stop();

return 1; /\* 执行成功 \*/

cmd\_fail: /\* 命令执行失败后，切记发送停止信号，避免影响I2C总线上其他设备 \*/

/\* 发送I2C总线停止信号 \*/

I2C\_Stop();

return 0;

}

uint8\_t max30100\_Bus\_Read(uint8\_t Register\_Address)

{

uint8\_t data;

/\* 第1步：发起I2C总线启动信号 \*/

I2C\_Start();

/\* 第2步：发起控制字节，高7bit是地址，bit0是读写控制位，0表示写，1表示读 \*/

I2C\_SendByte(max30100\_WR\_address | I2C\_WR); /\* 此处是写指令 \*/

/\* 第3步：发送ACK \*/

if (I2C\_WaitAck() != 0)

{

goto cmd\_fail; /\* EEPROM器件无应答 \*/

}

/\* 第4步：发送字节地址， \*/

I2C\_SendByte((uint8\_t)Register\_Address);

if (I2C\_WaitAck() != 0)

{

goto cmd\_fail; /\* EEPROM器件无应答 \*/

}

/\* 第6步：重新启动I2C总线。下面开始读取数据 \*/

I2C\_Start();

/\* 第7步：发起控制字节，高7bit是地址，bit0是读写控制位，0表示写，1表示读 \*/

I2C\_SendByte(max30100\_WR\_address | I2C\_RD); /\* 此处是读指令 \*/

/\* 第8步：发送ACK \*/

if (I2C\_WaitAck() != 0)

{

goto cmd\_fail; /\* EEPROM器件无应答 \*/

}

/\* 第9步：读取数据 \*/

{

data = I2C\_RadeByte(); /\* 读1个字节 \*/

I2C\_NoAck(); /\* 最后1个字节读完后，CPU产生NACK信号(驱动SDA = 1) \*/

}

/\* 发送I2C总线停止信号 \*/

I2C\_Stop();

return data; /\* 执行成功 返回data值 \*/

cmd\_fail: /\* 命令执行失败后，切记发送停止信号，避免影响I2C总线上其他设备 \*/

/\* 发送I2C总线停止信号 \*/

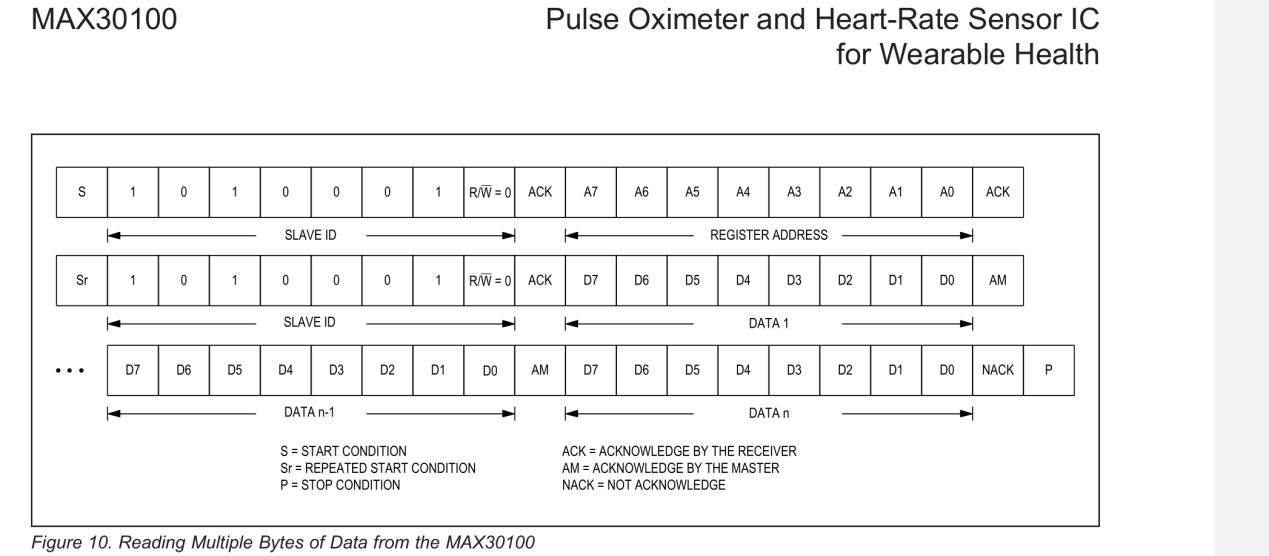
I2C\_Stop();

return 0;

}

可以看出，MAX30100即便是在读写数据，也不是标准的I2C协议，他其中夹杂了多次起始信号。

除了基本的读写1字节的命令外，MAX30100内部存在一个FIFO队列，用来存储采集到的数据AD值，我们需要读这个寄存器来获得数据：



void max30100\_FIFO\_Read(uint8\_t Register\_Address,uint16\_t Word\_Data[][2],uint8\_t count)

{

uint8\_t i=0;

uint8\_t no = count;

uint8\_t data1, data2;

/\* 第1步：发起I2C总线启动信号 \*/

I2C\_Start();

/\* 第2步：发起控制字节，高7bit是地址，bit0是读写控制位，0表示写，1表示读 \*/

I2C\_SendByte(max30100\_WR\_address | I2C\_WR); /\* 此处是写指令 \*/

/\* 第3步：发送ACK \*/

if (I2C\_WaitAck() != 0)

{

goto cmd\_fail; /\* EEPROM器件无应答 \*/

}

/\* 第4步：发送字节地址， \*/

I2C\_SendByte((uint8\_t)Register\_Address);

if (I2C\_WaitAck() != 0)

{

goto cmd\_fail; /\* EEPROM器件无应答 \*/

}

/\* 第6步：重新启动I2C总线。下面开始读取数据 \*/

I2C\_Start();

/\* 第7步：发起控制字节，高7bit是地址，bit0是读写控制位，0表示写，1表示读 \*/

I2C\_SendByte(max30100\_WR\_address | I2C\_RD); /\* 此处是读指令 \*/

/\* 第8步：发送ACK \*/

if (I2C\_WaitAck() != 0)

{

goto cmd\_fail; /\* EEPROM器件无应答 \*/

}

/\* 第9步：读取数据 \*/

while (no)

{

data1 = I2C\_RadeByte();

I2C\_Ack();

data2 = I2C\_RadeByte();

I2C\_Ack();

Word\_Data[i][0] = (((uint16\_t)data1 << 8) | data2); //

data1 = I2C\_RadeByte();

I2C\_Ack();

data2 = I2C\_RadeByte();

if(1==no)

I2C\_NoAck(); /\* 最后1个字节读完后，CPU产生NACK信号(驱动SDA = 1) \*/

else

I2C\_Ack();

Word\_Data[i][1] = (((uint16\_t)data1 << 8) | data2);

no--;

i++;

}

/\* 发送I2C总线停止信号 \*/

I2C\_Stop();

cmd\_fail: /\* 命令执行失败后，切记发送停止信号，避免影响I2C总线上其他设备 \*/

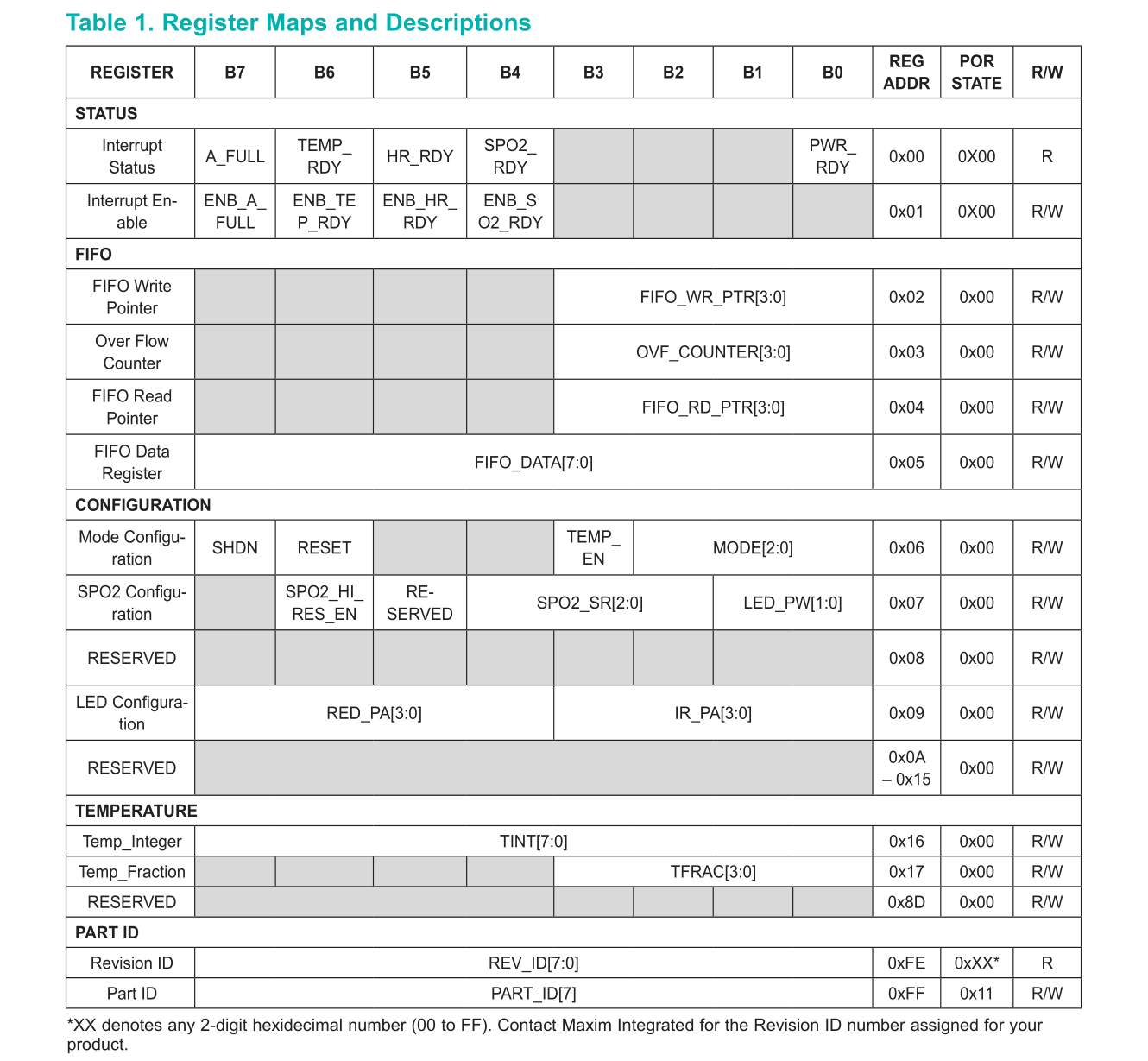
/\* 发送I2C总线停止信号 \*/

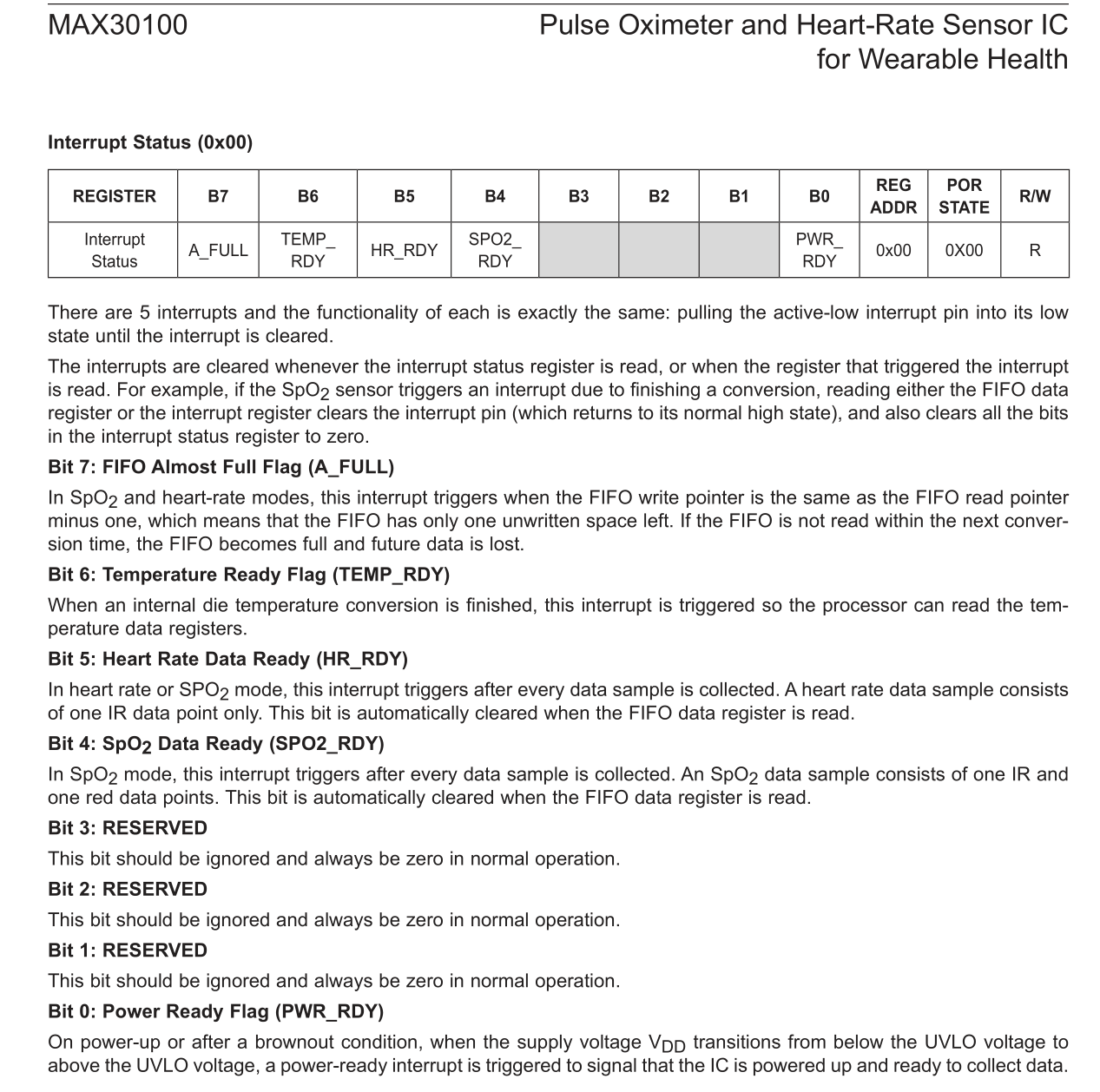
I2C\_Stop();

}

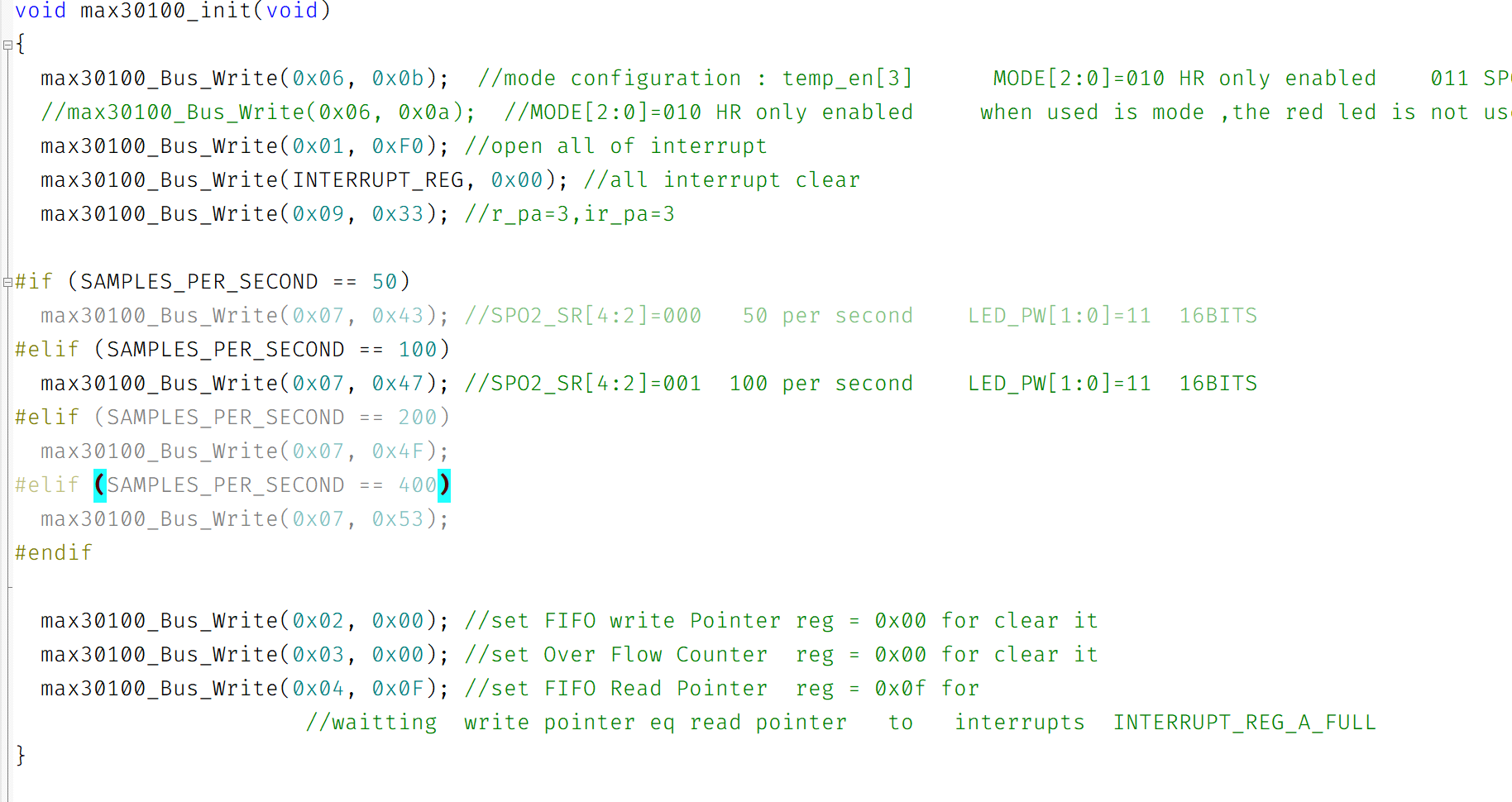
有了这三个最基本的函数过后，我们就可以进行MAX30100的基本配置了，但也仅仅还只是配置阶段，离读取到心率和血氧还远远不够。

配置MAX30100涉及工作模式、LED工作电流、采样率等等大量参数





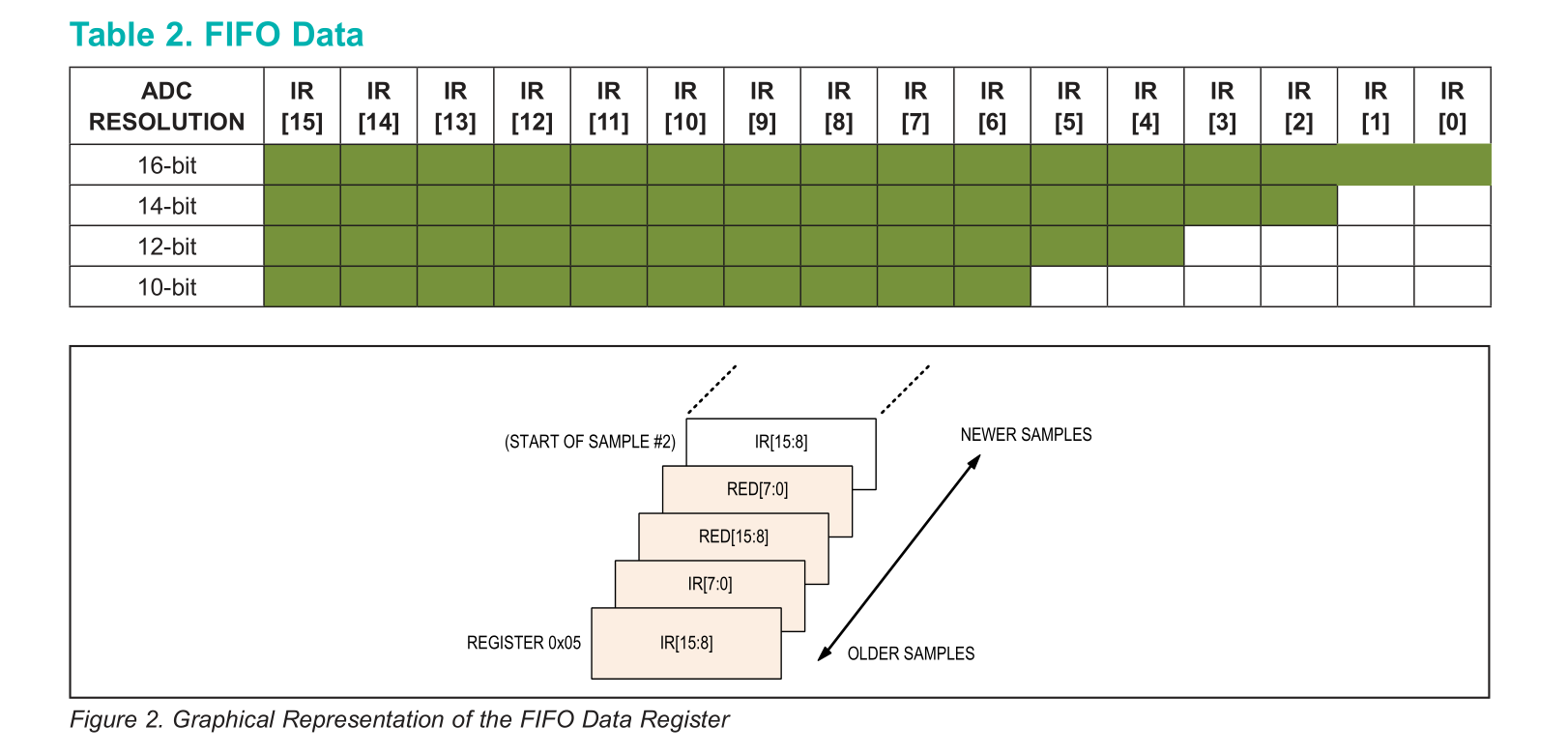
MAX30100的基本初始化过程如下：



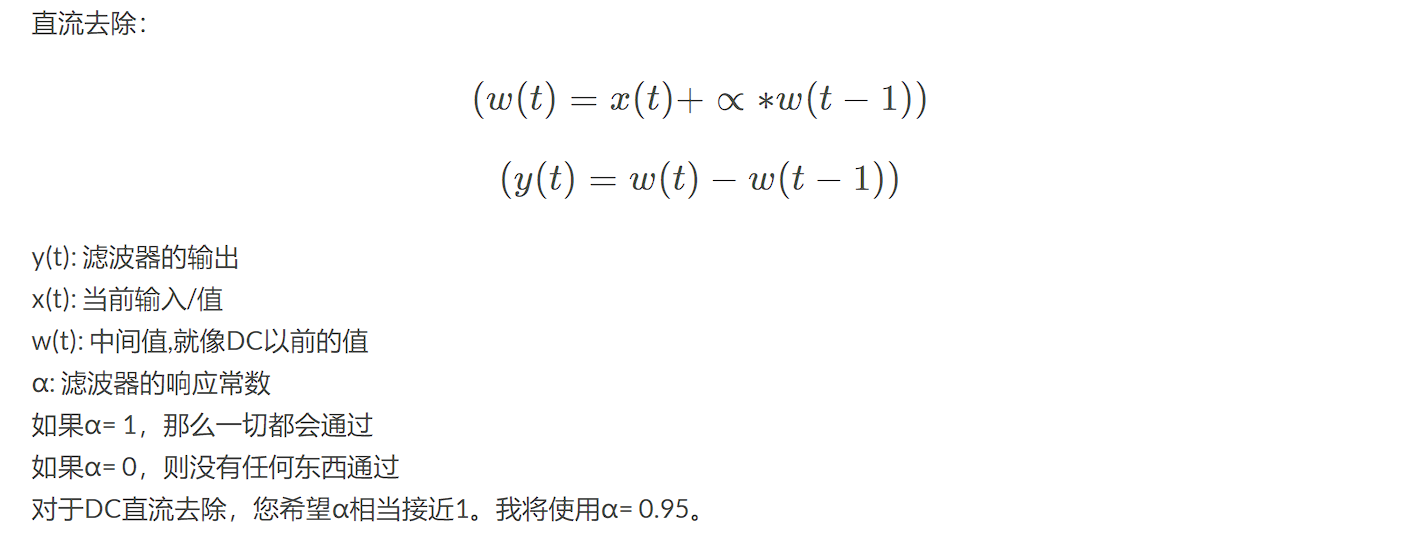
这里有三个对于我们采集参数重要的指针，分别为FIFO写指针(地址0x02)、计数溢出指针(0x03)、FIFO读指针(0x04)。

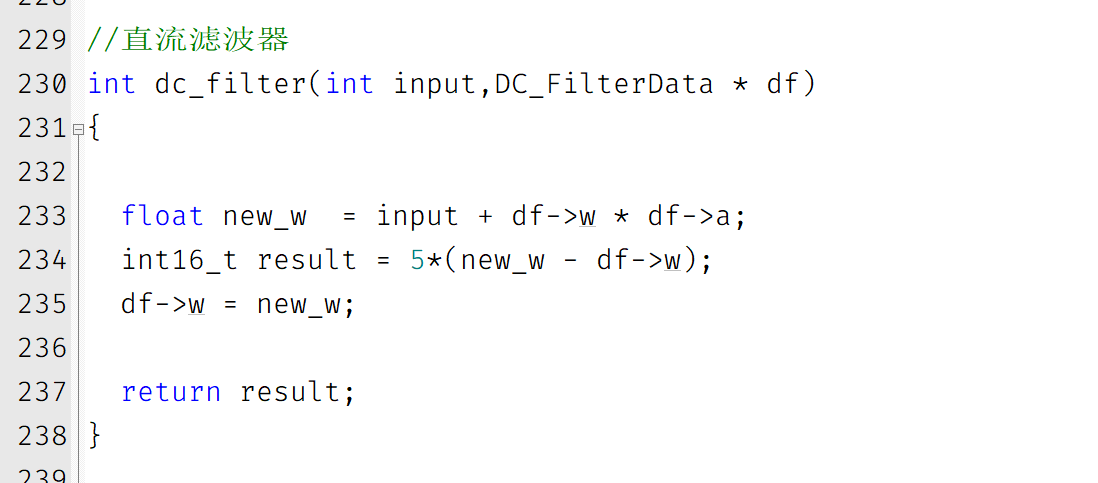
每次我们使用他们的时候他们会自动累加，需要我们发送命令归位。

FIFO中存储了两种数据，一种是（RED表示血氧）另一种是（IR）心率

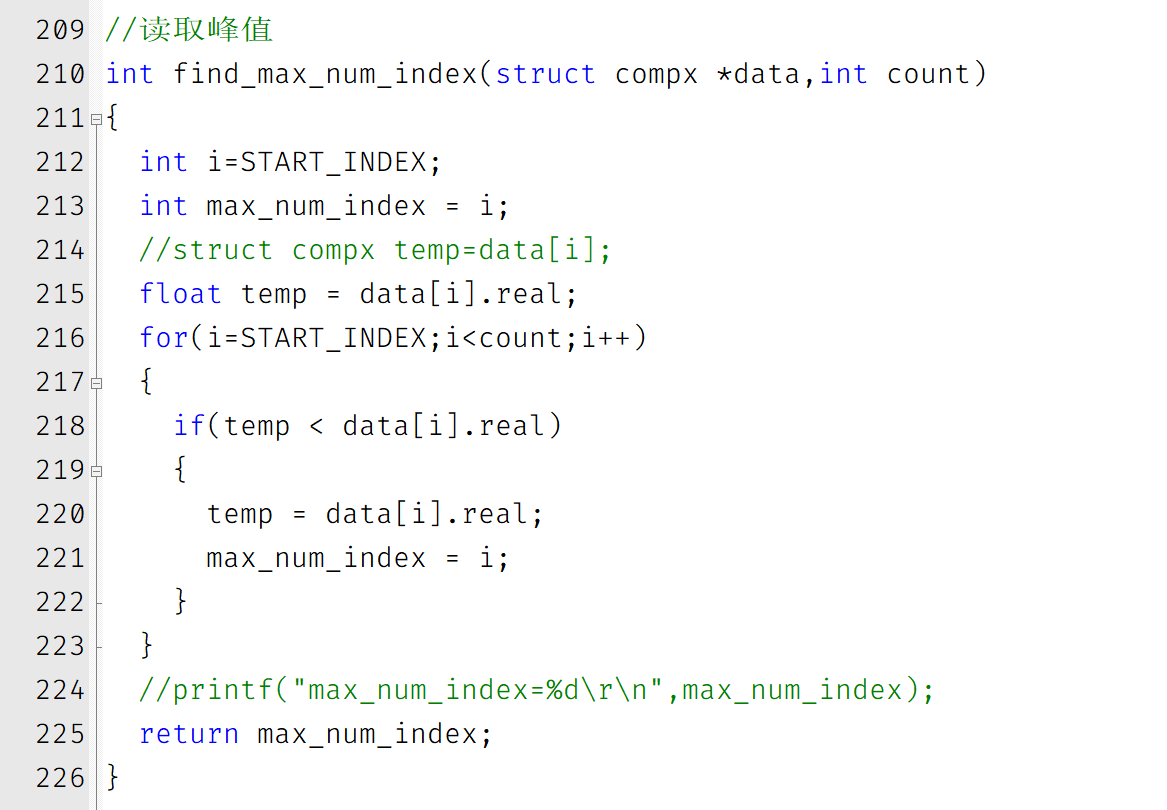


读到的数据中只有交流成分是我们需要的信号，所以要进行直流成分的滤波。

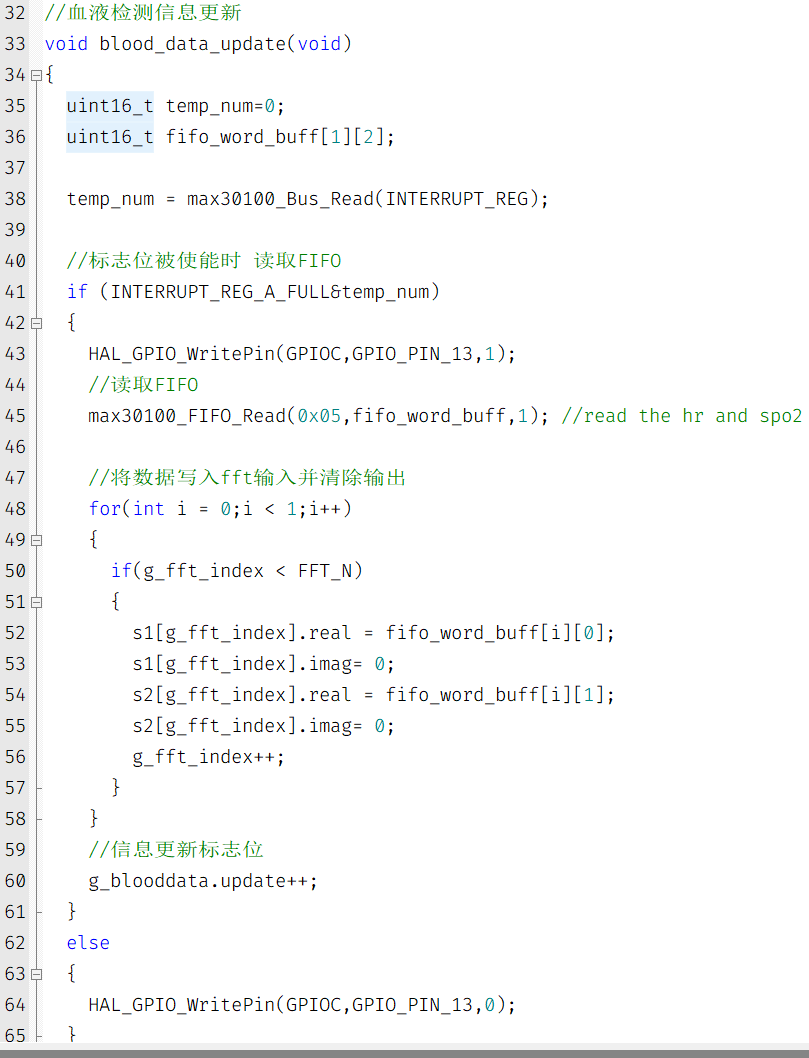




而对于心率的计算，我们需要将读取到的IR样本取峰值，然后计算峰值间隔来得到心率。



有了上面的函数，就可以开始做信息的采集部分了



该函数从FIFO读入数据后，会作为单次样本存储起来，当样本个数达到FFT\_N的个数后，g\_blooddata.update这个信息更新的标志位个数也会达到预期值，然后进行后续的处理(滤波等)，同时用板载的小灯做提示输出。

/血液信息转换

void blood\_data\_translate(void)

{

//缓冲区写入结束

if(g\_fft\_index>=FFT\_N)

{

g\_fft\_index = 0;

//数据更新标志位

g\_blooddata.display = 1;

//快速傅里叶变换

FFT(s1);

FFT(s2);

//解平方

for(int i = 0;i < FFT\_N;i++)

{

s1[i].real=sqrtf(s1[i].real\*s1[i].real+s1[i].imag\*s1[i].imag);

s2[i].real=sqrtf(s2[i].real\*s2[i].real+s2[i].imag\*s2[i].imag);

}

//读取峰值点 结果的物理意义为

uint16\_t s1\_max\_index = find\_max\_num\_index(s1, 60);

uint16\_t s2\_max\_index = find\_max\_num\_index(s2, 60);

//检查HbO2和Hb的变化频率是否一致

if(s1\_max\_index == s2\_max\_index)

{

//心率计算

uint16\_t Heart\_Rate = 60 \* SAMPLES\_PER\_SECOND \*

s2\_max\_index / FFT\_N;

g\_blooddata.heart = Heart\_Rate - 10;

//血氧含量计算

float sp02\_num = (s2[s1\_max\_index].real \* s1[0].real)

/(s1[s1\_max\_index].real \* s2[0].real);

sp02\_num = (1 - sp02\_num) \* SAMPLES\_PER\_SECOND + CORRECTED\_VALUE;

g\_blooddata.SpO2 = sp02\_num;

//测温并串口输出

short temp = DS18B20\_Get\_Temp();

double f\_temp = (temp / 10.0) + ((temp % 10) / 10.0);

if((f\_temp > 0.0 && f\_temp < 50.0) && (g\_blooddata.SpO2 > 0.0))

printf("{%d,%.1f,%.1f}", g\_blooddata.heart, g\_blooddata.SpO2, f\_temp);

//状态正常

g\_blooddata.state = BLD\_NORMAL;

}

else //数据发生异常

{

g\_blooddata.heart = 0;

g\_blooddata.SpO2 = 0;

g\_blooddata.state = BLD\_ERROR;

}

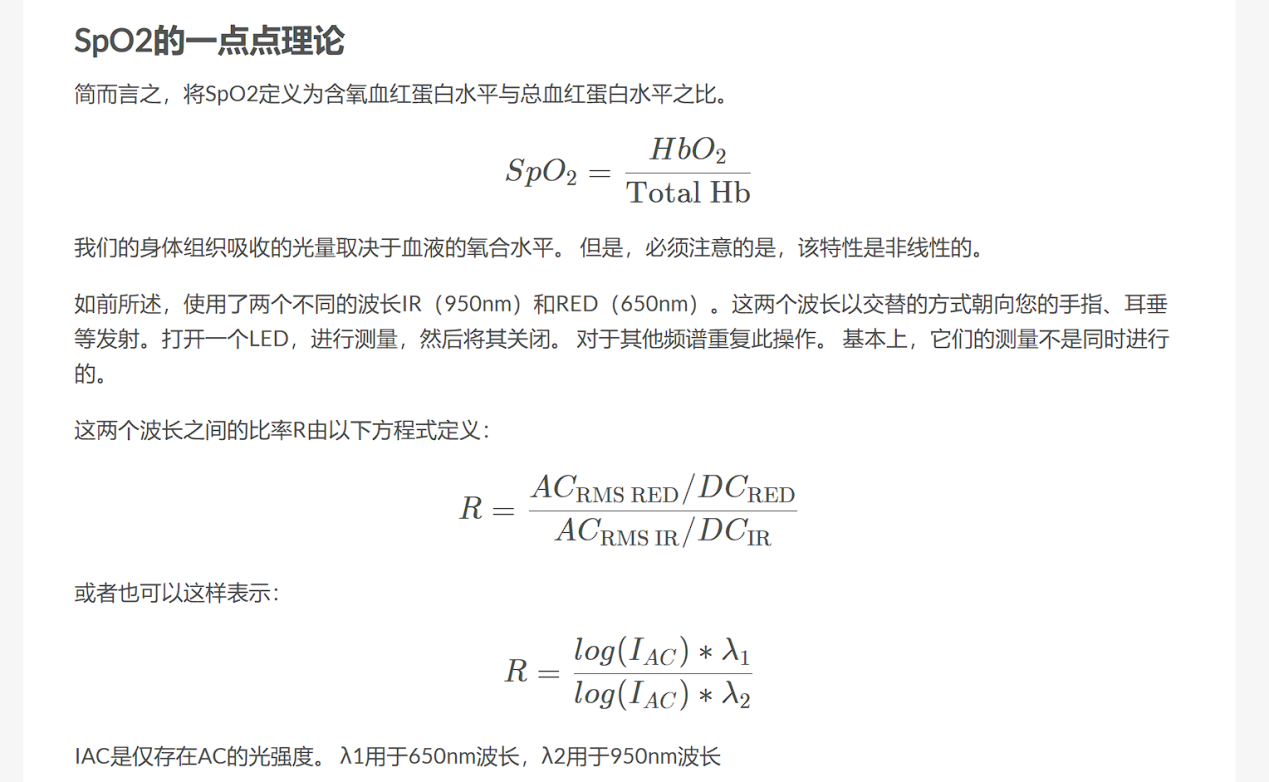
//结束变换显示

}

}

通过采集足够的样本数据，进行快速傅里叶变换（FFT），将采集到的交流电压信号转换为频域信号，同时采集两个样本的最高频率点进行比较，当脉搏和血氧的变化频率一致，我们才认为它的值正确，才会开始读值（因此在接触不好或者别的情况时，信息的输出很慢），否则会重新开始采样测量。

只有在他们变化频率一致时，才开始计算心率和血氧饱和度



为了保证串口发送时数据一致方面读取，且不繁忙占用串口影响ESP8266端的接收处理，所只有在采集血氧和心率成功时才会开始采集温度，然后合并在一起后从串口发送，ESP8266端接收到数据后，利用MQTT协议上报到服务器主题。