第1章STM32 开发

本章节主要用于学习 STM32 的总线通信，和外设数据处理功能。主要包含了：

1、 stm32 开发环境搭建
2、 串口通信实验
3、 stm32 单总线实验(温湿度)
4、 stm32 传感器数据采集实验(火焰)
5、 stm32 传感器数据采集实验(红外人体)
6、 stm32 传感器控制实验(LED 灯光控制)
7、 stm32 OLE 显示实验
8、 stm32 实时操作系统

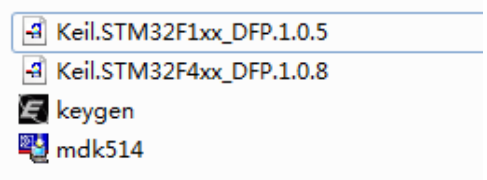
1.1 stm32 开发环境搭建

1.1.1 KEIL5 软件获取

要在电脑上成功安装 KEIL5，首先必须要有安装包，我们可以通过万能的百度，搜索关键字“KEIL5 下载”，上面会有很多下载渠道，也可以在 KEIL 的官网上下载：https://www.keil.com/download/product/，打开界面如图所示。不过我们光盘内已经给大家提供了下载链接，在光盘“开发工具\KEIL5 软件”内，大家直接下载即可，省去了查找下载的时间。我们使用 KEIL5 是 5.14 版本，如果后面出了更高的版本选择性升级即可，不过也没有必要什么都追求最新的，用习惯了一个软件就行。

KEIL5 下载界面

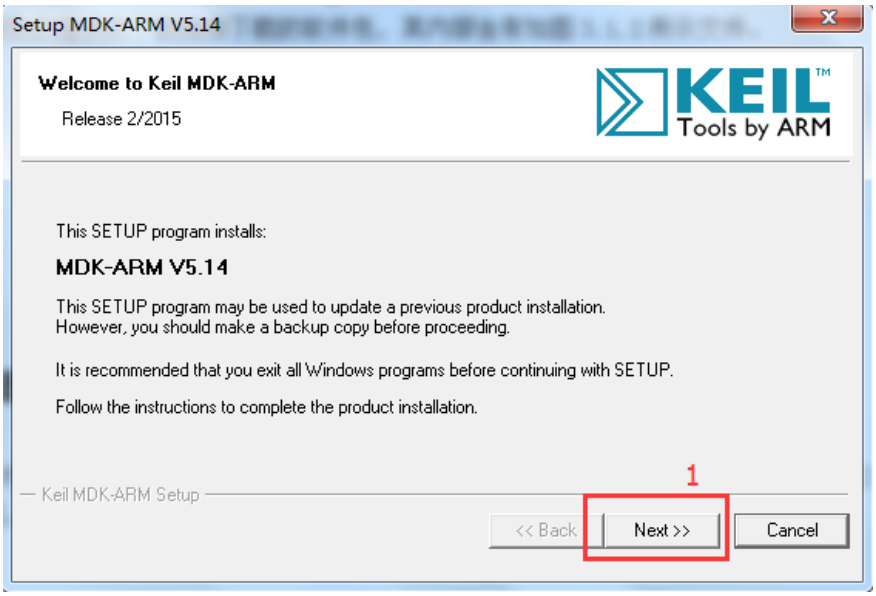
使用我们提供的下载链接下载的软件包，其内部含有如图所示文件。



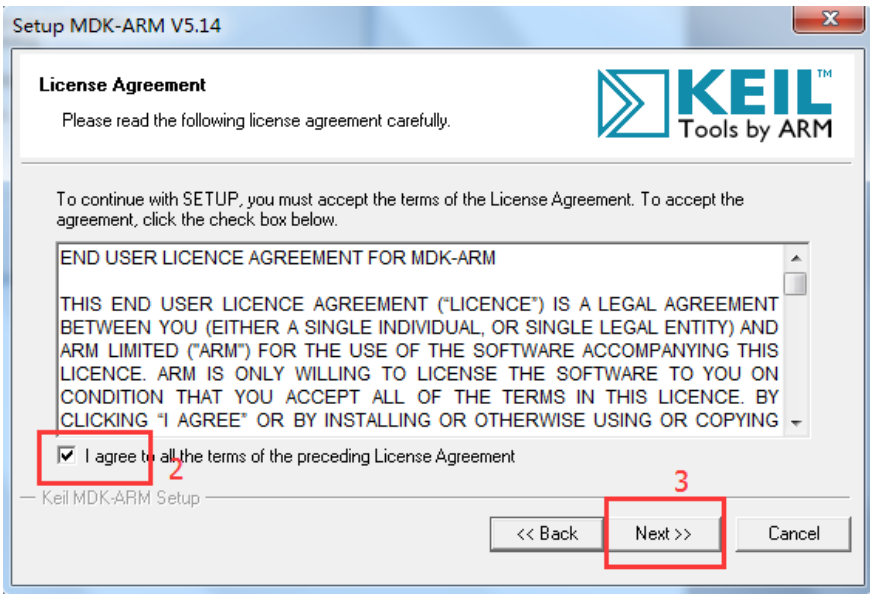
KEIL5 软件包内容

1.1.2 KEIL5 软件安装

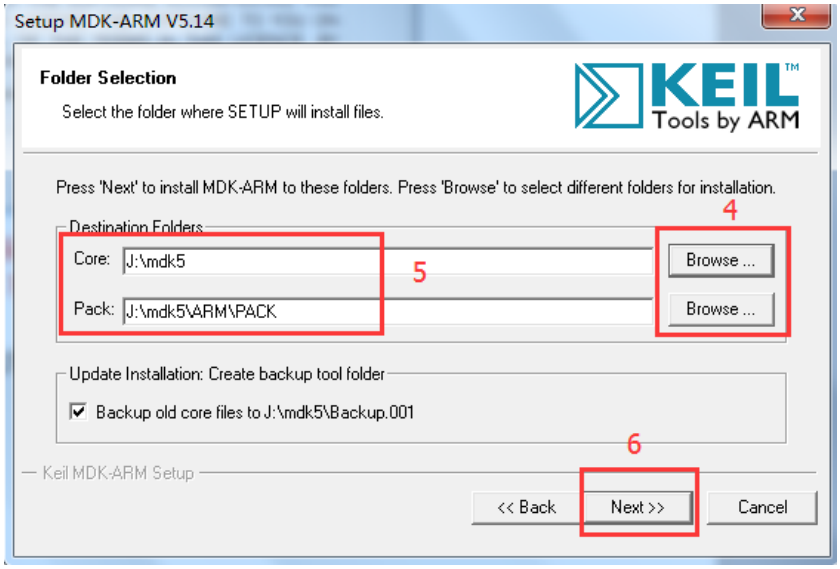
软件包下载完成之后，我们双击 mdk514.exe 这个应用程序，弹出如下所示对话框。



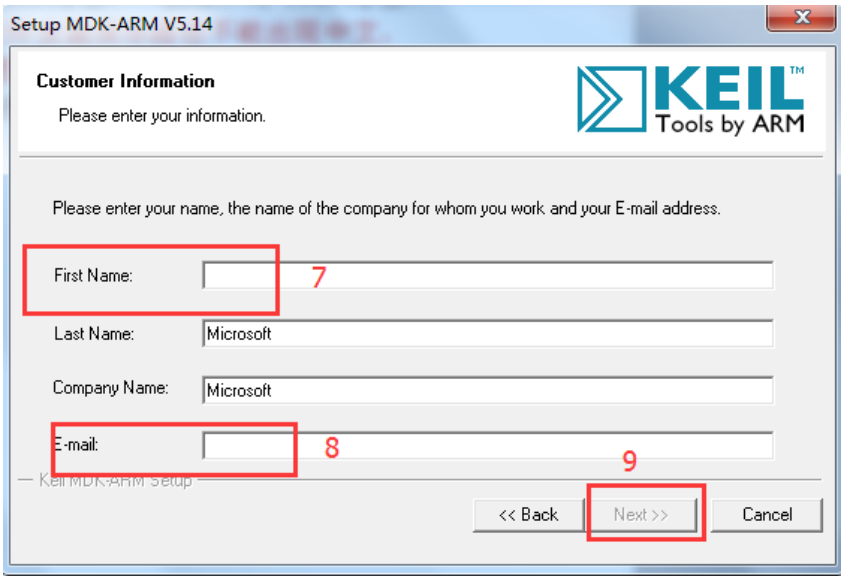
点击 Next 按钮。弹出如下对话框。



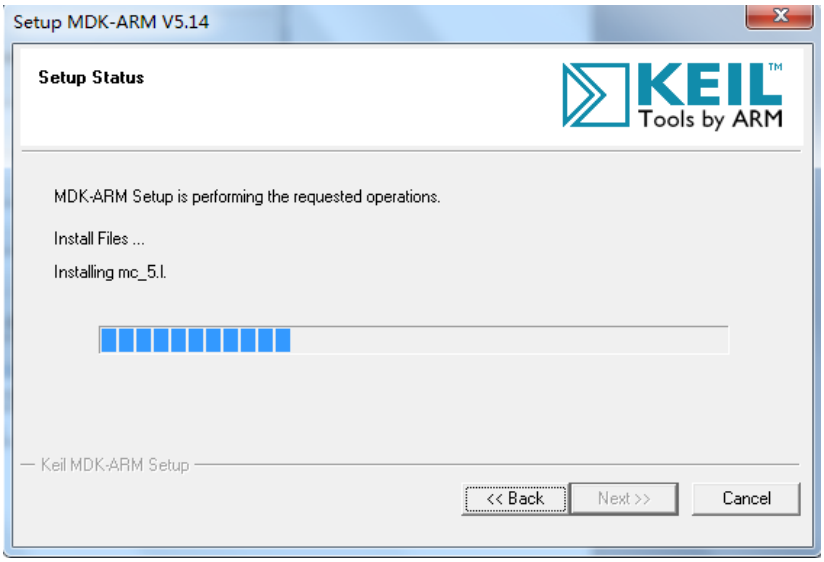
在红框 2 中勾选上，点击 Next 按钮。弹出如下对话框。



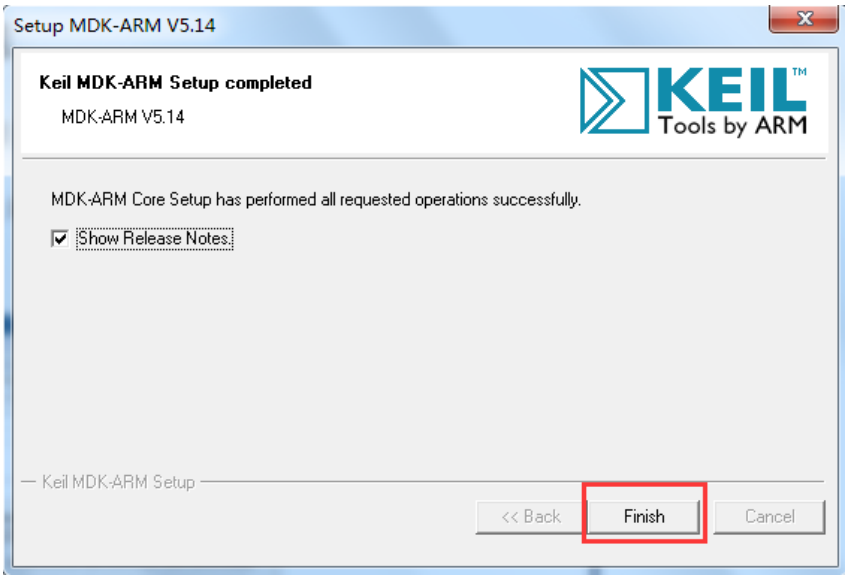
点击红框 4 选择软件安装的 code 路径和 pack 路径，一般选择好 code 路径，pack 路径自动就会出现。特别要注意：（1）软件安装保存路径不能出现中文，否则会出现很多奇怪的错误，到时候很难找问题。（2）不要将 KEIL5 软件和 KEIL4 或者 51 的 KEIL 安装在一个文件夹内。然后点击 Next。弹出如下对话框。



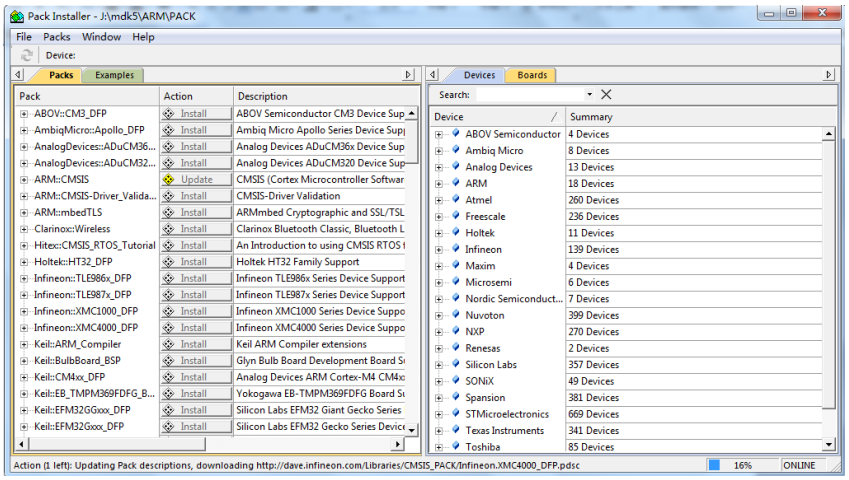
在红色框 7 和 8 中我们随便输入一些东西，我们这里输入数字 11，当然也可以输入空格，但是一定要输入，否则红色框 9 就一直是灰色状态，输入完成后点击 Next，弹出如下对话框。



说明 KEIL5 软件正在安装，只要等一段时间即可，安装完成以后会出现如下界面。

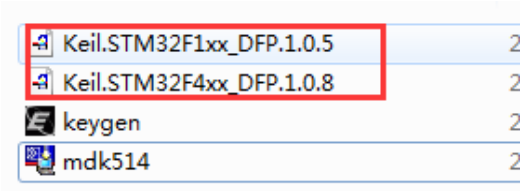


点击 Finsh 按钮弹出如下对话框，此对话框是说安装 STM32 芯片包。我们后面就会手动安装，所以这步直接关掉。



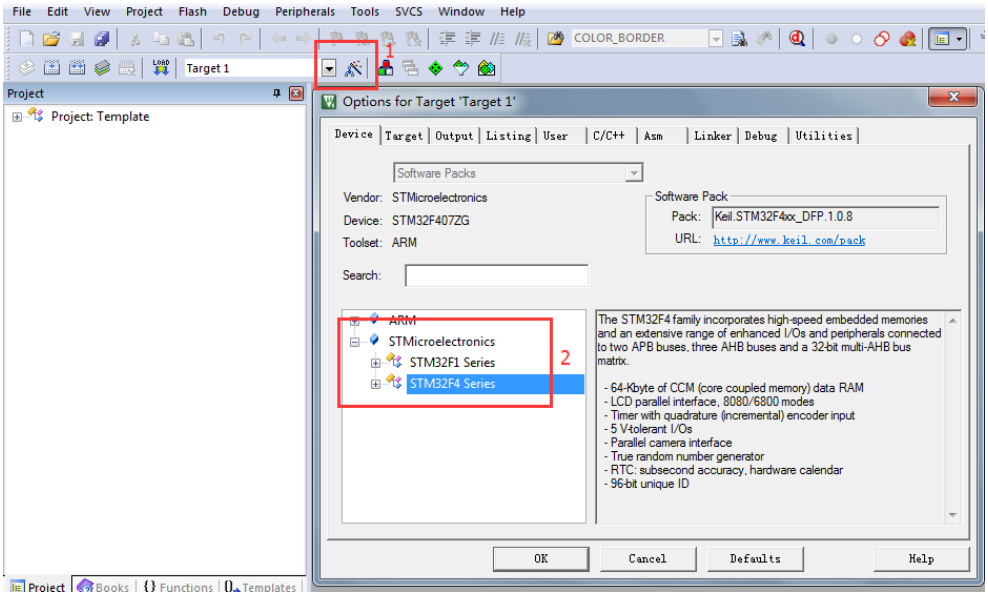
1.1.3 安装 STM32 芯片包

上一步安装完成并不是真正安装好，和以前安装 KEIL4 等软件不同的是，KEIL5 需要单独安装芯片包，否则无法选择芯片类型。STM32 芯片包需要去 KEIL 的官网下载，有 F0/1/2/6/4/7 这几个系列，具体下载和安装哪个系列的包，要看你的芯片型号。我们给大家的 KEIL5 软件内提供了 STM32F1 和 F4 的芯片包。如下所示。大家直接双击红色框内文件，安装和 KEIL5 同一目录即可。



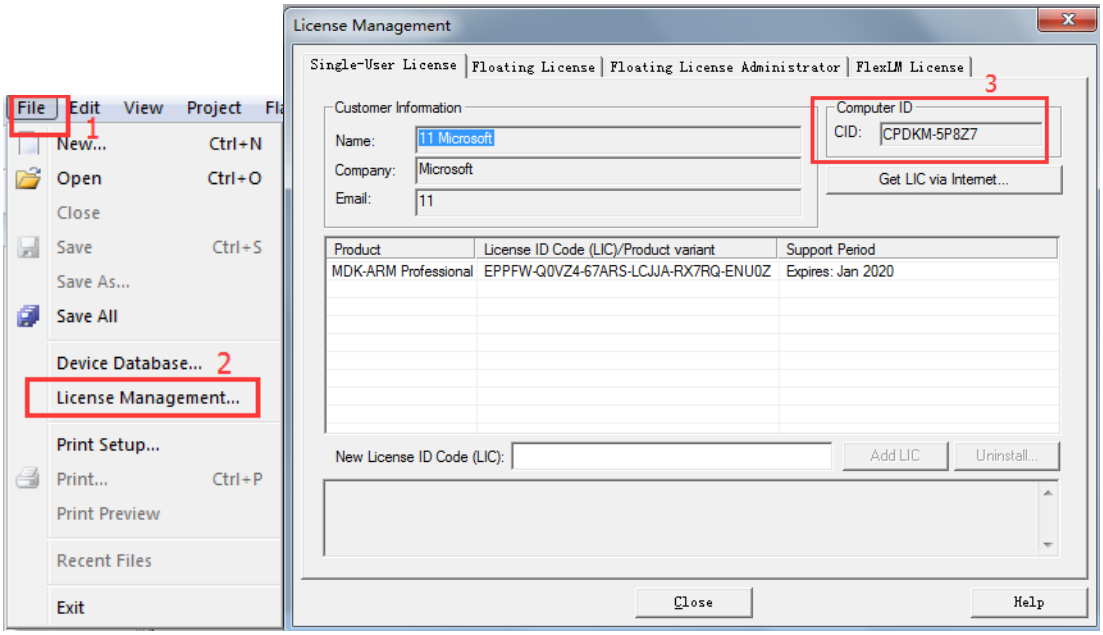
KEIL5 安装到这里电脑桌面上会有一个快捷方式，如下图所示。

我们直接双击此快捷图标，打开 KEIL5 软件，看看刚才的芯片包是否安装成功。



从上图可以看到，已经出现了我们安装的 STM32F1 和 F4 的芯片系列。后面我们就能够选择使用的 STM32 芯片型号来创建工程模板了。

安装完 KEIL5 后，我们还需要对其破解，首先打开 KEIL5 软件，点击 File-License Management..,复制红色框 3 中的 CID。操作步骤如下：



关于软件注册部分详见补充说明文档。

1.2 串口通信实验

1.实验目的

1.1 熟悉 Keil 开发环境的使用；
1.2 熟悉 STM32 通用串口 1 的使用；

2.实验设备

硬件：带有 STM32 芯片节点、电脑、J-LINK、串口线等；
软件：Keil 开发环境；

3.实验原理

3.1 STM32 串口控制寄存器说明

串口是单片机开发调试软件的重要手段，在恰当的时候调用串口输出信息能起到事半功倍的效果。STM32 最多可提供 5 路串口（本实使用的芯片是STM32F103RBT6有3个串口），有分数波特率发生器、支持同步单线通信和半双工单线通讯、支持 LIN、支持调制解调器操作、智能卡协议和 IrDA SIR ENDEC 规范（仅串口 3 支持）、具有 DMA 等。本次实验使用到的是芯片的UART2。

串口最基本的设置，就是波特率的设置。STM32 的串口使用起来还是蛮简单的，只要开启了串口时钟，并设置相应I/O口的模式，然后配置一下波特率，数据位长度，奇偶校验位等信息，就可以使用了。下面，我们就简单介绍下与串口基本配置直接相关的寄存器。

USART_SR:状态寄存器。串口的状态可以通过状态寄存器 USART_SR 读取。USART_SR 的各位描述如图3.1.1所示。

图3.1.1 串口状态寄存器

RXNE：读数据寄存器非空 (Read data register not empty)

当RDR移位寄存器中的数据被转移到USART_DR寄存器中，该位被硬件置位。如果

USART_CR1寄存器中的RXNEIE为1，则产生中断。对USART_DR的读操作可以

将该位清

零。RXNE位也可以通过写入0来清除，只有在多缓存通讯中才推荐这种清除程序。

0：数据没有收到；

1：收到数据，可以读出

TC：发送完成（Transmission complete）

当包含有数据的一帧发送完成后，并且TXE=1时，由硬件将该位置’1’。如果USART_CR1中的TCIE为’1’，则产生中断。由软件序列清除该位(先读USART_SR，然后写入USART_DR)。TC位也可以通过写入’0’来清除，只有在多缓存通讯中才推荐这种清除程序。

0：发送还未完成；

1：发送完成

USART_DR:数据寄存器。STM32 的发送与接收是通过数据寄存器 USART_DR 来实现的，这是一个双寄存器，包含了 TDR 和 RDR。当向该寄存器写数据的时候，串口就会自动发送，当收到收据的时候，也是存在该寄存器内。该寄存器的各位描述如图3.1.2所示。

图3.1.2　串口数据寄存器

可以看出，虽然是一个 32 位寄存器，但是只用了低 9 位（DR[8：0]），其他都是保留。DR[8：0]为串口数据，包含了发送或接收的数据。由于它是由两个寄存器组成的，一个给发送用(TDR)，一个给接收用(RDR)，该寄存器兼具读和写的功能。TDR 寄存器提供了内部总线和输出移位寄存器之间的并行接口。RDR 寄存器提供了输入移位寄存器和内部总线之间的并行接口。

UART_BRR:波特比率寄存器。该寄存器主要是用来配置不同波特率的。STM32采用了分数比特率，所以STM32的串口波特率设置范围很宽，而且误差很小。

1	0	9	8	7	6	5	4	3	2	1	0	9	8	7	6
保留															
5	4	3	2	1	0										
DIV_Mantissa[11：0]												DIV_Fraction[3：0]			
w	w	w	w	w	w	w	w	w	w	w	w	w	w	w	w
位		保留位，硬件强制为 0													

31:16	
位 15:4	**DIV_Mantissa[11:0]**：USARTDIV 的整数部分 这 12 位定义了 USART 分频器除法因子(USARTDIV)的整数部分。
位 3:0	**DIV_Fraction[3:0]**：USARTDIV 的小数部分 这 4 位定义了 USART 分频器除法因子(USARTDIV)的小数部分。

图3.1.3 串口波特率寄存器

前面提到 STM32 的分数波特率概念，其实就是在这个寄存器（USART_BRR）里面体现的。

USART_BRR 的最低 4 位（位[3:0]）用来存放小数部分 DIV_Fraction，紧接着的 12 位（位[15：4]）用来存放整数部分 DIV_Mantissa，最高 16 位未使用。这里，我们简单介绍一下波特率的计算，STM32 的串口波特率计算公式如下：

$$\text{Tx / Rx 波特率} = \frac{f_{PCLKx}}{(16 * USARTDIV)}$$

分数波特率的计算请详细参考技术手册P524。

UART_CR1:串口控制寄存器。该寄存器用于STM32的串口工作时的校验位，数据位等的设置及。该寄存器的描述如图3.1.4下：

地址偏移：0x0C
复位值：0x0000

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
保留		UE	M	WAKE	PCE	PS	PEIE	TXEIE	TCIE	RXNEIE	IDLEIE	TE	RE	RWU	SBK
res		rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

图3.1.4 串口控制寄存器

该寄存器中常用的几位如下介绍如下：

位 31:14	保留位，硬件强制为 0。
位 13	**UE**：USART 使能 (USART enable) 当该位被清零，在当前字节传输完成后 USART 的分频器和输出停止工作，以减少功耗。该位由软件设置和清零。 0：USART 分频器和输出被禁止； 1：USART 模块使能。
位 12	**M**：字长 (Word length) 该位定义了数据字的长度，由软件对其设置和清零 0：一个起始位，8 个数据位，n 个停止位； 1：一个起始位，9 个数据位，n 个停止位。 注意：在数据传输过程中(发送或者接收时)，不能修改这个位。

位 11	**WAKE**：唤醒的方法 (Wakeup method) 这位决定了把 USART 唤醒的方法，由软件对该位设置和清零。 0：被空闲总线唤醒； 1：被地址标记唤醒。
位 10	**PCE**：检验控制使能 (Parity control enable) 用该位选择是否进行硬件校验控制(对于发送来说就是校验位的产生；对于接收来说就是校验位的检测)。当使能了该位，在发送数据的最高位(如果 M=1，最高位就是第 9 位；如果 M=0，最高位就是第 8 位)插入校验位；对接收到的数据检查其校验位。软件对它置'1'或清'0'。一旦设置了该位，当前字节传输完成后，校验控制才生效。 0：禁止校验控制； 1：使能校验控制。
位 9	**PS**：校验选择 (Parity selection) 当校验控制使能后，该位用来选择是采用偶校验还是奇校验。软件对它置'1'或清'0'。当前字节传输完成后，该选择生效。 0：偶校验； 1：奇校验。
位 8	**PEIE**：PE 中断使能 (PE interrupt enable) 该位由软件设置或清除。 0：禁止产生中断； 1：当 USART_SR 中的 PE 为'1'时，产生 USART 中断。
位 7	**TXEIE**：发送缓冲区空中断使能 (TXE interrupt enable) 该位由软件设置或清除。 0：禁止产生中断； 1：当 USART_SR 中的 TXE 为'1'时，产生 USART 中断。
位 6	**TCIE**：发送完成中断使能 (Transmission complete interrupt enable) 该位由软件设置或清除。 0：禁止产生中断； 1：当 USART_SR 中的 TC 为'1'时，产生 USART 中断。
位 5	**RXNEIE**：接收缓冲区非空中断使能 (RXNE interrupt enable) 该位由软件设置或清除。 0：禁止产生中断； 1：当 USART_SR 中的 ORE 或者 RXNE 为'1'时，产生 USART 中断。
位 4	**IDLEIE**：IDLE 中断使能 (IDLE interrupt enable) 该位由软件设置或清除。 0：禁止产生中断； 1：当 USART_SR 中的 IDLE 为'1'时，产生 USART 中断。
位 3	**TE**：发送使能 (Transmitter enable) 该位使能发送器。该位由软件设置或清除。 0：禁止发送； 1：使能发送。

	注意：1．在数据传输过程中，除了在智能卡模式下，如果 TE 位上有个 0 脉冲(即设置为'0'之后再设置为'1')，会在当前数据字传输完成后，发送一个"前导符"(空闲总线)。 2．当 TE 被设置后，在真正发送开始之前，有一个比特时间的延迟。
位 2	**RE**：接收使能 (Receiver enable) 该位由软件设置或清除。 0：禁止接收； 1：使能接收，并开始搜寻 RX 引脚上的起始位。
位 1	**RWU**：接收唤醒 (Receiver wakeup) 该位用来决定是否把 USART 置于静默模式。该位由软件设置或清除。当唤醒序列到来时，硬件也会将其清零。 0：接收器处于正常工作模式； 1：接收器处于静默模式。 注意：1．在把 USART 置于静默模式(设置 RWU 位)之前，USART 要已经先接收了一个数据字节。否则在静默模式下，不能被空闲总线检测唤醒。 2．当配置成地址标记检测唤醒(WAKE 位=1)，在 RXNE 位被置位时，不能用软件修改 RWU 位。
位 0	**SBK**：发送断开帧 (Send break) 使用该位来发送断开字符。该位可以由软件设置或清除。操作过程应该是软件设置位它，然后在断开帧的停止位时，由硬件将该位复位。 0：没有发送断开字符； 1：将要发送断开字符。

关于串口更详细的介绍，请参考《STM32 中文参考手册_V1.0》第 540 页。

3.2 硬件原理设计

该实验的硬件电路设计如图

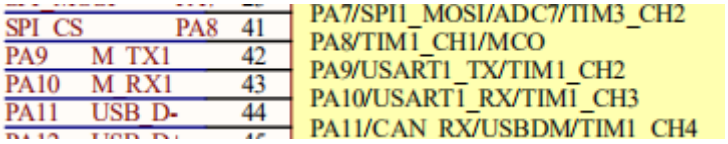


图3.2.1 STM32串口1原理图

从原理图可以看出，串口1是由芯片的PA9及PA10两个复用管脚完成的。

3.3 串口 1 在库函数下初始化过程

在本实验源代码里面的USART.C里面

第一步：启动串口1的时钟

```
RCC_APB2PeriphClockCmd(RCC_APB2Periph_USART1,ENABLE); //开启串口1的时钟
```

第二步：配置串口1复位管脚PA9、PA10

```
GPIO_InitStructure_PA9.GPIO_Pin = GPIO_Pin_9;
GPIO_InitStructure_PA9.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure_PA9.GPIO_Mode = GPIO_Mode_AF_PP;
GPIO_Init(GPIOA,&GPIO_InitStructure_PA9);

GPIO_InitStructure_PA10.GPIO_Pin = GPIO_Pin_10;
GPIO_InitStructure_PA10.GPIO_Speed = GPIO_Speed_50MHz;
GPIO_InitStructure_PA10.GPIO_Mode = GPIO_Mode_IN_FLOATING;
GPIO_Init(GPIOA,&GPIO_InitStructure_PA10);
```

这一配置可以参考技术手册P110，里面有说明串口复用时候的设置

第三步：复位串口1

```
USART_DeInit(USART1);
```

第四步：设置串口1的串口通信参数，波特率设为115200，数据为8，不检验，1停止位

```
USART1_InitStructure.USART_BaudRate = 115200;
USART1_InitStructure.USART_HardwareFlowControl = USART_HardwareFlowControl_None;
USART1_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
USART1_InitStructure.USART_Parity = USART_Parity_No ;
USART1_InitStructure.USART_StopBits = USART_StopBits_1;
USART1_InitStructure.USART_WordLength = USART_WordLength_8b;
USART_Init(USART1,&USART1_InitStructure);
```

第五步：设置串口1的中断优先级

```
NVIC_InitStructure_USART1.NVIC_IRQChannel = USART1_IRQn;                    //设置串口1的中断
NVIC_InitStructure_USART1.NVIC_IRQChannelPreemptionPriority = 3; //设置抢占优先级为3
NVIC_InitStructure_USART1.NVIC_IRQChannelSubPriority = 2;          //设置子优先级为2
NVIC_InitStructure_USART1.NVIC_IRQChannelCmd = ENABLE;             //使能中断通道
NVIC_Init(&NVIC_InitStructure_USART1);                                      //初始化中断优先级
```

第六步：使能串口1

```
USART_Cmd(USART1,ENABLE);
```

第七步：使能串口1的接收中断

```
USART_ITConfig(USART1,USART_IT_RXNE,ENABLE);
```

第八步：写串口 1 中断复位函数

```
void USART1_IRQHandler(void)
{
  if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)        //判定是否有接收到数据
     {
        USART1_RX_BUF[USART1_WR ++] =USART_ReceiveData(USART1);//有数据，那么放在缓存里
        USART1_RX_flag=1;                                                    //标志为1，表示有接收到数据
     }
}
```

4.实验步骤

4.1 编译并将程序下载到节点

在 KEIL 开发环境下打开**串口通信实验\RVMDK** 下的 Test.uvprojx 工程，编译并下载到的节点上。

4.2 实验运行效果

将程序下载到节点板，使用配套的串口线将计算机串口与节点的 DB9 串口接头相连，如图 4.2.0 所示，**注意，将节点右下角上的三位拨打开关拨至左边**，给节点重新上电，打开**串口调试工具.exe** 程序，打开正确的端口，进行 115200-8-N-1 设置，打开串口,就可以与芯片进行数据交互了如图 4.2.1 所示。我们在发送区发送 10 几个字符，正常的话芯片的串口 1 接收到数据就把数据原样返回给上位机，即串口调试助手接收到一样的数据，并在接收区显示。

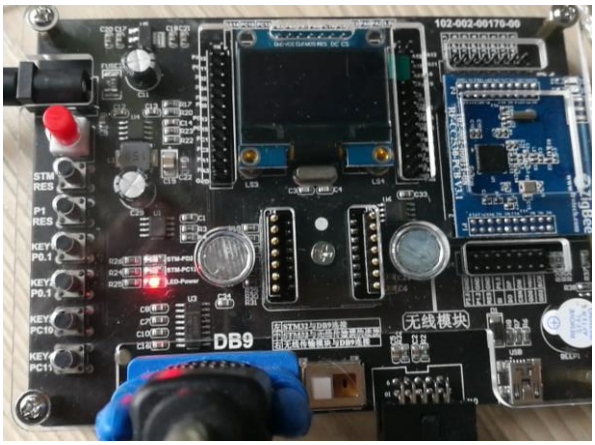


图 4.2.0 正确连接串口及白色拨打开关

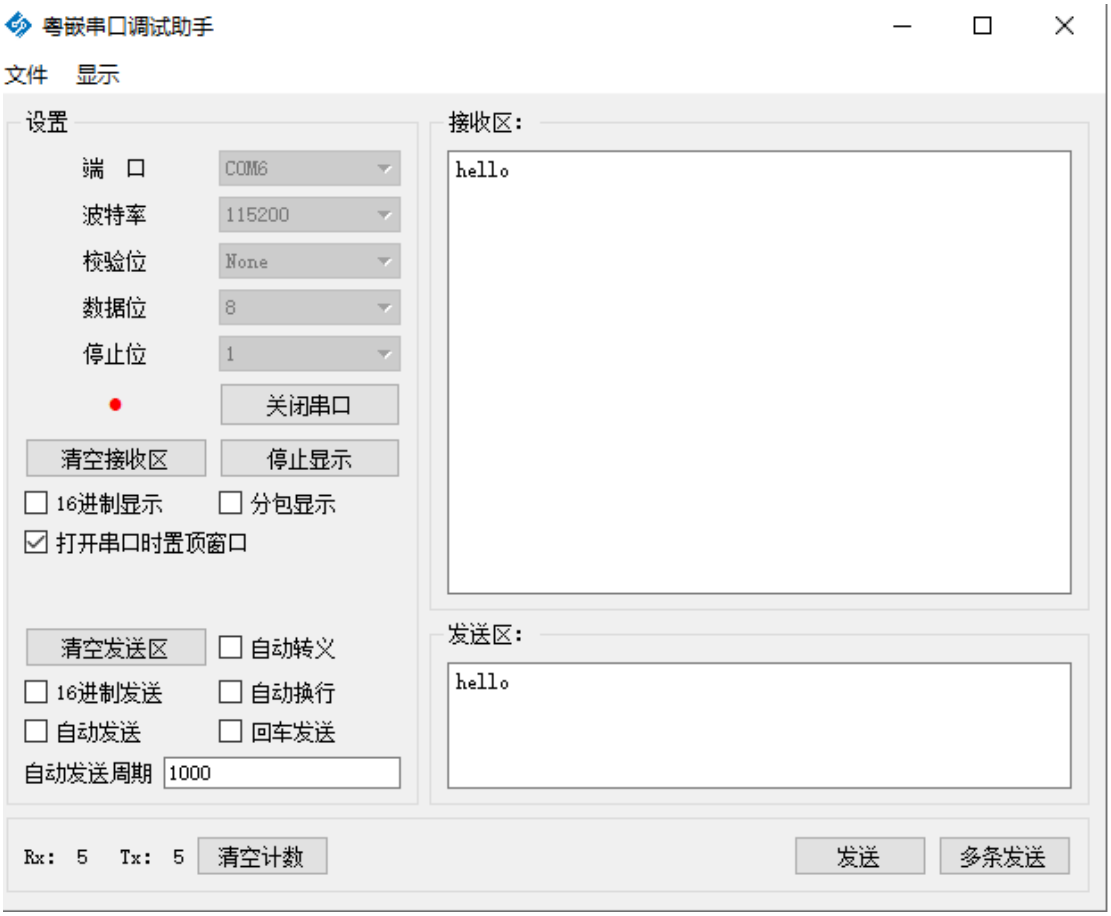


图 4.2.1 串口收发数据

注意：串口参数为 115200-8-N-1。

1.3 灯光控制实验

1.实验目的

学习使用串口控制 LED 灯的使用方法；

2.实验设备

硬件：带有 STM32 芯片节点、电脑、J-LINK、串口线等；
软件：Keil 开发环境；

3.实验原理

3.1 STM32F103 芯片 I/O 简介

STM32 系列芯片在设计时的架构共有 A、B、C、D、E 五组，每组 16 根 I/O 管脚，供用户根据需要选择。本产品中选用的 STM32 芯片具体型号为 STM32F103RBT6，采用 LQFP64 封装。该芯片的 I/O 管脚分为 A、B、C、D 四组，其中 A、B、C 三组的 16 根对应为 PA0-PA15、PB0-PB15、PC0-PC15。这些管脚可以通过相应寄存器配置成以下 8 种模式：

1. 输入浮空
2. 输入上拉
3. 输入下拉
4. 模拟输入
5. 开漏输出
6. 推挽输出
7. 推挽式复用功能
8. 开漏复用功能

每个 I/O 口可以自由编程，单个 I/O 口也必须要按照 32 位字访问。STM32 的很多 I/O 口也是 5V 兼容的，比如 PB3、PB4、PB6、PB7 等，具体哪些引脚 可以通过查看《STM32F103x8B_DS_CH_V1.0》中第 17 页到第 20 页介绍，切不可以随便接入 5V 电压和大电流给芯片管脚！

STM32 的每个 I/O 口都有 7 个寄存器来控制。他们分别是：两个 32 位端口配置寄存器 CRL 和 CRH；两个 32 位的数据寄存器 IDR 和 ODR；一个 32 位的置位/复位寄存器 BSRR；一个 16 位的复位寄存器 BRR；一个 32 位的锁存寄存器 LCKR；这里我们仅介绍常用的几个寄存器，我们编写程序时常用 4 个寄存器只有 4 个：CRL、CRH、ODR、IDR。

GPIOx_CRL:端口配置寄存器低位。该寄存器控制着每组 0 到 7 低位上 I/O 口的模式以及输出速率。

GPIOx_CRH：端口配置寄存器高位。该寄存器控制着该组 8 到 15 位高位上 I/O 口的模式以及输出速率。

STM32 的I/O口位配置如图3.1.1所示：

配置模式		CNF1	CNF0	MODE1	MODE0	PxODR寄存器
通用输出	推挽(Push-Pull)	0	0	01 10 11 见表18		0 或 1
	开漏(Open-Drain)		1			0 或 1
复用功能输出	推挽(Push-Pull)	1	0			不使用
	开漏(Open-Drain)		1			不使用
输入	模拟输入	0	0	00		不使用
	浮空输入		1			不使用
	下拉输入	1	0			0
	上拉输入					1

图3.1.1 STM32 的 IO 口位配置

STM32 的输出模式配置如图 3.1.2 所示：

MODE[1:0]	意义
00	保留
01	最大输出速度为 10MHz
10	最大输出速度为 2MHz
11	最大输出速度为 50MHz

图 3.1.2 输出模式配置

接下来我们看看端口低配置寄存器 CRL 的描述，如图3.1.3所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
CNF7[1:0]		MODE7[1:0]		CNF6[1:0]		MODE6[1:0]		CNF5[1:0]		MODE5[1:0]		CNF4[1:0]		MODE4[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
CNF3[1:0]		MODE3[1:0]		CNF2[1:0]		MODE2[1:0]		CNF1[1:0]		MODE1[1:0]		CNF0[1:0]		MODE0[1:0]	
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位 31:30	CNFy[1:0]:端口 x 配置位（y=0.....7）
27:26	软件通过这些位配置相应的 I/O 端口，请参考上图端口位配置表。
23:22	在输入模式(MODE[1:0]=00)：
19:18	00：模拟输入模式
15:14	01：浮空输入模式(复位后的状态)
11:10	10：上拉/下拉输入模式

7:6 3:2	11：保留 在输出模式(MODE[1:0]>00)： 00：通用推挽输出模式 01：通用开漏输出模式 10：复用功能推挽输出模式 11：复用功能开漏输出模式
位 29:28 25:24 21:20 17:16 13:12 9:8 5:4 1:0	**MODEy[1:0]**：端口 x 的模式位(y = 0…7) 软件通过这些位配置相应的 I/O 端口，请参考上图端口位配置表。 00：输入模式(复位后的状态) 01：输出模式，最大速度 10MHz 10：输出模式，最大速度 2MHz 11：输出模式，最大速度 50MHz

图 3.1.3 端口低配置寄存器 CRL 各位描述

该寄存器的复位值为 0X4444 4444，从上图可以看到，复位值其实就是配置端口为浮空输入模式。从上图还可以得出：STM32 的 CRL 控制着每个 IO 端口（A~G）的低 8 位的模式。每个 IO 端口的位占用 CRL 的 4 个位，高两位为 CNF，低两位为 MODE。这里我们可以记住几个常用的配置，比如 0X0 表示模拟输入模式（ADC 用）、0X3 表示推挽输出模式（做输出口用，50M 速率）、0X8 表示上/下拉输入模式（做输入口用）、0XB 表示复用输出（使用 IO 口的第二功能，50M 速率）。

CRH 的作用和 CRL 完全一样，只是 CRL 控制的是低 8 位输出口，而 CRH 控制的是高 8位输出口。这里我们对 CRH 就不做详细介绍了。

下面给个实例，比如我们要设置 PORTC 的 第11 位为上拉输入，12 位为推挽输出。代码如下：

```
GPIOC->CRH&=0XFFF00FFF;  //清掉这 2 个位原来的设置，同时也不影响其他位的设置
GPIOC->CRH|=0X00038000;     //0011 1000 PC11 输入，PC12 输出
```

通过这 2 句话的配置，我们就设置了 PC11 为上拉输入，PC12 为推挽输出。

GPIOx_IDR:端口输入寄存器：该寄存器是一个端口输入数据寄存器，只用了低 16 位。该寄存器为只读寄存器，并且只能以16位的形式读出。该寄存器各位的描述如图3.1.4 所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
IDR15	IDR14	IDR13	IDR12	IDR11	IDR10	IDR9	IDR8	IDR7	IDR6	IDR5	IDR4	IDR3	IDR2	IDR1	IDR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:16	保留，始终读为0。
位15:0	IDRy[15:0]：端口输入数据(y = 0…15) 这些位为只读并只能以字(16位)的形式读出。读出的值为对应I/O口的状态。

图3.1.4 端口输入数据寄存器 IDR 各位描述

要想知道某个 IO 口的状态，你只要读这个寄存器，再看某个位的状态就可以了。使用起来是比较简单的。

GPIOx_ODR:端口输入数据寄存器。该寄存器是一个端口输出数据寄存器，也只用了低 16 位。该寄存器为可读写，从该寄存器读出来的数据可以用于判断当前 IO 口的输出状态。而向该寄存器写数据，则可以控制某个 IO 口的输出电平。该寄存器的各位描述如图3.1.5所示：

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
保留															
15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
ODR15	ODR14	ODR13	ODR12	ODR11	ODR10	ODR9	ODR8	ODR7	ODR6	ODR5	ODR4	ODR3	ODR2	ODR1	ODR0
rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw	rw

位31:16	保留，始终读为0。
位15:0	ODRy[15:0]：端口输入数据(y = 0…15) 这些位可读可写并只能以字(16位)的形式操作。 注：对GPIOx_BSRR(x = A…E)，可以分别地对各个ODR位进行独立的设置/清除。

图3.1.5 端口输出数据寄存器 ODR 各位描述

关于I/O口更详细的介绍，请参考《STM32中文参考手册_V1.0》第 105 页开始的内容。

3.2 硬件原理设计

该实验的硬件电路设计如图

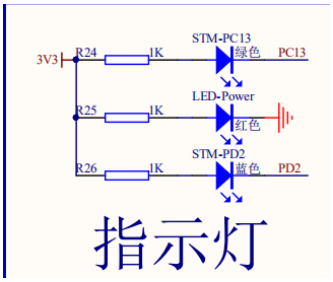


图3.2.1 LED灯原理图

从上面原理图可以看出我们的STM-PD2蓝色灯与我们的STM32芯片的PD2管脚连接着，如果PD2输出低电平，蓝色灯正向导通发光，如果PD2输出高电平，蓝色灯两边没有压降差不发光。本次实验通过对PD2管脚输出状态的改变来控制LED灯的状态。

3.3 部分代码说明

```
/**************************************
函数名称：USART1_IRQHandler(void)
函数参数：无
功能说明：串口1中断处理函数，接收的数据放在缓存USART1_RX_BUF，
          并用USART1_WR计数接收到了多少字节
函数返回：无
编写作者：FRO-PJF01
修改时间：2017.02.22
**************************************/
void USART1_IRQHandler(void)
{
  unsigned char res;
  if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)        //判定是否有接收到数据
  {
    //SART1_RX_BUF[USART1_WR ++] =USART_ReceiveData(USART1);//有数据，那么放在缓存里
    //USART1_RX_flag=1;                                      //标志为1，表示有接收到数据
    res = USART_ReceiveData(USART1);
    if(res == 'O')
    {
      LED1_ON;
    }
    else if(res == 'C')
    {
      LED1_OFF;
    }
  }
}
```

在串口中断中判断接收的串口数据，控制 led 灯的亮灭。

4.实验步骤

4.1 编译并将程序下载到节点

在 KEIL 开发环境下打开**灯光控制实验\RVMDK** 下的 Test.uvprojx 工程，编译并下载到的节点上。

4.2 实验运行效果

将程序下载到节点板，使用配套的串口线将计算机串口与节点的 DB9 串口接头相连，如图 4.2.0 所示，**注意，将节点右下角上的三位拨打开关拨至左边**，给节点重新上电，打开**串口调试工具.exe** 程序，打开正确的端口，进行 115200-8-N-1 设置，打开串口。

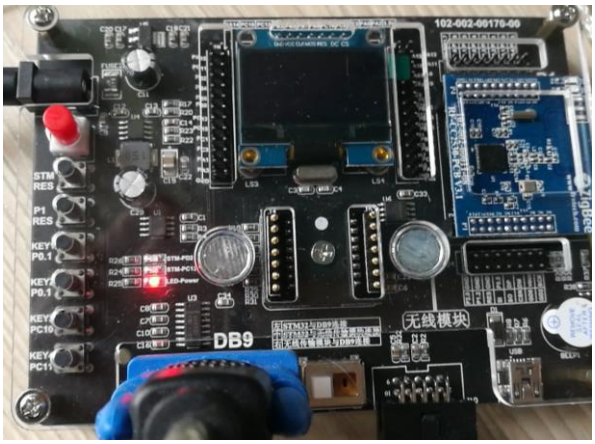


图 4.2.0 正确连接串口及白色拨打开关

往串口助手中发送‘0’，LED 灯亮；发送‘C’，LED 灯灭。

1.4 stm32 单总线实验(温湿度)

1.实验目的

学习温湿度传感器的使用方法；

2.实验设备

硬件：带有 STM32 芯片节点、电脑、J-LINK、串口线等；
软件：Keil 开发环境；

3.实验原理

3.1 AM2321 介绍

AM2321 湿敏电容数字温湿度模块是一款含有已校准数字信号输出的温湿度复合传感器。它应用专用的数字模块采集技术和温湿度传感技术，确保产品具有极高的可靠性与卓越的长期稳定性。传感器包括一个电容式感湿元件和一个高精度测温元件，并与一个高性能 8 位单片机相连接。因此该产品具有品质卓越、超快响应、抗干扰能力强、性价比极高等优点。每个传感器都在极为精确的湿度校验室中进行校准。校准系数以程序的形式储存在单片机中，传感器内部在检测信号的处理过程中要调用这些校准系数。标准单总线接口，使系统集成变得简易快捷。超小的体积、极低的功耗，信号传输距离可达 20 米以上。产品为 3 引线（单总线接口）连接方便。AM2321 的响应时间约 2S，这比一般的温湿度传感器的响应时间要快。

3.2 温湿度传感器 AM2321 的电路原理图

AM2321 温湿度传感器的电路原理如图 3.1 所示。

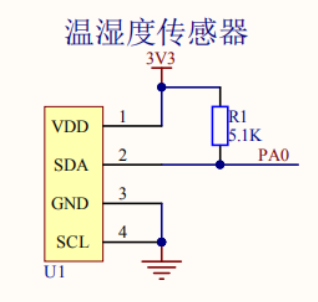


图 3.1　AM2321 温湿度传感器电路

其 SDA 口连接到 STM32 的 PA0 进行单总线通信，R1 为上拉电阻，详情见其技术手册的 P4。

3.3 AM2321 的传感器性能

AM2321 的传感器参数如图 3.2 所示。

参数	条件	min	typ	max	单位
分辨率			0.1		%RH
			16		bit
精度[1]	25℃		±2		%RH
重复性			±0.3		%RH
互换性		完全互换			
响应时间[2]	1/e(63%)		<5		S
迟滞			<0.3		%RH
漂移[3]	典型值		<0.5		%RH/yr

图 3.2　AM2321 传感器参数 1

AM2321 传感器另外一部分参数如图 3.3 所示。

参数	条件	min	typ	max	单位
分辨率			0.1		℃
			16		bit
精度			±0.5	±1	℃
量程范围		-40		80	℃
重复性			±0.2		℃
互换性		完全互换			
响应时间	1/e(63%)		<10		S
漂移			±0.3		℃/yr

图 3.3　AM2321 传感器参数 2

3.4 AM2321 的单总线通信协议

SDA 用于微处理器与 AM2321 之间的通讯和同步如图 3.4 所示， 采用单总线数据格式，一次传送 40 位数据，高位先出。具体通信时序如图 3.5 所示。

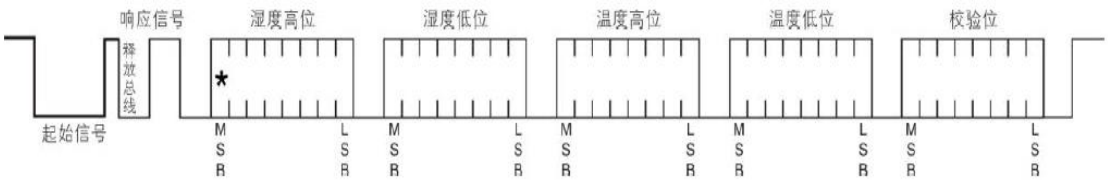


图 3.4　单总线协议时序

单总线格式定义如表 2 所示。

表 2　　单总线格式定义

名称	单总线格式定义
起始信号	微处理器把数据总线（SDA）拉低一段时间（至少 800us），通知传感器准备数据
响应信号	传感器把数据总线（SDA）拉低 80us,再拉高 80us 以响应主机的起始信号。
数据格式	收到主机起始信号后，传感器一次性从数据总线(SDA）串出 40 位数据，高位先出
湿度	湿度分辨率是 16Bit,高位在前；传感器传出的湿度值是实际湿度值的 10 倍
温度	温度分辨率是 16Bit,高位在前；传感器传出的温度值是实际温度值的 10 倍； 温度最高位（Bit15)等于 1 表示负温度，温度最高位（Bit15)等于 0 表示正温度； 温度除了最高位（Bit14~~Bib0)表示温度值。
校验位	校验位 = 湿度高位 + 湿度低位 + 温度高位 + 温度低位

总线数据计算举例说明：

如果接收到的 40 位数据如表 3 所示。

0000 0010	1001 0010	0000 0001	0000 1101	1010 0010
湿度高 8 位	湿度低 8 位	温度高 8 位	温度低 8 位	校验位

计算：

0000 0010 + 1001 0010 + 0000 0001 + 0000 1101 + 1010 0010 = 1010 0010 （校验位）

接收数据正确，则湿度及温度的值如下：

湿度：0000 0010 1001 0010 = 0x0292H = 658，即湿度 = 65.8%RH

温度：0000 0001 0000 1101 = 0x10DH = 269，即温度 = 26.9℃

单总线具体通信时序：

用户主机（MCU）发送一次起始信号（把数据总线 SDA 拉低至少 800μs）后，AM2321 从休眠模式转换到高速模式。待主机开始信号结束后，AM2321 发送响应信号，从数据总线 SDA 串行送出 40Bit 的数据，先发送字节的高位；发送的数据依次为湿度高位、湿度低位、温度高位、温度低位、校验位，发送数据结束触发一次信息采集，采集结束传感器自动转入休眠模式，直到下一次通信来临。单总线具体通信时序如图 5 所示。

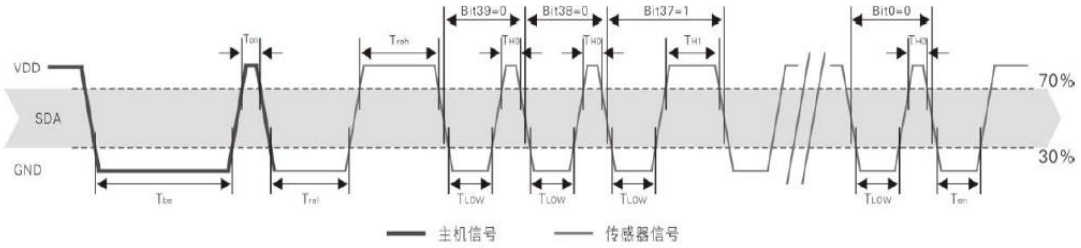


图 3.5　具体单总线通讯时序

注：主机从 AM2321 读取的温湿度数据总是前一次的测量值，如两次测量间隔时间很长，请连续读两次以第二次获得的值为实时温湿度值，同时两次读取间隔时间最小为 2S。

单总线时间表如表 4 所示。

表 4　个时间段信息

符号	参数描述	Min	Typ	Max	单位
Tbe	主机起始信号拉低时间	0.8	1	20	mS
Tgo	主机释放总线时间	20	30	200	uS
Trel	响应低电平时间	75	80	85	uS
Treh	响应高电平时间	75	80	85	uS
Tlow	信号“0”、“1”低电平时间	48	50	55	uS
Tho	信号“0”高电平时间	22	26	30	uS
Th1	信号“1”高电平时间	68	70	75	uS
Ten	传感器释放总线时间	45	50	55	uS

3.5 外部设备读取温湿度信号的步骤

步骤一：AM2321 上电后（AM2321 上电后要等待 2S 以越过不稳定状态，在此期间读取设备不能发送任何指令），测试环境温湿度数据，并记录数据，此后传感器自动转入休眠状态。AM2321 的 SDA 数据线由上拉电阻拉高一直保持高电平，此时 AM2321 的 SDA 引脚处于输入状态，时刻检测外部信号。

步骤二：微处理器的 I/O 设置为输出，同时输出低电平，且低电平保持时间不能小于 800us，典型值是拉低 1MS，然后微处理器的 I/O 设置为输入状态，释放总线，由于上拉电阻，微处理器的 I/O 即 AM2321 的 SDA 数据线也随之变高，等主机释放总线后，AM2321 发送响应信号，即输出 80 微秒的低电平作为应答信号，紧接着输出 80 微秒的高电平通知外设准备接收数据，信号传输如下图 3.6 所示。

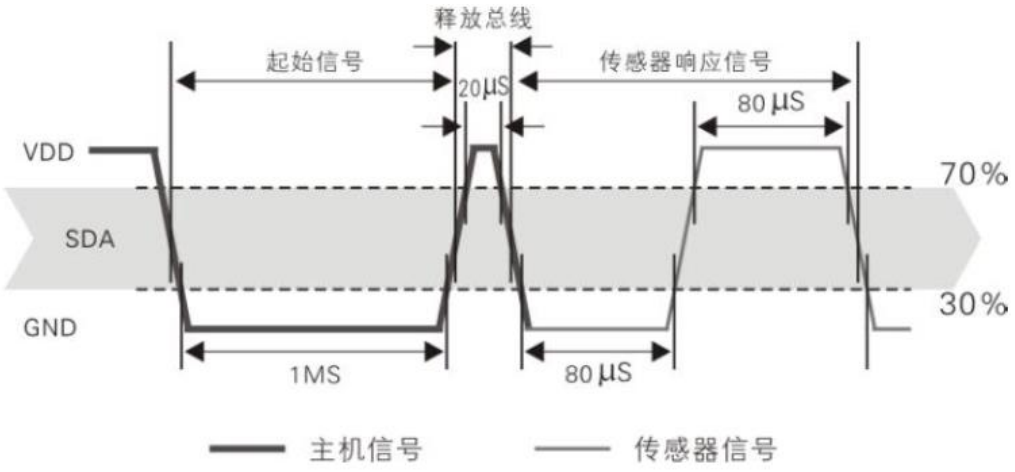


图 3.6　SDA 信号传输时序

步骤三：AM2321 发送完响应后，随后由数据总线 SDA 连续串行输出 40 位数据，微处理器根据 I/O 电平的变化接收 40 位数据。

位数据“0”的格式为：　50 微秒的低电平加 26-28 微秒的高电平；

位数据“1”的格式为：　50 微秒的低电平加 70 微秒的高电平；

位数据“0”、位数据“1”格式信号如图 3.7 所示。

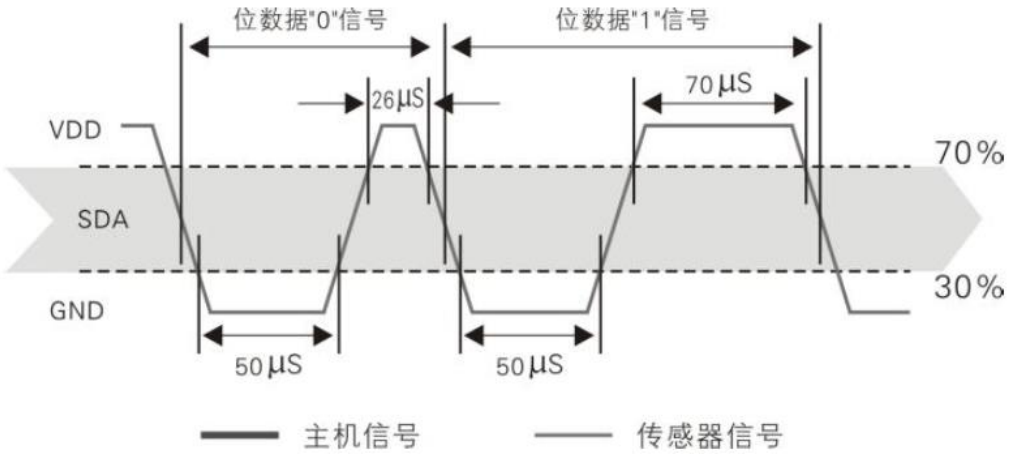


图 3.7　SDA 传输 0 和 1 时序

AM2321 的数据总线 SDA 输出 40 位数据后，继续输出低电平 50 微秒后转为输入状态，由于上拉电阻随之变为高电平。同时 AM2321 内部重测环境温湿度数据，并记录数据，测试记录结束，单片机自动进入休眠状态。单片机只有收到主机的起始信号后，才重新唤醒传感器，进入工作状态。

3.6 实验源码

```
#define SDA_H()      GPIO_SetBits(GPIOA,GPIO_Pin_0)
#define SDA_L()      GPIO_ResetBits(GPIOA,GPIO_Pin_0)
#define GET_SDA() GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0)
static uint8_t       sensor_data[5] = {0x00, 0x00, 0x00, 0x00, 0x00};
```

```
void Sensor_Init(void)
{
    GPIO_InitTypeDef     GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_Out_PP;
    GPIO_Init(GPIOA,&GPIO_InitStructure);
}

void Sensor_Pin_In(void)
{
    GPIO_InitTypeDef     GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IPU;
    GPIO_Init(GPIOA,&GPIO_InitStructure);
}

/**
 * @brief 读传感器发送的单个字节
 *
 * @return uint8_t 传感器发送的单个字节
 */
static uint8_t read_sensor_data(void)
{
    uint16_t cnt;
    uint8_t   i;
    uint8_t   buffer, tmp;
    buffer = 0;
    for (i = 0; i < 8; i++) {
        cnt = 0;
        while (!GET_SDA()) //检测上次低电平是否结束
        {
            if (++cnt >= 3000) {
                break;
            }
        }
```

```
            //延时 Min=26us Max50us 跳过数据"0" 的高电平
            delay_us(30); //延时 30us

            //判断传感器发送数据位
            tmp = 0;
            if (GET_SDA()) {
                tmp = 1;
            }
            cnt = 0;
            while (GET_SDA()) //等待高电平 结束
            {
                if (++cnt >= 2000) {
                    break;
                }
            }
            buffer <<= 1;
            buffer |= tmp;
        }
        return buffer;
}

/**
 * @brief 读传感器
 *
 * @return int32_t 0：读取成功； -1：读取失败
 */
static int32_t read_sensor(void)
{
        uint16_t overtime_cnt;
        uint8_t   i;
       Sensor_Init();
        //主机拉低(Min=800US Max=20Ms)
        SDA_L();
        delay_ms(2); //延时 2Ms

        //释放总线 延时(Min=30us Max=50us)
        SDA_H();
        delay_us(30); //延时 30us
        //主机设为输入 判断传感器响应信号
        SDA_H();
       Sensor_Pin_In();
```

```
    //判断从机是否有低电平响应信号 如不响应则跳出，响应则向下运行
    if (GET_SDA() == 0) {
        overtime_cnt = 0;
        //判断从机是否发出 80us 的低电平响应信号是否结束
        while ((!GET_SDA())) {
            if (++overtime_cnt > 3000) //防止进入死循环
            {
                return -1;
            }
        }
        overtime_cnt = 0;
        //判断从机是否发出 80us 的高电平，如发出则进入数据接收状态
        while ((GET_SDA())) {
            if (++overtime_cnt > 3000) //防止进入死循环
            {
                return -1;
            }
        }
        // 数据接收 传感器共发送 40 位数据
        // 即 5 个字节 高位先送 5 个字节分别为湿度高位 湿度低位 温度高位 温度低位
        // 校验和 校验和为：湿度高位+湿度低位+温度高位+温度低位
        for (i = 0; i < 5; i++) {
            sensor_data[i] = read_sensor_data();
        }
    } else {
        return -1; // 未收到传感器响应
    }
    return 0;
}


int main()
{
    int i;
    RCC_Configuration();
    LED_Init();
    Sensor_Init();
    USART1_Init();
    Delay_Init(72);
```

```
    while(1)
    {
        i=read_sensor();
        if(i==0)
        {


    printf("humi:%.1f,temp:%.1f\r\n",(float)((sensor_data[0]<<8)|sensor_data[1])/10,
(float)((sensor_data[2]<<8)|sensor_data[3])/10);
        }
        else
        {
            printf("error\r\n");
        }
        delay_ms(1200);
  }

}
```

4.实验步骤

4.1 编译并将程序下载到节点

在 KEIL 开发环境下打开**温湿度传感器实验\RVMDK** 下的 Test.uvprojx 工程，编译并下载到的节点上。

4.2 实验运行效果

将程序下载到节点板，使用配套的串口线将计算机串口与节点的 DB9 串口接头相连，如图 4.2.0 所示，**注意，将节点右下角上的三位拨打开关拨至左边**，给节点重新上电，打开**串口调试工具.exe** 程序，打开正确的端口，进行 115200-8-N-1 设置，打开串口,放置温湿度传感器，串口助手显示温湿度数据。

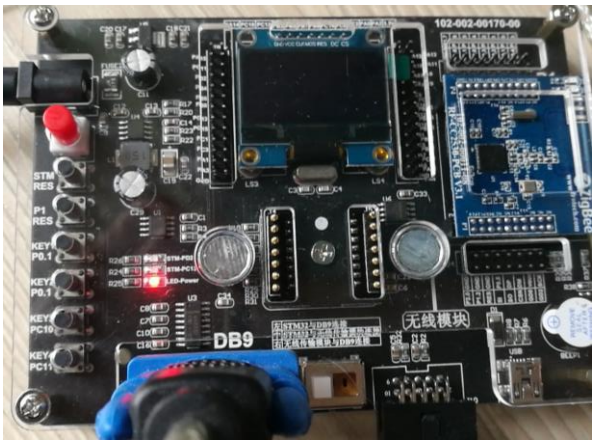


图 4.2.0 正确连接串口及白色拨打开关

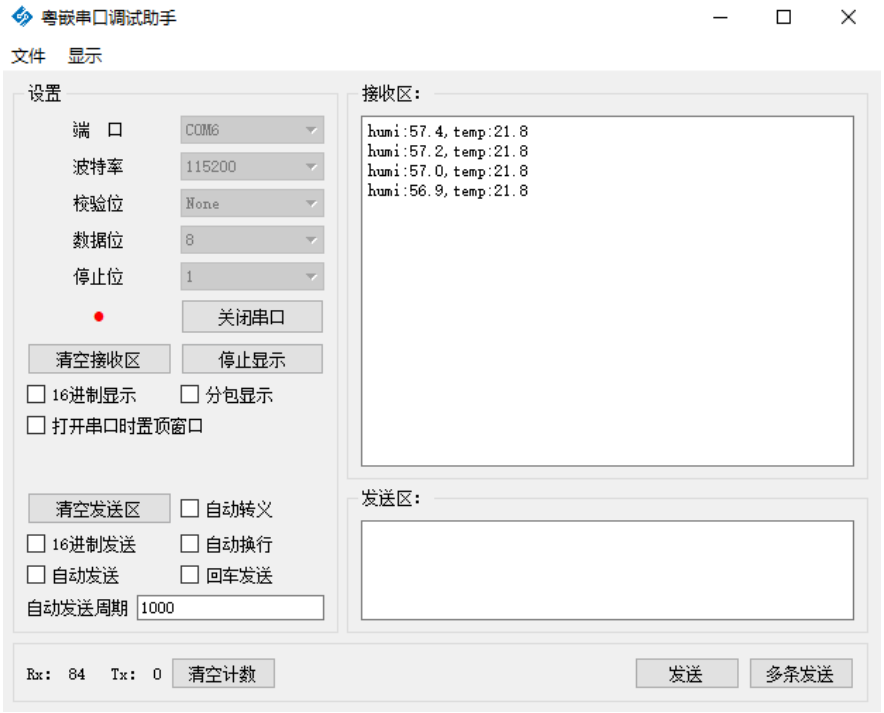


图 4.2.1 串口收发数据

注意：串口参数为 115200-8-N-1。

1.5 stm32 传感器数据采集实验(火焰)

1.实验目的

学习火焰传感器的使用方法；

2.实验设备

硬件：带有 STM32 芯片节点、电脑、J-LINK、串口线等；
软件：Keil 开发环境；

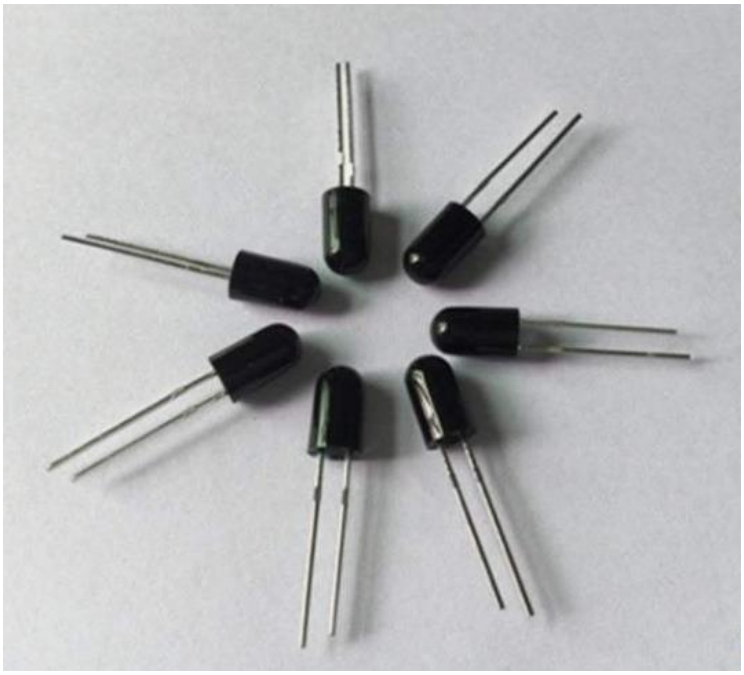
3.实验原理

➢ 红外光基本原理

我们知道，人的眼睛能看到的可见光按波长从长到短排列，依次为红、橙、黄、绿、青、蓝、紫。其中红光的波长范围为 0.62～0.76μm；紫光的波长范围为 0.38～0.46μm。比紫光波长还短的光叫紫外线，比红光波长还长的光叫红外线。红外线是波长介于微波和可见光之间的电磁波，波长在 760 纳米到 1 毫米之间，是波形比红光长的非可见光。自然界中的一切物体，只要它的温度高于绝对零度(-273)就存在分子和原子的无规则运动，其表面就会不停的辐射红外线。当然了，虽然是都辐射红外线，但是不同的物体辐射的红外强度是不一样的，而我们正是利用了这一点把红外技术应用到我们实际开发中。

➢ 火焰（红外线）接收管

红外接收管内部带了一个具有红外光敏感特征的 PN 节，属于光敏二极管，但是它只对红外光有反应。无红外光时，光敏管不导通，有红外光时，光敏管导通形成光电流，并且在一定范围内电流随着红外光的强度的增强而增大。它广泛用于各种家用电器的遥控接收器中，如音响、彩色电视机、空调器、VCD 视盘机、DVD 视盘机以及录像机等。能很好地接收红外发光二极管发射的波长为940nm的红外光信号，而对于其他波长的光线则不能接收，因而保证了接收的准确性和灵敏度。



➢ 传感器参数

1) 可以检测火焰或者波长在 760 纳米～1100 纳米范围内的光源

2) 探测角度 60 度左右，对火焰光谱特别灵敏

3) 灵敏度可调（图中蓝色数字电位器调节）

4) 对火焰的探测距离：跟灵敏度和火焰强度有关，一般 1m 以内适用（以打火机火焰测试，半米内能够触发传感器）

5) 比较器输出，信号干净，波形好，驱动能力强，超过 15mA

6) 工作电压 2.3V-5V

7) 输出形式 ：a 能够输出数字信号（高低电平），易于使用

b 能够输出模拟信号（电压信号），适合高精度的场合

8) 设有固定螺栓孔，方便安装

9) 小板 PCB 尺寸：2.2cm x 1.4cm

使用宽电压 LM393 比较器

3.2 火焰传感器的电路原理图

火焰传感器的电路原理如图 3.1 所示。

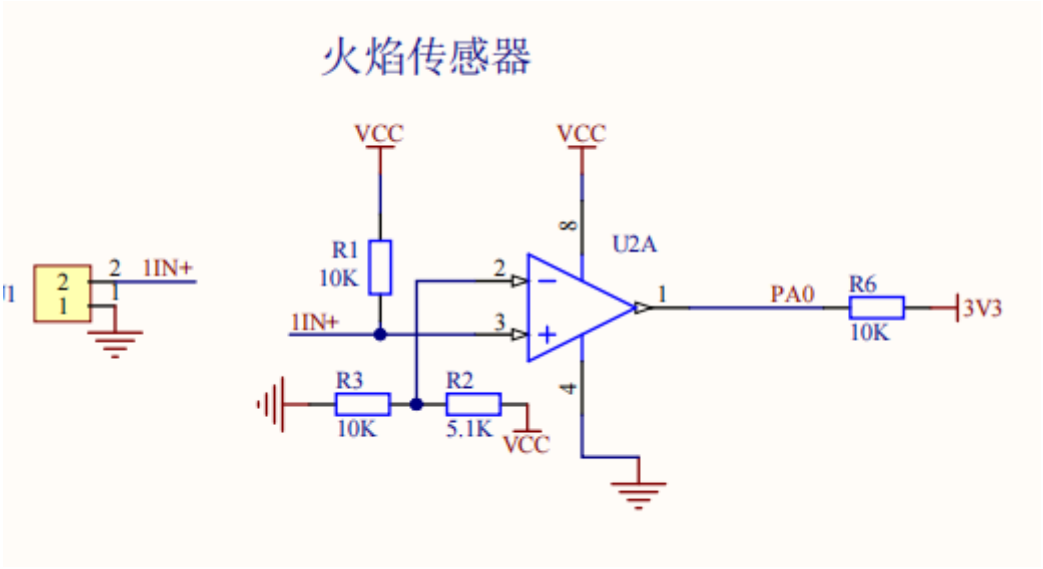


图 3.1　火焰感器电路

其数据口连接到 STM32 的 PA0 进行电平检测。

3.3 实验源码

```
#include "stm32f10x.h"

#include "Delay.h"
#include "LED.h"
#include "Time.h"
#include "USART.h"

void RCC_Configuration(void);


#define GET_SDA() GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0)


void Sensor_Init(void)
{
    GPIO_InitTypeDef     GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
```

```
    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IPU;
    GPIO_Init(GPIOA,&GPIO_InitStructure);
}

int main()
{
    RCC_Configuration();
    LED_Init();
    Sensor_Init();
    USART1_Init();
    Delay_Init(72);
    while(1)
    {
        if(GET_SDA()==1)
        {

            printf("没火\r\n");
        }
        else
        {
            printf("有火\r\n");
        }
        delay_ms(1000);
  }

}



/**************************************
函数名称: RCC_Configuration(void)
函数参数：无
功能说明：STM32 端口时钟初始化
函数返回：无
编写作者：FRO-PJF01
修改时间：2017.02.22
**************************************/
void RCC_Configuration(void)
{
  SystemInit();
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
```

```
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC
                         | RCC_APB2Periph_GPIOD| RCC_APB2Periph_GPIOE , ENABLE);
}
```

4.实验步骤

4.1 编译并将程序下载到节点

在 KEIL 开发环境下打开火焰**传感器实验\RVMDK** 下的 Test.uvprojx 工程，编译并下载到的节点上。

4.2 实验运行效果

将程序下载到节点板，使用配套的串口线将计算机串口与节点的 DB9 串口接头相连，如图 4.2.0 所示，**注意，将节点右下角上的三位拨打开关拨至左边**，给节点重新上电，打开**串口调试工具.exe** 程序，打开正确的端口，进行 115200-8-N-1 设置，打开串口,放置火焰传感器，串口助手显示火焰传感器是否有火焰。

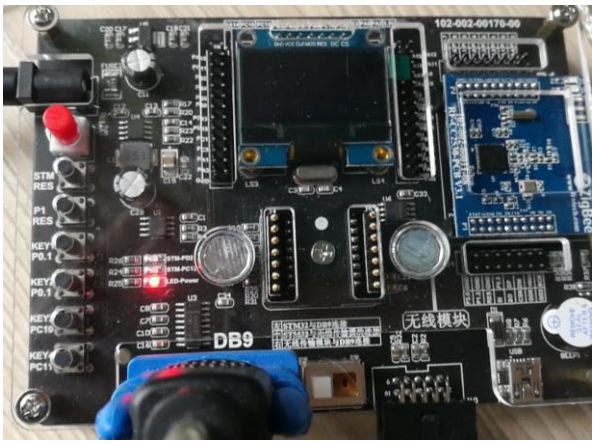


图 4.2.0 正确连接串口及白色拨打开关

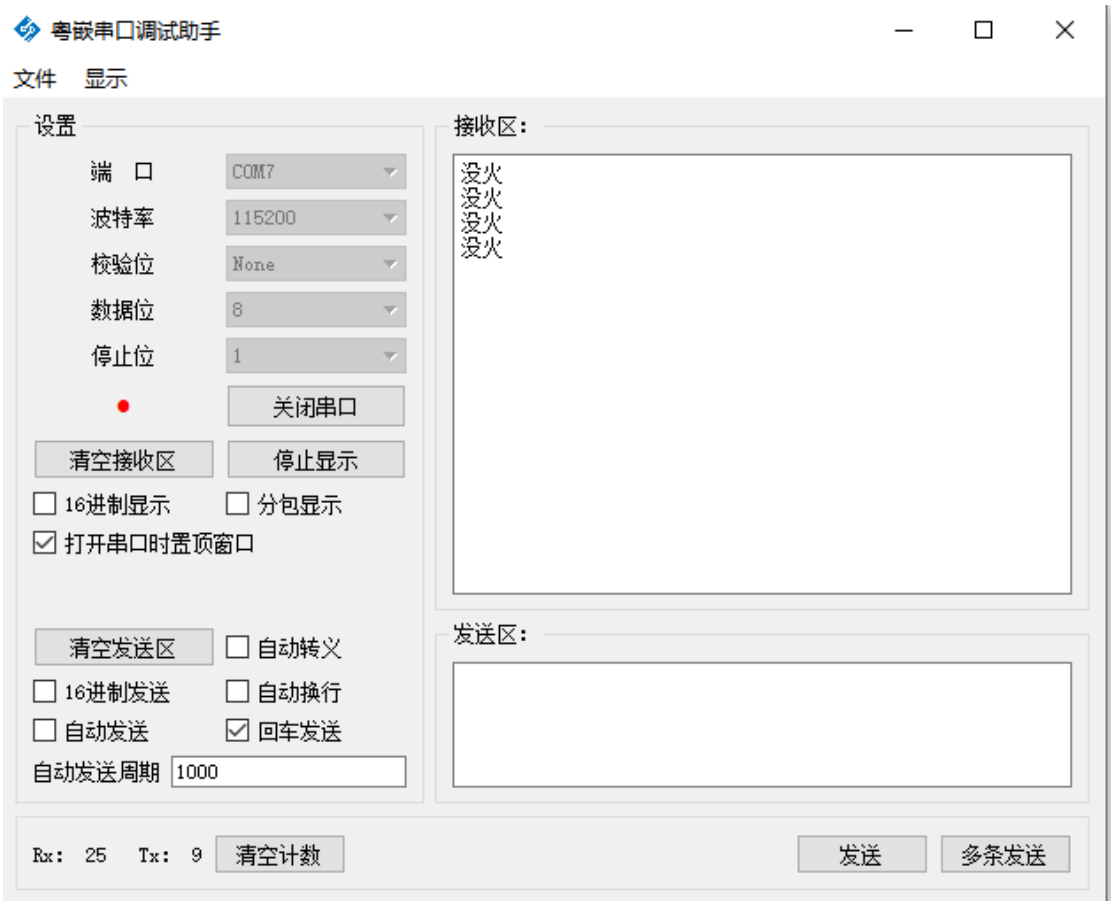


图 4.2.1 串口收发数据

注意：串口参数为 115200-8-N-1。

1.6 stm32 传感器数据采集实验(红外人体)

1.实验目的

学习人体红外传感器的使用方法；

2.实验设备

硬件：带有 STM32 芯片节点、电脑、J-LINK、串口线等；
软件：Keil 开发环境；

3.实验原理

3.1 人体红外传感器介绍

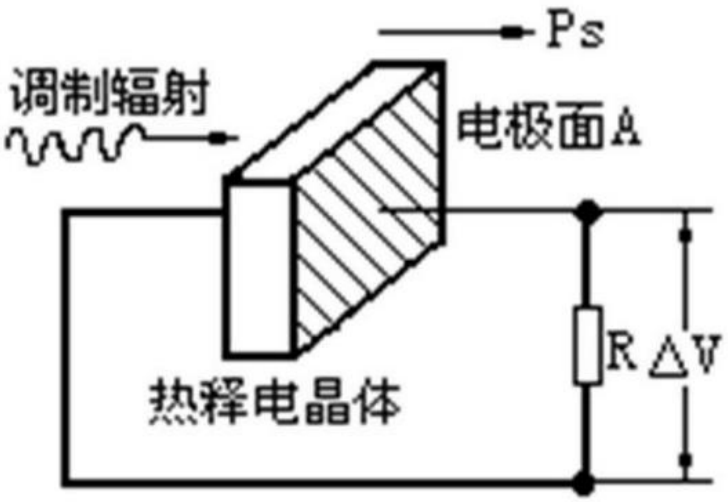
- **热释电红外探头**

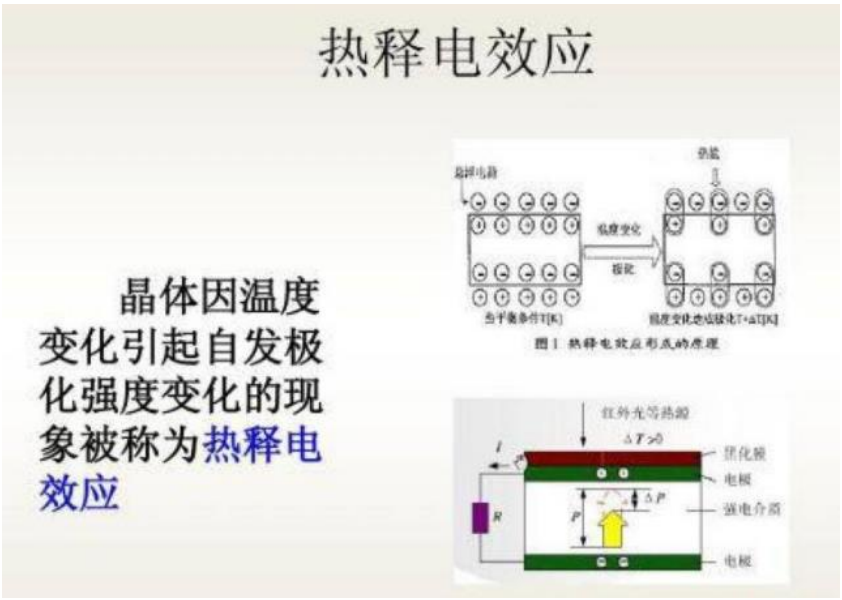
在结构上引入场效应管，其目的在于完成阻抗变换。由于热电元输出的是电荷信号，并不能直接使用，因而需要用电阻将其转换为电压形式。故引入的 N 沟道结型场效应管应接成共漏形式来完成阻抗变换。热释电红外传感器由传感探测元、干涉滤光片和场效应管匹配器三部分组成。设计时应将高热电材料制成一定厚度的薄片，并在它的两面镀上金属电极，然后加电对其进行极化，这样便制成了热释电探测元。

- **热释电效应**

是指极化强度随温度改变而表现出的电荷释放现象，宏观上是温度的改变使在材料的两端出现电压或产生电流。热释电效应与压电效应类似，热释电效应也是晶体的一种自然物理效应。对于具有自发式极化的晶体，当晶体受热或冷却后，由于温度的变化（△T）而导致自发式极化强度变化（△Ps），从而在晶体某一定方向产生表面极化电荷的现象称为热释

电效应。具有热释电性质的材料称为热释电体。压电陶瓷属于热释电体。若不考虑温度的不均匀性，热释电体一般具有一级和二级热释电效应。其中二级热释电效应是由于温度变化引起材料形变，再由压电效应产生电荷的二级效应。一般情况下，若温度变化率相同，升降温过程中产生的热释电电荷大小相等，但符号相反 。热释电效应在近 10 年被用于热释电红外探测器中，广泛地用于辐射和非接触式温度测量、红外光谱测量、激光参数测量、工业自动控制、空间技术、红外摄像中。我国利用 ATGSAS 晶体制成的红外摄像管已开始出口国外。其温度响应率达到 4～5μA/℃，温度分辨率小于 0.2℃，信号灵敏度高，图像清晰度和抗强光干扰能力也明显地提高，且滞后较小。此外，由于生物体中也存在热释电现象，故可预期热释电效应将在生物，乃至生命过程中有重要的应用。





➢ 热释电探头结构

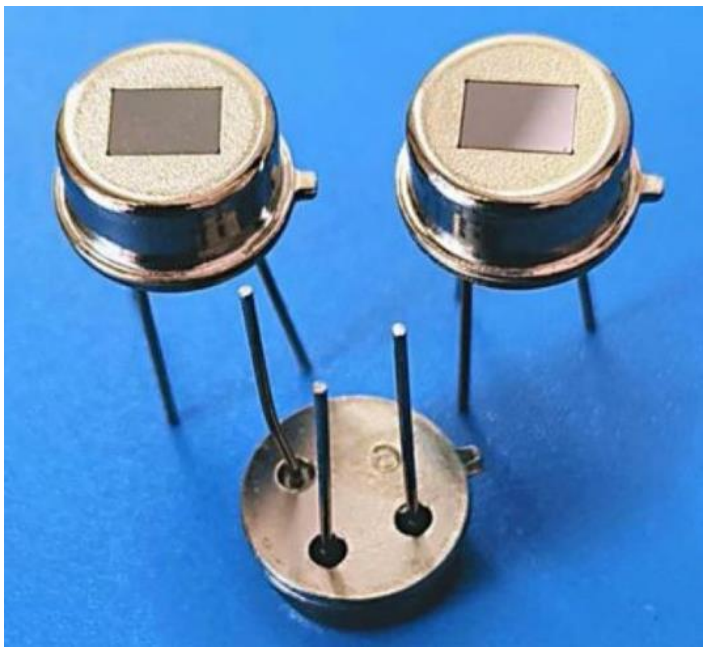
由滤光片、热释电探测元和前置放大器组成，补偿型热释电传感器还带有温度补偿元件。为防止外部环境对传感器输出信号的干扰，上述元件被真空封装在一个金属管内 。

1、热释电传感器的滤光片为带通滤光片，它封装在传感器壳体的顶端，使特定波长的红外辐射选择性地通过，到达热释电探测元+在其截止范围外的红外辐射则不能通过。

2、热释电探测元是热释电传感器的核心元件，它是在热释电晶体的两面镀上金属电极后，加电极化制成，相当于一个以热释电晶体为电介质的平板电容器。当它受到非恒定强度的红外光照射时，产生的温度变化导致其表面电极的电荷密度发生改变，从而产生热释电电流。

3、前置放大器由一个高内阻的场效应管源极跟随器构成，通过阻抗变换，将热释电探测元微弱的电流信号转换为有用的电压信号输出。

4、前置放大器将微弱的热释电电流转换为有效电压输出。前置放大器必须具备高增益、低噪声、抗干扰能力强的特点，以便从众多的噪声干扰中提取微弱的有用信号。热释电探测元和前置放大器通常集成封装在晶体管内，以避免空气湿度使泄露电流增大。这种结构的前置放大器信噪比高，受温度影响小。



➢ 热释电红外探头的工作原理

人体都有恒定的体温，一般在 37 度，所以会发出特定波长 10UM 左右的红外线，被动式红外探头就是靠探测人体发射的 10UM 左右的红外线而进行工作的。人体发射的 10UM 左右的红外线通过菲泥 尔滤光片增强后聚集到红外感应源上。红外感应源通常采用热释电元件，这种元件在接收到人体红外辐射温度发生变化时就会失去电荷平衡，向外释放电荷，后续电路经检测处理后就能产生报警信号。

1)这种探头是以探测人体辐射为目标的。所以热释电元件对波长为 10UM 左右的红外辐射必须非常敏感。

2)为了仅仅对人体的红外辐射敏感，在它的辐射照面通常覆盖有特殊的菲泥尔滤光片，使环境的干扰受到明显的控制作用。

3)被动红外探头，其传感器包含两个互相串联或并联的热释电元。而且制成的两个电极化方向正好相反，环境背景辐射对两个热释元件几乎具有相同的作用，使其产生释电效应相互抵消，于是探测器无信号输出。

4)一旦人侵入探测区域内，人体红外辐射通过部分镜面聚焦，并被热释电元接收，但是两片

热释电元接收到的热量不同，热释电也不同，不能抵消，经信号处理而报警。
5)菲泥尔滤光片根据性能要求不同，具有不同的焦距(感应距离)，从而产生不同的监控视场，视场越多，控制越严密。

3.2 人体红外传感器的电路原理图

人体红外传感器的电路原理如图 3.1 所示。

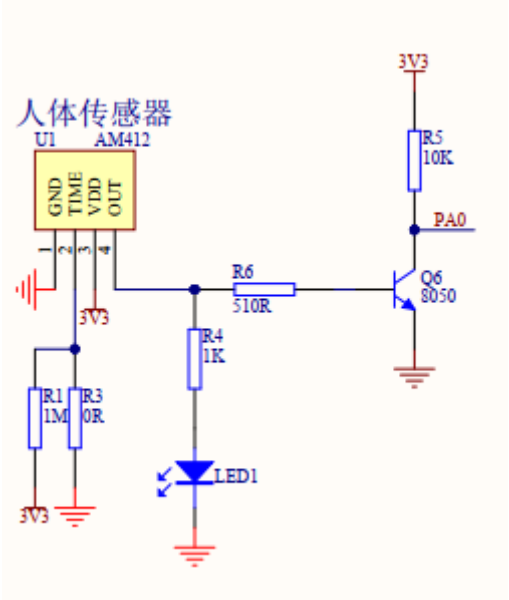


图 3.1　人体传感器电路

其数据口连接到 STM32 的 PA0 进行电平检测。当检测到人体时，传感器输出高电平，三极管导通，数据口 PA0 引脚输入低电平；反之输入高电平。

3.3 实验源码

```
#include "stm32f10x.h"
#include "stm32f10x.h"

#include "Delay.h"
#include "LED.h"
#include "Time.h"
#include "USART.h"

void RCC_Configuration(void);


#define GET_SDA() GPIO_ReadInputDataBit(GPIOA,GPIO_Pin_0)
```

```
void Sensor_Init(void)
{
    GPIO_InitTypeDef     GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_0;
    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_IPU;
    GPIO_Init(GPIOA,&GPIO_InitStructure);
}

int main()
{
    RCC_Configuration();
    LED_Init();
    Sensor_Init();
    USART1_Init();
    Delay_Init(72);
    while(1)
    {
        if(GET_SDA()==1)
        {

            printf("没人\r\n");
        }
        else
        {
            printf("有人\r\n");
        }
        delay_ms(1000);
   }

}
```

/**
函数名称: RCC_Configuration(void)
函数参数：无
功能说明：STM32 端口时钟初始化

```
函数返回：无
编写作者：FRO-PJF01
修改时间：2017.02.22
*************************************/
void RCC_Configuration(void)
{
  SystemInit();
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA                    |
RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC
                        | RCC_APB2Periph_GPIOD| RCC_APB2Periph_GPIOE ,
ENABLE);
}
```

4.实验步骤

4.1 编译并将程序下载到节点

在 KEIL 开发环境下打开**人体感应传感器实验\RVMDK** 下的 Test.uvprojx 工程，编译并下载到的节点上。

4.2 实验运行效果

将程序下载到节点板，使用配套的串口线将计算机串口与节点的 DB9 串口接头相连，如图 4.2.0 所示，**注意，将节点右下角上的三位拨打开关拨至左边**，给节点重新上电，打开**串口调试工具.exe** 程序，打开正确的端口，进行 115200-8-N-1 设置，打开串口,放置人体传感器，串口助手显示是否检测到人体。

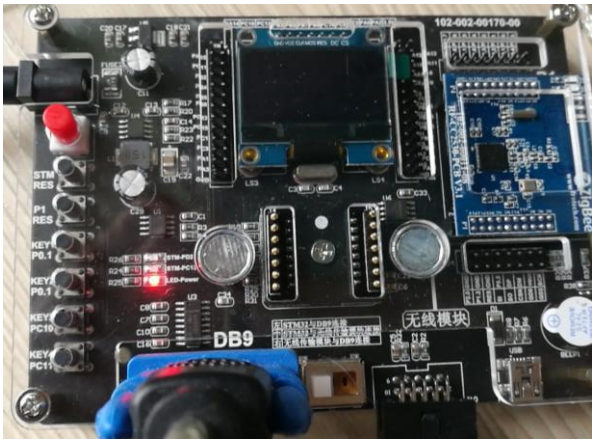


图 4.2.0 正确连接串口及白色拨打开关

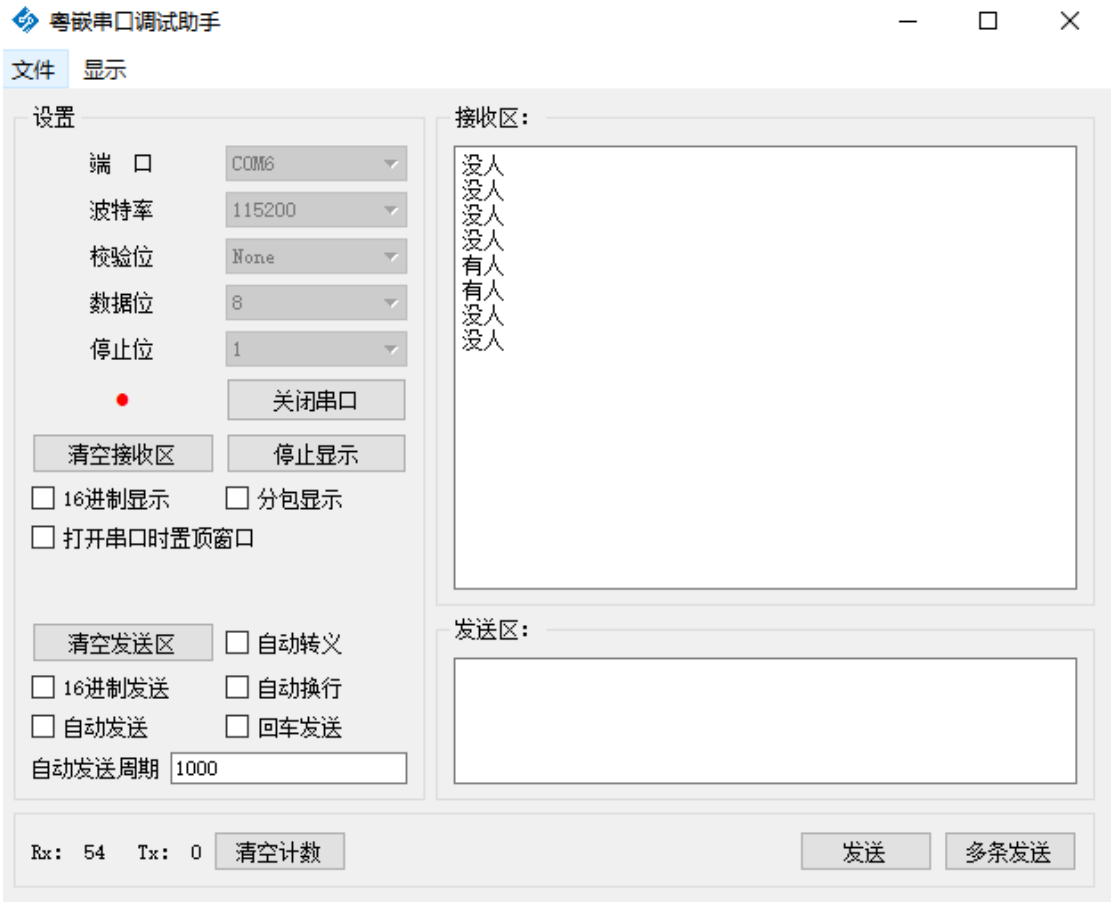


图 4.2.1 串口收发数据

注意：串口参数为 115200-8-N-1。

1.7 stm32 传感器数据采集实验(烟雾)

1.实验目的

学习烟雾传感器的使用方法；

2.实验设备

硬件：带有 STM32 芯片节点、电脑、J-LINK、串口线等；
软件：Keil 开发环境；

3.实验原理

3.1 烟雾检测传感器的介绍

烟雾检测传感器采用可燃气体传感器MQ-2，MQ-2气体传感器所使用的气敏材料是在清洁空气中电导率较低的二氧化锡(SnO_2)。当传感器所处环境中存在可燃气体时，传感器的电导率随空气中可燃气体浓度的增加而增大。使用简单的电路即可将电导率的变化转换为与该气体浓度相对应的输出信号。

MQ-2气体传感器对液化气、丙烷、氢气的灵敏度高，对天然气和其它可燃蒸汽的检测也很理想。这种传感器可检测多种可燃性气体，另外烟雾中含有多种MQ-2可检测的其它，则其可作为烟雾传感器使用，是一款适合多种应用的低成本传感器。

3.1.1 元器件结构图

MQ-2型气体传感器结构如图3.1所示。

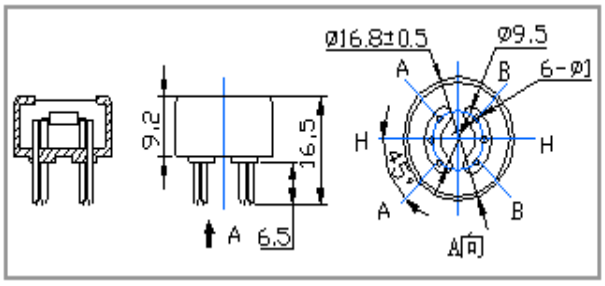


图 3.1 MQ-2 型气体传感器结构

3.1.2 基本测试回路

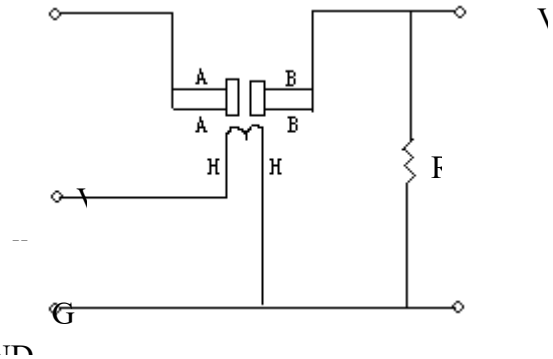


图 3.2 MQ-2 型气体传感器测试电路

图3.2是传感器的基本测试电路。该传感器需要施加2个电压：加热器电压（V_H）和测试电压（V_C）。其中 V_H用于为传感器提供特定的工作温度。V_C 则是用于测定与传感器串联的负载电阻（R_L）上的电压（V_{RL}）。这种传感器具有轻微的极性，V_C需用直流电源。在满足传感器电性能要求的前提下，V_C和V_H可以共用同一个电源电路。为更好利用传感器的 性能，需要

选择恰当的 RL 值。

3.1.3 技术指标

MQ-2 的基本参数如图表 1。

表 1 MQ-2 技术指标

产品型号			MQ-2
产品类型			半导体气敏元件
标准封装			胶木（黑胶木）
检测气体			可燃气体、烟雾
检测浓度			300-10000ppm(可燃气体)
标准电路条件	回路电压	Vc	≤24V DC
	加热电压	VH	5.0V±0.2V ACorDC
	负载电阻	RL	可调
标	加热	R	31Ω±3Ω（室温）

准测试条件下气敏元件特性	电阻	H	
	加热功耗	H	≤900mW
	敏感体表面电阻	s	2KΩ-20KΩ(in 2000ppm C3H8)
	灵敏度	S	Rs(in air)/Rs(1000ppm 异丁烷) ≥5
	浓度斜率		≤ 0.6(R3000ppm/R1000ppm C3H8)
标准测试条件	温度、湿度		20℃±2℃；65%±5%RH
	标准测试电路		Vc:5.0V±0.1V；VH: 5.0V±0.1V
	预热时间		不少于 48 小时

3.1.4 灵敏度特性

MQ-2 烟雾传感器灵敏度曲线如图 3.3 所示。

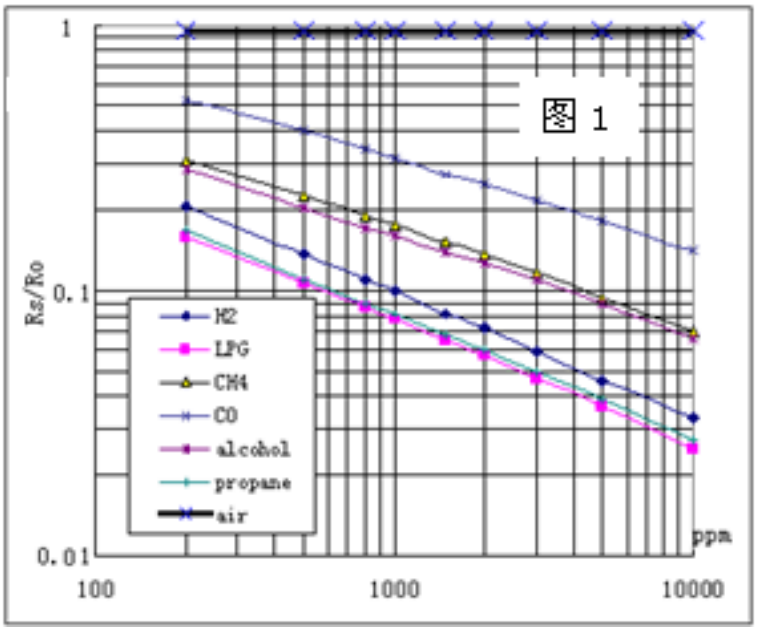


图 3.3　MQ-2 烟雾传感器灵敏度曲线

3.2 烟雾检测传感器电路原理图

MQ-2 烟雾传感器电路原理如图 3.4 所示，烟雾传感器的检测点 D3 与 PA4 连接

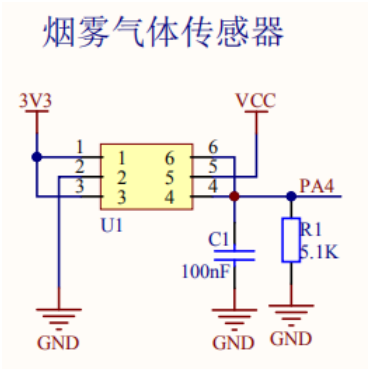


图 3.4　MQ-2 烟雾传感器电路

其中 U1(MQ-2)的 PIN5 与 PIN2 为加热电路，对应结构图中的两个 H 端；PIN1、PIN3、PIN4、PIN6 构成检测电路。

MQ-2 传感器的供电电压 Vc 和加热电压 Vh 都为 5V，负载电阻 R1 为 5.1K 欧姆。ADC(PA4)在清洁空气中的值以及检测到烟雾时的值需要根据实际应用情况进行调整，以下仅为在实验条件下做的不完全的实验结果，仅供参考。在清洁空气中，ADC 的 AD 采样值为 50 左右；在烟雾中(燃烧纸产生的烟雾或者液化气)，ADC 的 AD 采样值为大于 85。当 AD 采集的数值大于 85 时表明检测到烟雾。

3.3 实验源码

```
void Sensor_Init(void)
{
    GPIO_InitTypeDef     GPIO_InitStructure;

    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_4;
    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_AIN;
    GPIO_Init(GPIOA,&GPIO_InitStructure);
}

//初始化 ADC1
//这里我们仅以规则通道为例
//我们默认仅开启通道 4
void    Adc1_Init(void)
{
    ADC_InitTypeDef ADC_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_ADC1,ENABLE );  // 使 能 ADC1 通道时钟

  RCC_APB2PeriphResetCmd(RCC_APB2Periph_ADC1,ENABLE);//ADC 复位

  RCC_APB2PeriphResetCmd(RCC_APB2Periph_ADC1,DISABLE);// 复 位 结 束


    ADC_DeInit(ADC1);  //复位 ADC1,将外设 ADC1 的全部寄存器重设为缺省值


    ADC_InitStructure.ADC_Mode = ADC_Mode_Independent;     //ADC 工 作 模 式: 独立模式
    ADC_InitStructure.ADC_ScanConvMode = DISABLE; //模数转换工作在单通道模式
    ADC_InitStructure.ADC_ContinuousConvMode = DISABLE;  //模数转换工作在单次转换模式
    ADC_InitStructure.ADC_ExternalTrigConv  =  ADC_ExternalTrigConv_None;
    //转换由软件而不是外部触发启动
    ADC_InitStructure.ADC_DataAlign = ADC_DataAlign_Right;  //ADC 数 据 右
```

```
对齐
    ADC_InitStructure.ADC_NbrOfChannel = 1;  //顺序进行规则转换的 ADC 通道的数目
    ADC_Init(ADC1, &ADC_InitStructure);  //根据 ADC_InitStruct 中指定的参数初始化外设 ADCx 的寄存器


    ADC_Cmd(ADC1, ENABLE);    //使能指定的 ADC1

    ADC_ResetCalibration(ADC1);   //使能复位校准

    while(ADC_GetResetCalibrationStatus(ADC1));    //等待复位校准结束

    ADC_StartCalibration(ADC1);     //开启 AD 校准

    while(ADC_GetCalibrationStatus(ADC1)); //等待校准结束
}

//获得 ADC1 通道 4 的值
//返回值:转换结果
u16 Get_Adc1(void)
{
  //设置指定 ADC 的规则组通道，一个序列，采样时间
    ADC_RegularChannelConfig(ADC1, ADC_Channel_4, 1, ADC_SampleTime_239Cycles5 ); //ADC1,ADC 通道,采样时间为 239.5 周期


    ADC_SoftwareStartConvCmd(ADC1, ENABLE);     //使能指定的 ADC1 的软件转换启动功能

    while(!ADC_GetFlagStatus(ADC1, ADC_FLAG_EOC ));//等待转换结束

    return ADC_GetConversionValue(ADC1); //返回最近一次 ADC1 规则组的转换结果
}

int main()
{
    u16 adc;
    RCC_Configuration();
    LED_Init();
```

```
    Sensor_Init();
    USART1_Init();
    Delay_Init(72);
    Adc1_Init();
    while(1)
    {
        adc=Get_Adc1();
        printf("烟雾传感器模拟数值：%d\r\n",adc);
        delay_ms(1000);
  }

}
```

```
/****************************************
函数名称: RCC_Configuration(void)
函数参数：无
功能说明：STM32 端口时钟初始化
函数返回：无
编写作者：FRO-PJF01
修改时间：2017.02.22
****************************************/
void RCC_Configuration(void)
{
  SystemInit();
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE);
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA | RCC_APB2Periph_GPIOB | RCC_APB2Periph_GPIOC
                         | RCC_APB2Periph_GPIOD| RCC_APB2Periph_GPIOE , ENABLE);
}
```

4.实验步骤

4.1 编译并将程序下载到节点

在 KEIL 开发环境下打开**烟雾传感器实验\RVMDK** 下的 Test.uvprojx 工程，编译并下载到的节点上。

4.2 实验运行效果

将程序下载到节点板，使用配套的串口线将计算机串口与节点的 DB9 串口接头相连，如图 4.2.0 所示，**注意，将节点右下角上的三位拨打开关拨至左边**，给节点重新上电，打开**串口调试工具.exe** 程序，打开正确的端口，进行 115200-8-N-1 设置，打开串口,放置烟雾传感器，串口助手显示烟雾传感器数值。

（烟雾传感器上电后需预热两分钟）

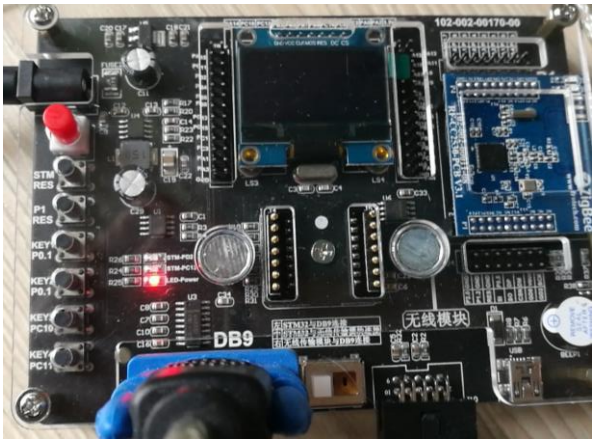


图 4.2.0 正确连接串口及白色拨打开关

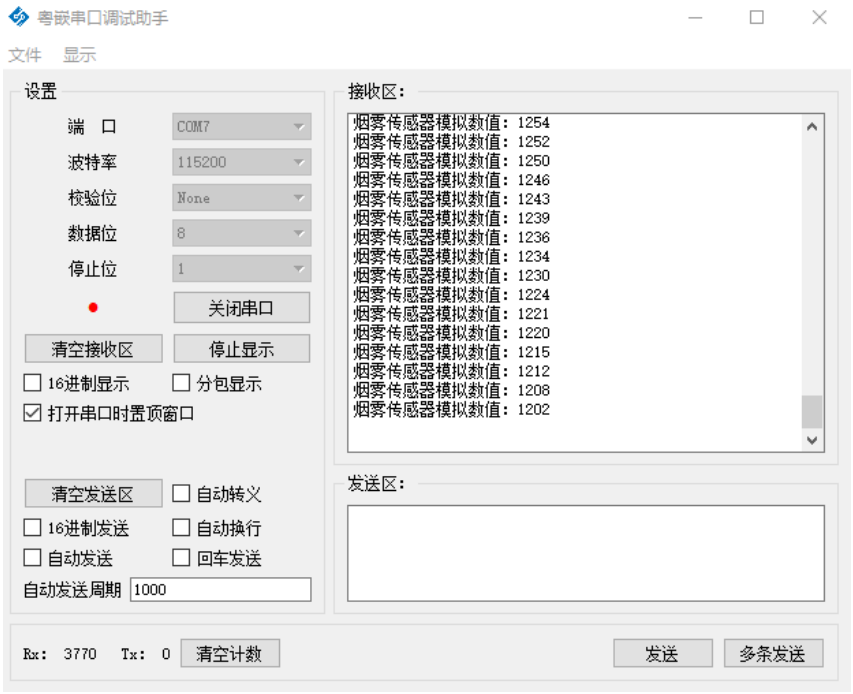


图 4.2.1 串口收发数据

注意：串口参数为 115200-8-N-1。

1.8 风扇控制实验

1.实验目的

学习使用串口控制风扇的使用方法；

2.实验设备

硬件：带有 STM32 芯片节点、电脑、J-LINK、串口线等；
软件：Keil 开发环境；

3.实验原理

3.1 直流电机控制模块的电路图

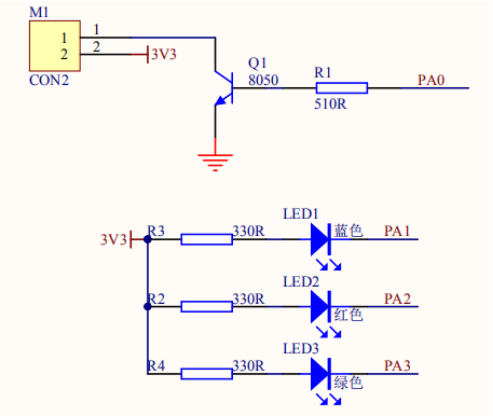


图 3.2 直流电机模块

我们可以看到直流电机控制端 Q1 与 STM32 的 PA0 口连接，当 PA0 为高电平时候 Q1 导通，那么直流电机风扇有电。

3.2 部分代码说明

```
14
15  void Fan_Init(void)
16  {
17    GPIO_InitTypeDef    GPIO_InitStructure;
18
19    GPIO_InitStructure.GPIO_Pin=GPIO_Pin_0;
20    GPIO_InitStructure.GPIO_Speed=GPIO_Speed_50MHz;
21    GPIO_InitStructure.GPIO_Mode=GPIO_Mode_Out_PP;
22    GPIO_Init(GPIOA,&GPIO_InitStructure);
23  }
24
25
26 /*****************************************
27  函数名称：main()
28  函数参数：无
29  功能说明：初始化串口1，实现把从上位机接收的数据原样输出给上位机
30  函数返回：无
31  编写作者：FRO-PJF01
32  修改时间：2017.02.22
33 *****************************************/
34  int main()
35  {
36
37    RCC_Configuration();
38    Fan_Init();
39    USART1_Init();//串口1初始化
40
41    while(1)
42    {
43
44    }
45
46  }
47
72 *****************************************/
73  void USART1_IRQHandler(void)
74  {
75    unsigned char res;
76    if(USART_GetITStatus(USART1, USART_IT_RXNE) != RESET)        //判定是否有接收到数据
77    {
78      //SART1_RX_BUF[USART1_WR ++] =USART_ReceiveData(USART1);//有数据，那么放在缓存里
79      //USART1_RX_flag=1;                                      //标志为1，表示有接收到数据
80      res = USART_ReceiveData(USART1);
81      if(res == 'O')
82      {
83        GPIO_SetBits(GPIOA,GPIO_Pin_0);
84      }
85      else if(res == 'C')
86      {
87        GPIO_ResetBits(GPIOA,GPIO_Pin_0);
88      }
89    }
90  }
91
```

在串口中断中判断接收的串口数据，控制风扇的启停。

4.实验步骤

4.1 编译并将程序下载到节点

在 KEIL 开发环境下打开**风扇控制实验\RVMDK** 下的 Test.uvprojx 工程，编

译并下载到的节点上。

4.2 实验运行效果

将程序下载到节点板，使用配套的串口线将计算机串口与节点的 DB9 串口接头相连，如图 4.2.0 所示，**注意，将节点右下角上的三位拨打开关拨至左边**，给节点重新上电，打开**串口调试工具.exe** 程序，打开正确的端口，进行 115200-8-N-1 设置，打开串口,放置风扇传感器。

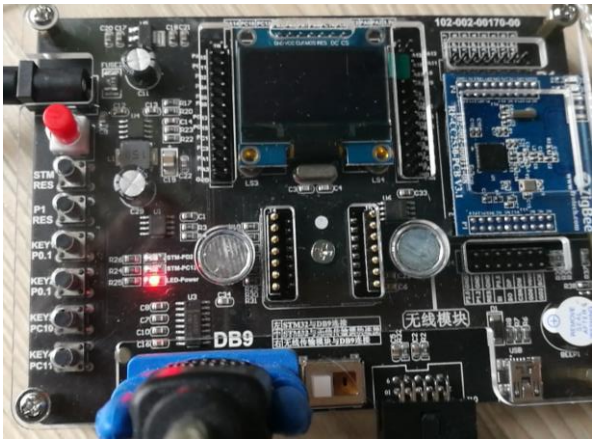


图 4.2.0 正确连接串口及白色拨打开关

往串口助手中发送‘0’，风扇启动；发送‘C’，风扇停止。

1.9 stm32 OLED 显示实验

1.实验目的

学习 OLED 显示屏的使用方法；

2.实验设备

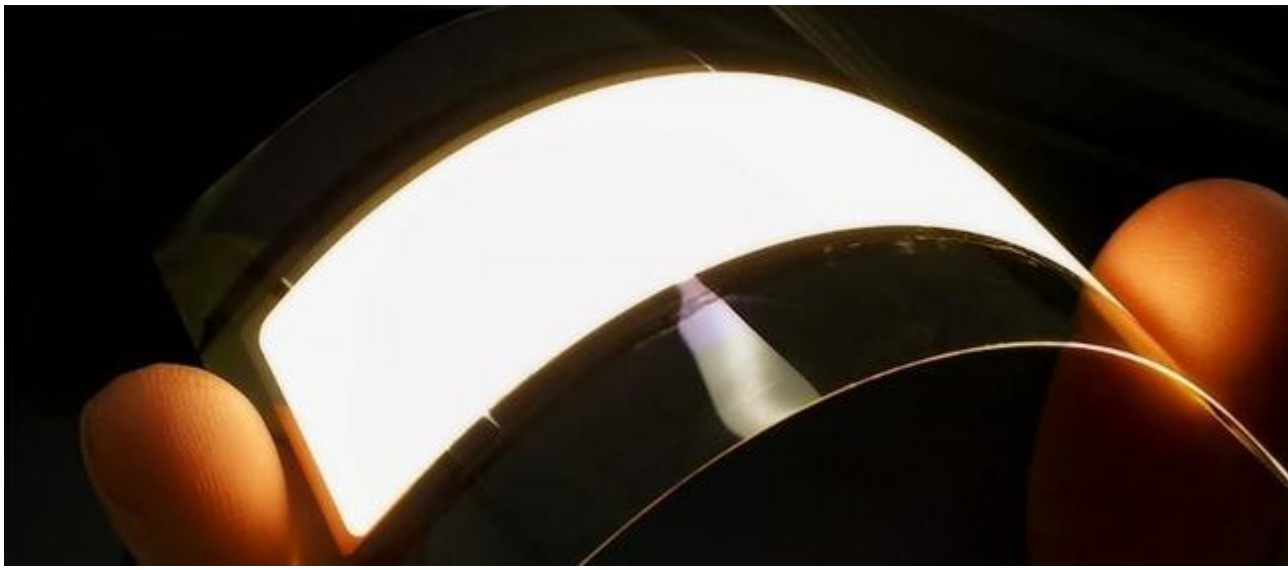
硬件：带有 STM32 芯片节点、电脑、J-LINK、串口线等；
软件：Keil 开发环境；

3.实验原理

3.1 OLED 介绍

- OLED 初步认识

OLED(Organic Light-Emitting Diode)，即有机发光二极管. OLED 由于同时具备自发光，不需背光源，对比度高，厚度薄，视角广，反应速度快，可用于挠曲性面板，使用温度范围广，构造及制程简单等优异特性，被认为是下一代的平面显示器新兴应用技术。具有轻，薄，省电等特性，通信接口简单.



这里我们描述的是 IIC 接口的 0.96 寸 OLED，其实对于其他接口的 OLED，分析思路是一样的。显示模块我们就关心三个问题，显示模块初始化；在哪里显示；显示什么。对于 OLED 同样适用，它只不过显示的地方比较多，显示内容更加丰富一点，但是思路基本相同。下面我们就从这个三个问题入手，简单讲述如

何使用这款显示模块。

- OLED 初始化

关于 OLED 初始化，芯片手册上描述的非常清楚了，这没什么好讨论的，照做即可。

（1）Set MUX Ratio A8h, 3Fh
（2）Set Display OffsetD3h, 00h
（3）Set Display StartLine 40h
（4）Set Segment re-mapA0h/A1h
（5）Set COM Output ScanDirection C0h/C8h
（6）Set COM Pinshardware configuration DAh,02h
（7）Set Contrast Control81h,7Fh
（8）Disable EntireDisplay On A4h
（9）Set Normal DisplayA6h
（10）Set Osc FrequencyD5h, 80h
（11）Enable charge pumpregulator 8Dh, 14h
（12）Display On AFh

- OLED 显示

要知道知道在哪里显示，就需要先知道哪些地方可能显示。oled 模块的分辨率是 128×64，也就是说一共能显示 128×64 这么多个“点”，下面我们用一个 128 列，64 行的表格来描述更清晰一些，如下所示：

			Col 0	Col 1	Col 2	Col 3	Col 4	....	Col 125	Col 126	Col 127
PAGE 0	bit 0	Row 0	0								
	bit 1	Row 1	0								
	bit 2	Row 2	0								
	bit 3	Row 3	1								
	bit 4	Row 4	0								
	bit 5	Row 5	0								
	bit 6	Row 6	0								
	bit 7	Row 7	0								
PAGE 1	bit 0	Row 8									
	bit 1	Row 9									
	bit 2	Row 10									
	bit 3	Row 11									
	bit 4	Row 12									
	bit 5	Row 13									
	bit 6	Row 14									
	bit 7	Row 15									
PAGE 2											
PAGE 3											
PAGE 4											
PAGE 5											
PAGE 6											
PAGE 7											

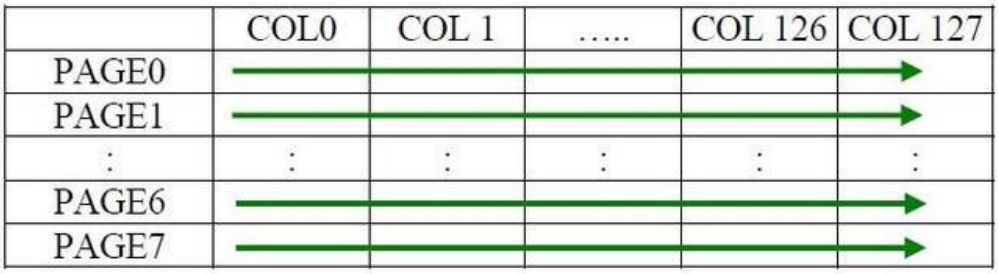
显示模块上的每一个点对应着这个表格的一个空格，假设你在某一个空格中放 1 表示这个‘点’亮，那么放 0 就表示这个‘点’暗。由于我们在写入数据时通常以字节为单位，那么现在把表格中的 Col0 这一列对应的 Row0-Row7 作为一个单位，一共 8 个空格，刚好对应一个字节。那么这个字节的高低位如何分配呢？最低位放到 Row0-Col0 对应的空格，最高位放到 Row7-Col0 对应的空格。这样

当你写入一个字节的数据 0x08 时，对应的 Col0-Row3 这个‘点’就亮了，其他 7 个点为暗。那么这个数据 0x08 写到哪里去了？这个显示模块一定有个存储空间来存放这些写入的数据，暂且把它叫做 PAGE0,PAGE1,...PAGE7. 这样每一个 PAGE 就对应着 8 行，比如 PAGE0 就对应着 Row0-Row7.

经过以上分析，问题就变得简单了，就是如何访问 PAGE0-PAGE7，然后往里面填数据就行了。这个就涉及到寻址模式了，一共三种，分别 是页寻址，水平寻址和垂直寻址模式。以下三幅图描述了这三种寻址模式

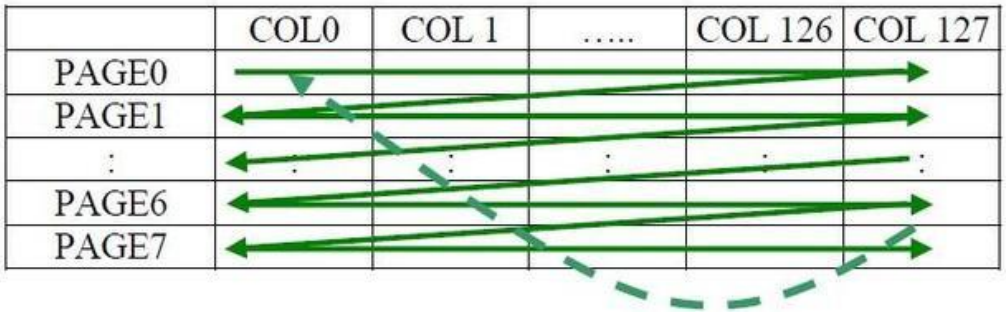
（1）页寻址模式

Figure 10-1 : Address Pointer Movement of Page addressing mode



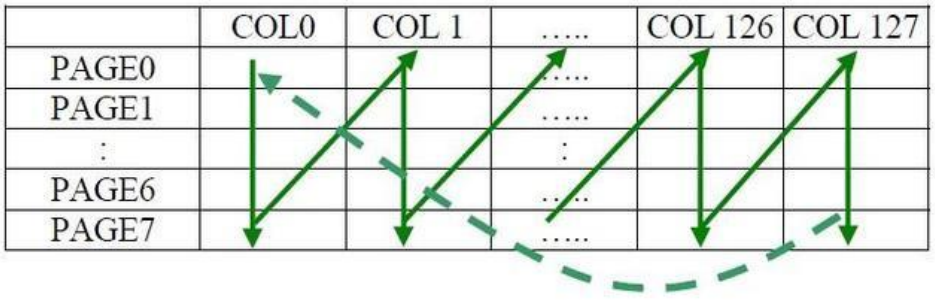
（2）水平寻址模式

Figure 10-3 : Address Pointer Movement of Horizontal addressing mode



（3）垂直寻址模式

Figure 10-4 : Address Pointer Movement of Vertical addressing mode



下面就去查看 OLED 模块说明书的指令表确定其中任意一种寻址模式，然后根据需要显示的内容填数据就可以了。

3.2 OLED 的电路原理图

OLED 的电路原理如图 3.1 所示。

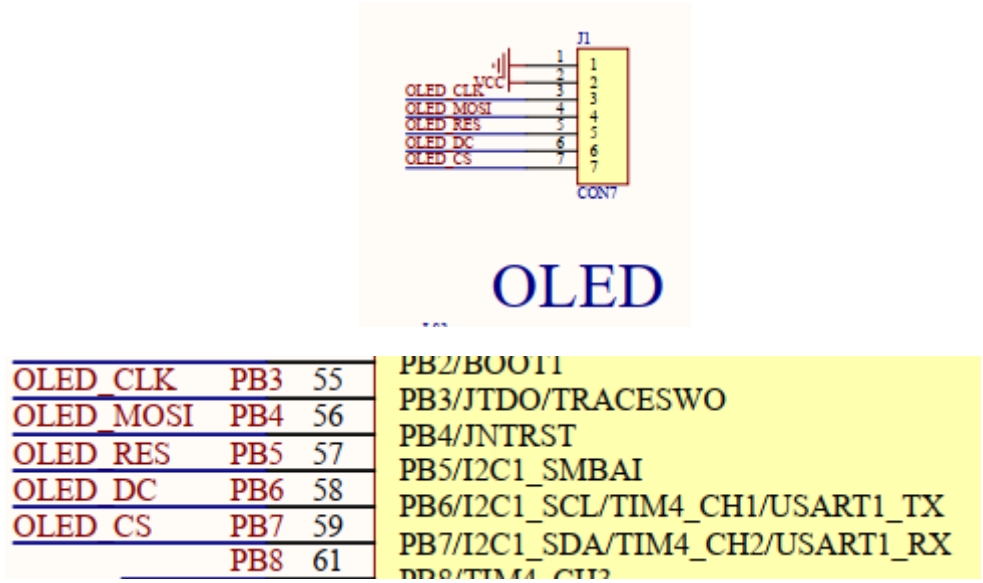


图 3.1 OLED 电路

其数据口连接到 STM32 的 PB3-PB7 引脚。其中 PB3 为 JTAG 的复用引脚，在使用时需要将 JTAG 失能。

3.3 实验源码

```
//反显函数
void OLED_ColorTurn(u8 i)
{
    if(i==0)
        {
            OLED_WR_Byte(0xA6,OLED_CMD);//正常显示
        }
    if(i==1)
        {
            OLED_WR_Byte(0xA7,OLED_CMD);//反色显示
        }
}

//屏幕旋转 180 度
void OLED_DisplayTurn(u8 i)
{
    if(i==0)
```

```
        {
            OLED_WR_Byte(0xC8,OLED_CMD);//正常显示
            OLED_WR_Byte(0xA1,OLED_CMD);
        }
    if(i==1)
        {
            OLED_WR_Byte(0xC0,OLED_CMD);//反转显示
            OLED_WR_Byte(0xA0,OLED_CMD);
        }
}

void OLED_WR_Byte(u8 dat,u8 cmd)
{
    u8 i;
    if(cmd)
    {
      OLED_DC_Set();
    }
    else
    {
      OLED_DC_Clr();
    }
    OLED_CS_Clr();
    for(i=0;i<8;i++)
    {
        OLED_SCL_Clr();
        if(dat&0x80)
            OLED_SDA_Set();
        else
            OLED_SDA_Clr();
        delay_us(5);
        OLED_SCL_Set();
        dat<<=1;
        delay_us(5);
    }
    OLED_CS_Set();
    OLED_DC_Set();
}

//开启 OLED 显示
void OLED_DisPlay_On(void)
```

```
{
    OLED_WR_Byte(0x8D,OLED_CMD);//电荷泵使能
    OLED_WR_Byte(0x14,OLED_CMD);//开启电荷泵
    OLED_WR_Byte(0xAF,OLED_CMD);//点亮屏幕
}

//关闭 OLED 显示
void OLED_DisPlay_Off(void)
{
    OLED_WR_Byte(0x8D,OLED_CMD);//电荷泵使能
    OLED_WR_Byte(0x10,OLED_CMD);//关闭电荷泵
    OLED_WR_Byte(0xAE,OLED_CMD);//关闭屏幕
}

//更新显存到 OLED
void OLED_Refresh(void)
{
    u8 i,n;
    for(i=0;i<8;i++)
    {
        OLED_WR_Byte(0xb0+i,OLED_CMD); //设置行起始地址
        OLED_WR_Byte(0x02,OLED_CMD);     //设置低列起始地址
        OLED_WR_Byte(0x10,OLED_CMD);     //设置高列起始地址
        for(n=0;n<128;n++)
          OLED_WR_Byte(OLED_GRAM[n][i],OLED_DATA);
  }
}
//清屏函数
void OLED_Clear(void)
{
    u8 i,n;
    for(i=0;i<8;i++)
    {
        for(n=0;n<128;n++)
            {
             OLED_GRAM[n][i]=0;//清除所有数据
            }
  }
    OLED_Refresh();//更新显示
}
```

```
//画点
//x:0~127
//y:0~63
//t:1  填充  0,清空
void OLED_DrawPoint(u8 x,u8 y,u8 t)
{
    u8 i,m,n;
    i=y/8;
    m=y%8;
    n=1<<m;
    if(t){OLED_GRAM[x][i]|=n;}
    else
    {
        OLED_GRAM[x][i]=~OLED_GRAM[x][i];
        OLED_GRAM[x][i]|=n;
        OLED_GRAM[x][i]=~OLED_GRAM[x][i];
    }
}

//画线
//x1,y1:起点坐标
//x2,y2:结束坐标
void OLED_DrawLine(u8 x1,u8 y1,u8 x2,u8 y2,u8 mode)
{
    u16 t;
    int xerr=0,yerr=0,delta_x,delta_y,distance;
    int incx,incy,uRow,uCol;
    delta_x=x2-x1; //计算坐标增量
    delta_y=y2-y1;
    uRow=x1;//画线起点坐标
    uCol=y1;
    if(delta_x>0)incx=1; //设置单步方向
    else if (delta_x==0)incx=0;//垂直线
    else {incx=-1;delta_x=-delta_x;}
    if(delta_y>0)incy=1;
    else if (delta_y==0)incy=0;//水平线
    else {incy=-1;delta_y=-delta_x;}
    if(delta_x>delta_y)distance=delta_x; //选取基本增量坐标轴
    else distance=delta_y;
    for(t=0;t<distance+1;t++)
    {
```

```
            OLED_DrawPoint(uRow,uCol,mode);//画点
            xerr+=delta_x;
            yerr+=delta_y;
            if(xerr>distance)
            {
                xerr-=distance;
                uRow+=incx;
            }
            if(yerr>distance)
            {
                yerr-=distance;
                uCol+=incy;
            }
        }
}
//x,y:圆心坐标
//r:圆的半径
void OLED_DrawCircle(u8 x,u8 y,u8 r)
{
    int a, b,num;
     a = 0;
     b = r;
     while(2 * b * b >= r * r)
     {
          OLED_DrawPoint(x + a, y - b,1);
          OLED_DrawPoint(x - a, y - b,1);
          OLED_DrawPoint(x - a, y + b,1);
          OLED_DrawPoint(x + a, y + b,1);

          OLED_DrawPoint(x + b, y + a,1);
          OLED_DrawPoint(x + b, y - a,1);
          OLED_DrawPoint(x - b, y - a,1);
          OLED_DrawPoint(x - b, y + a,1);

          a++;
          num = (a * a + b * b) - r*r;//计算画的点离圆心的距离
          if(num > 0)
          {
              b--;
              a--;
          }
```

```
    }
}



//在指定位置显示一个字符,包括部分字符
//x:0~127
//y:0~63
//size1:选择字体  6x8/6x12/8x16/12x24
//mode:0,反色显示;1,正常显示
void OLED_ShowChar(u8 x,u8 y,u8 chr,u8 size1,u8 mode)
{
    u8 i,m,temp,size2,chr1;
    u8 x0=x,y0=y;
    if(size1==8)size2=6;
    else size2=(size1/8+((size1%8)?1:0))*(size1/2);  //得到字体一个字符对应点阵集所占的字节数
    chr1=chr-' ';  //计算偏移后的值
    for(i=0;i<size2;i++)
    {
        if(size1==8)
                {temp=asc2_0806[chr1][i];} //调用 0806 字体
        else if(size1==12)
          {temp=asc2_1206[chr1][i];} //调用 1206 字体
        else if(size1==16)
          {temp=asc2_1608[chr1][i];} //调用 1608 字体
        else if(size1==24)
          {temp=asc2_2412[chr1][i];} //调用 2412 字体
        else return;
        for(m=0;m<8;m++)
        {
            if(temp&0x01)OLED_DrawPoint(x,y,mode);
            else OLED_DrawPoint(x,y,!mode);
            temp>>=1;
            y++;
        }
        x++;
        if((size1!=8)&&((x-x0)==size1/2))
        {x=x0;y0=y0+8;}
        y=y0;
   }
```

```
}


//显示字符串
//x,y:起点坐标
//size1:字体大小
//*chr:字符串起始地址
//mode:0,反色显示;1,正常显示
void OLED_ShowString(u8 x,u8 y,u8 *chr,u8 size1,u8 mode)
{
    while((*chr>=' ')&&(*chr<='~'))//判断是不是非法字符!
    {
        OLED_ShowChar(x,y,*chr,size1,mode);
        if(size1==8)x+=6;
        else x+=size1/2;
        chr++;
  }
}

//m^n
u32 OLED_Pow(u8 m,u8 n)
{
    u32 result=1;
    while(n--)
    {
      result*=m;
    }
    return result;
}

//显示数字
//x,y :起点坐标
//num :要显示的数字
//len :数字的位数
//size:字体大小
//mode:0,反色显示;1,正常显示
void OLED_ShowNum(u8 x,u8 y,u32 num,u8 len,u8 size1,u8 mode)
{
    u8 t,temp,m=0;
    if(size1==8)m=2;
    for(t=0;t<len;t++)
```

```
    {
        temp=(num/OLED_Pow(10,len-t-1))%10;
            if(temp==0)
            {
                OLED_ShowChar(x+(size1/2+m)*t,y,'0',size1,mode);
        }
            else
            {
              OLED_ShowChar(x+(size1/2+m)*t,y,temp+'0',size1,mode);
            }
  }
}

//x,y：起点坐标
//sizex,sizey,图片长宽
//BMP[]：要写入的图片数组
//mode:0,反色显示;1,正常显示
void OLED_ShowPicture(u8 x,u8 y,u8 sizex,u8 sizey,u8 BMP[],u8 mode)
{
    u16 j=0;
    u8 i,n,temp,m;
    u8 x0=x,y0=y;
    sizey=sizey/8+((sizey%8)?1:0);
    for(n=0;n<sizey;n++)
    {
        for(i=0;i<sizex;i++)
        {
            temp=BMP[j];
            j++;
            for(m=0;m<8;m++)
            {
                if(temp&0x01)OLED_DrawPoint(x,y,mode);
                else OLED_DrawPoint(x,y,!mode);
                temp>>=1;
                y++;
            }
            x++;
            if((x-x0)==sizex)
            {
                x=x0;
                y0=y0+8;
```

```
                }
                y=y0;
        }
        }
}

//OLED 的初始化
void OLED_Init(void)
{
    GPIO_InitTypeDef    GPIO_InitStructure;

    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOB, ENABLE);      //使能
A 端口时钟
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_AFIO, ENABLE); //
  GPIO_PinRemapConfig(GPIO_Remap_SWJ_JTAGDisable, ENABLE);//JTAG 失
能

    GPIO_InitStructure.GPIO_Pin                                    =
GPIO_Pin_2|GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPIO_Pin_7;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;//速度 50MHz
    GPIO_Init(GPIOB, &GPIO_InitStructure);

    GPIO_SetBits(GPIOB,GPIO_Pin_3|GPIO_Pin_4|GPIO_Pin_5|GPIO_Pin_6|GPI
O_Pin_7);
    OLED_RES_Clr();
    delay_ms(200);
    OLED_RES_Set();

    OLED_WR_Byte(0xAE,OLED_CMD);//--turn off oled panel
    OLED_WR_Byte(0x00,OLED_CMD);//---set low column address
    OLED_WR_Byte(0x10,OLED_CMD);//---set high column address
    OLED_WR_Byte(0x40,OLED_CMD);//--set start line address  Set Mapping
RAM Display Start Line (0x00~0x3F)
    OLED_WR_Byte(0x81,OLED_CMD);//--set contrast control register
    OLED_WR_Byte(0xCF,OLED_CMD); // Set SEG Output Current Brightness
    OLED_WR_Byte(0xA1,OLED_CMD);//--Set SEG/Column Mapping
0xa0???? 0xa1??
    OLED_WR_Byte(0xC8,OLED_CMD);//Set COM/Row Scan Direction
0xc0???? 0xc8??
    OLED_WR_Byte(0xA6,OLED_CMD);//--set normal display
```

```
    OLED_WR_Byte(0xA8,OLED_CMD);//--set multiplex ratio(1 to 64)
    OLED_WR_Byte(0x3f,OLED_CMD);//--1/64 duty
    OLED_WR_Byte(0xD3,OLED_CMD);//-set display offset Shift Mapping RAM Counter (0x00~0x3F)
    OLED_WR_Byte(0x00,OLED_CMD);//-not offset
    OLED_WR_Byte(0xd5,OLED_CMD);//--set display clock divide ratio/oscillator frequency
    OLED_WR_Byte(0x80,OLED_CMD);//--set divide ratio, Set Clock as 100 Frames/Sec
    OLED_WR_Byte(0xD9,OLED_CMD);//--set pre-charge period
    OLED_WR_Byte(0xF1,OLED_CMD);//Set Pre-Charge as 15 Clocks & Discharge as 1 Clock
    OLED_WR_Byte(0xDA,OLED_CMD);//--set com pins hardware configuration
    OLED_WR_Byte(0x12,OLED_CMD);
    OLED_WR_Byte(0xDB,OLED_CMD);//--set vcomh
    OLED_WR_Byte(0x40,OLED_CMD);//Set VCOM Deselect Level
    OLED_WR_Byte(0x20,OLED_CMD);//-Set Page Addressing Mode (0x00/0x01/0x02)
    OLED_WR_Byte(0x02,OLED_CMD);//
    OLED_WR_Byte(0x8D,OLED_CMD);//--set Charge Pump enable/disable
    OLED_WR_Byte(0x14,OLED_CMD);//--set(0x10) disable
    OLED_WR_Byte(0xA4,OLED_CMD);// Disable Entire Display On (0xa4/0xa5)
    OLED_WR_Byte(0xA6,OLED_CMD);// Disable Inverse Display On (0xa6/a7)
    OLED_WR_Byte(0xAF,OLED_CMD);//--turn on oled panel
    OLED_WR_Byte(0xAF,OLED_CMD); /*display ON*/
    OLED_Clear();
    delay_ms(500);
    // OLED_Pow(0,0);
}

int main()
{
    LED_Init();
    USART1_Init();
    Delay_Init(72);
    OLED_Init();
    OLED_ShowString(8,16,(u8*)"OLED TEST",16,1);
    OLED_Refresh();
    while(1)
    {
```

```
    }
}
```

4.实验步骤

4.1 编译并将程序下载到节点

在 KEIL 开发环境下打开 **OLED 显示实验\RVMDK** 下的 Test.uvprojx 工程，编译并下载到的节点上。

4.2 实验运行效果

将程序下载到节点板，节点板屏幕显示 OLED TEST。