1. 绪论

1.1 红外遥控的发展

二十世纪, 刚步入八十年代, 电视产品在日本快速发展, 相应的红外遥控技术也在这里得到了广泛应用。为了实现遥控码的发射, 他们使用了一种集成发射芯片, 这种芯片的特点是: 单一控制（内部预置固定编码的遥控器只能控制单一型号的电器）。

随着时代不断地进步, 在目前的日常家居生活里面, 一般家庭都会配置各种各样的家用电器, 相应的也会有各种各样的遥控器, 当需要遥控某一家用电器时, 需要从多个遥控器中选出特定的一个。当家用电器越来越多, 对应的遥控器也越来越多, 从众多的遥控器中选出所需要的遥控器也会更加麻烦, 过多的遥控器会占据相当一部分的家居空间, 也不方便整理。

针对这种情况, 人们便产生了一种想法: 是否可以用一只遥控器遥控所有家用电器[5]？答案是肯定的, 多用遥控器就此产生了, 这种遥控器的特点是: 遥控器里面预置了不止一套编码, 而是多套编码, 可供用户在不同场合选择不同模式。

伴随着科技的日新月异, 红外遥控技术和单片机技术已经越来越成熟, 嵌入式技术的大力发展, 使大部分商家都在生产一种新的遥控器: 这种遥控器内置一个动态的编码库, 库内存放用户自主录入的编码, 使其具备一中心的功能——红外学习功能, 这种遥控器也有了新的名称: 学习型红外遥控器。

1.2 红外通信原理概述

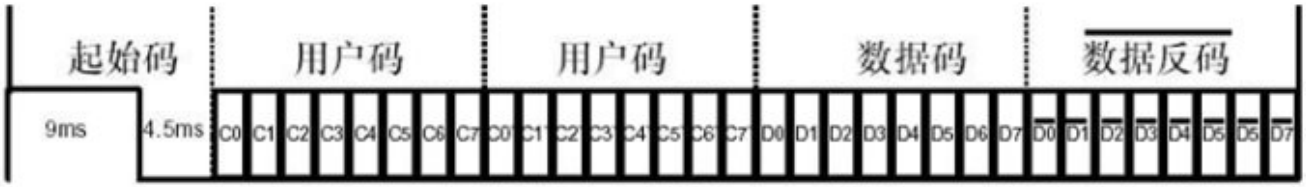
所谓红外通信, 就是通过对二进制数字的调节与调制, 来进行信号传输, 红外通信接口就是红外信道的调制解调器。利用红外技术, 实现两点间的近距离保密通信和信息转发, 称为红外通信。它一般由两部分组成, 即: 红外发射系统与红外接收系统。单片机本身并不具备红外通信接口, 我们可以利用红外发射电路与红外接收电路组成一个应用于单片机的红外通信接口。

红外通信的基本原理是通过红外发射管发射的将基带二进制信号调制为一系列的脉冲串信号载波信号。

在红外通信过程中调制和发送的数据共有三种常用的协议:

- NEC协议
- 夏普协议
- 索尼SIRC协议

其中应用最广泛的是NEC协议。NEC标准规定: 红外通信的载波频率为38KHz, 占空比为1: 3；数据格式包括了引导码、用户码、数据码和数据码反码, 编码总占 32 位。数据反码是 数据码反相后的编码, 编码时可用于对数据的纠错。注意: 第二段的用户码也可以在遥控应用电路中被设置成第一段用户码的反码, 如下图所示:



1.3 通用红外通信设备学码的实现

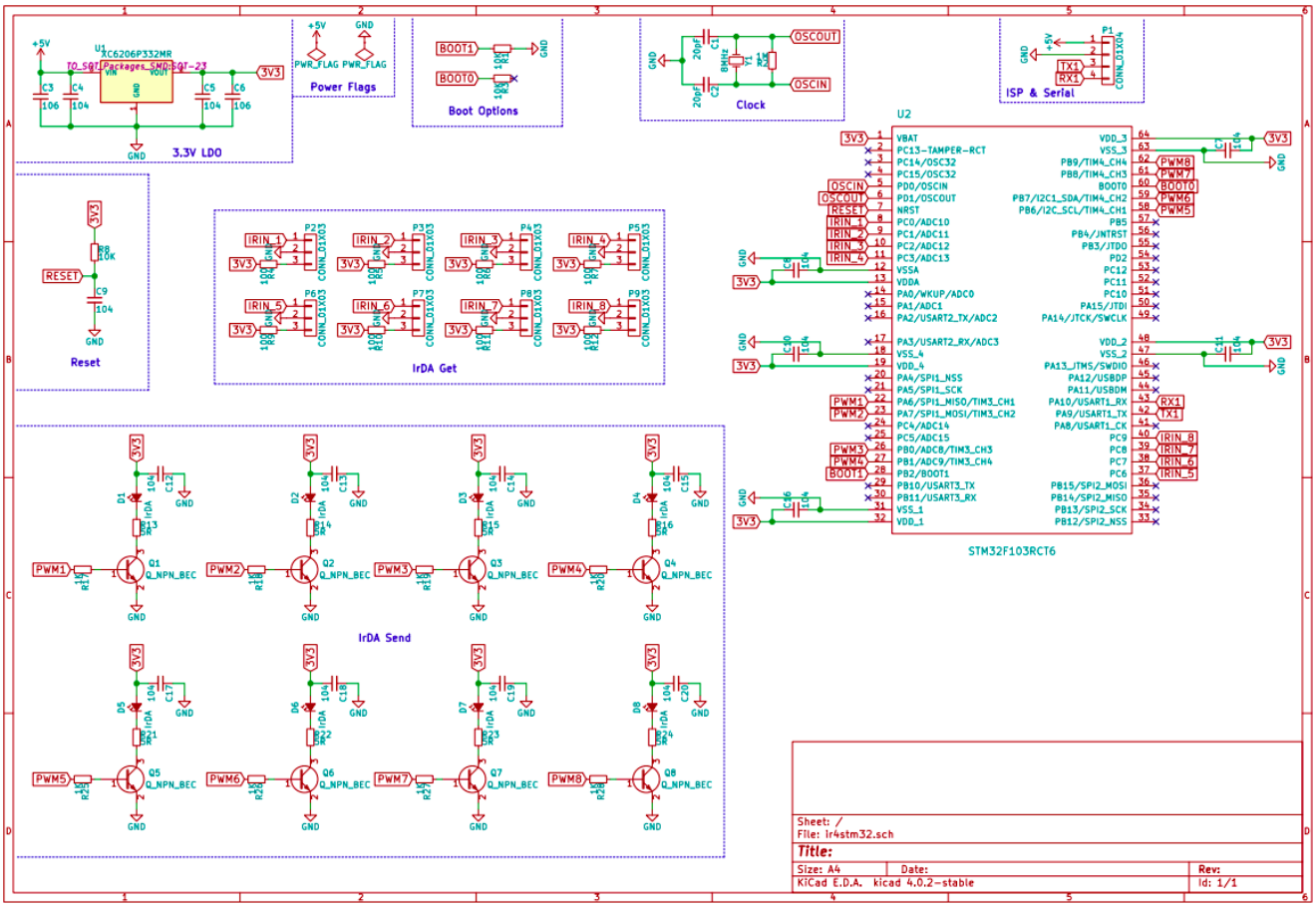
通用红外通信设备按照学习方式的不同可以分为两类:

- 固定码格式的通信设备
- 波形拷贝式的通信设备

第一种通信设备需要收集保存不同种类的红外设备信号, 然后识别比较, 最后再记录。这种红外设备的优点是硬件要求相对简易, 控制器的CPU频率可以较低；缺点是因为红外编码格式太多, 实现红外设备的成功复制比较难。

第二种通信设备是把原来红外设备发出的信号进行完全的复制, 不管原来红外设备红外信号是什么格式, 然后存储到非易失性的存储器(如EEPROM)中, 发送时再把保存的波形数据取出, 还原成原始信号。其优点是可以对任何一种红外设备进行学习；缺点是对控制器CPU的频率要求较高, RAM要大。 为了对尽可能多的设备进行控制, 本设计完成的是第二种红外通信设备。

2. 红外学码系统电路设计

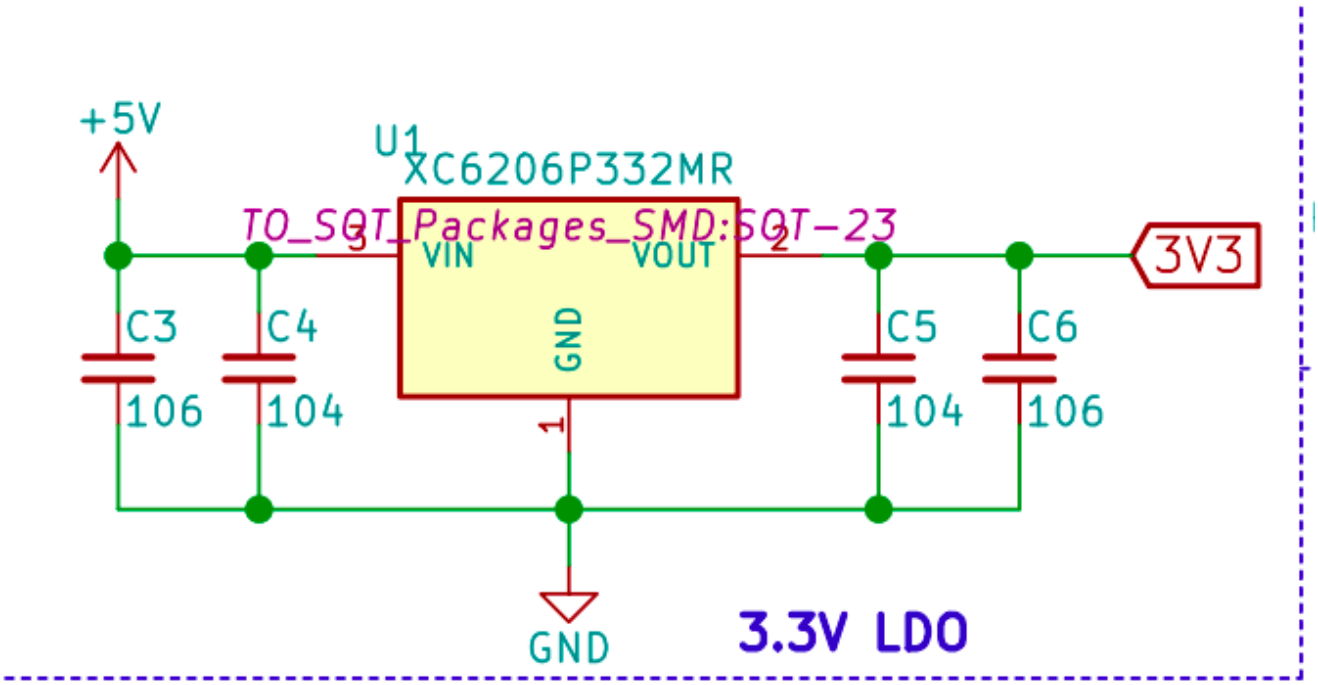


上图是该硬件系统的完整电路图, 该电路共分为 7 个部分:

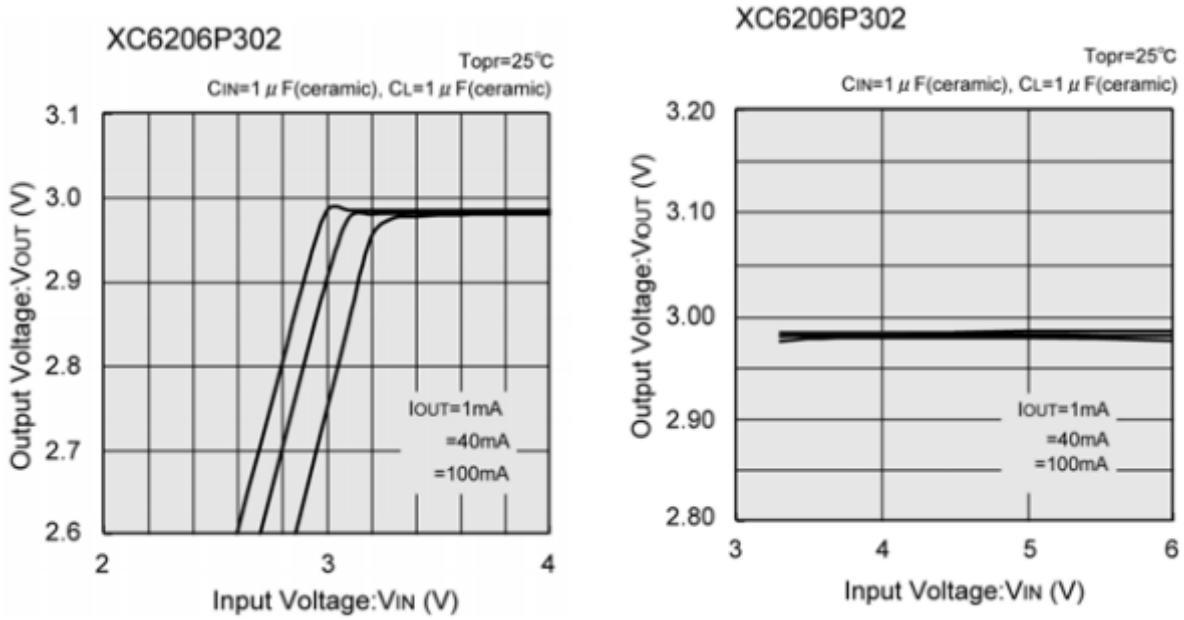
1. 稳压电源电路
2. 单片机启动配置电路
3. 单片机外部时钟晶振电路
4. 单片机上电复位电路
5. 8 路学码部分电路
6. 8 路发码部分电路
7. 电路板接口电路

2.1 稳压电源电路

稳压电源电路是用来将外部供电电源的电压转换成单片机所需的 3.3V 稳定的电压。



这里使用了型号为 XC6206P332MR 的稳压 LDO 芯片, LDO即 Low Dropout Regulator, 是一种低压差线性稳压器。并且在芯片的电源输入引脚和电源输出引脚分别接了两个滤波电容, 用来防止电源可能发生的波动, 提高数字电路工作的稳健性。



上图是该 LDO 芯片型号为 P302MR 的典型输入输出电压曲线, 横轴为输入电压, 纵轴为输出电压, 通过上图可以清楚的看出, 我们所使用的 P332MR 当输入电压在 3.4V ~ 6V 的范围内, 都可以输出稳定的 3.3V 电压。

(这张图表是XC6206P302的数据, 官方的数据手册中只有该型号的图表作为一个典型示例, 没有我们所使用的 P332MR, 所以图表中的输出电压不是3.3V)

2.2 单片机启动配置电路

在STM32F10xxx里, 可以通过BOOT[1: 0]引脚选择三种不同启动模式:

启动模式选择引脚		启动模式	说明
BOOT1	BOOT0		
X	0	主闪存存储器	主闪存存储器被选为启动区域
0	1	系统存储器	系统存储器被选为启动区域
1	1	内置SRAM	内置SRAM被选为启动区域

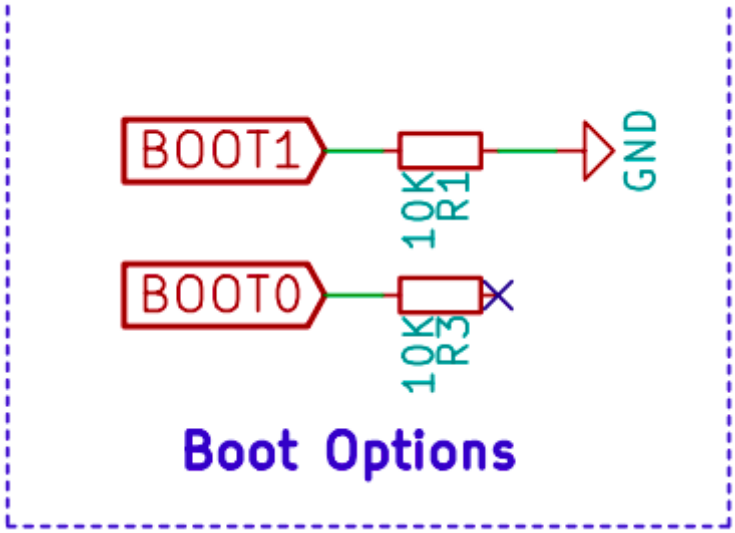
- 从主闪存存储器启动: 主闪存存储器被映射到启动空间 (0x0000 0000), 但仍然能够在它原有的地址 (0x0800 0000) 访问它, 即闪存存储器的内容可以在两个地址区域访问, 0x0000 0000 或 0x0800 0000。

  > 以这种模式启动会运行用户烧写的程序

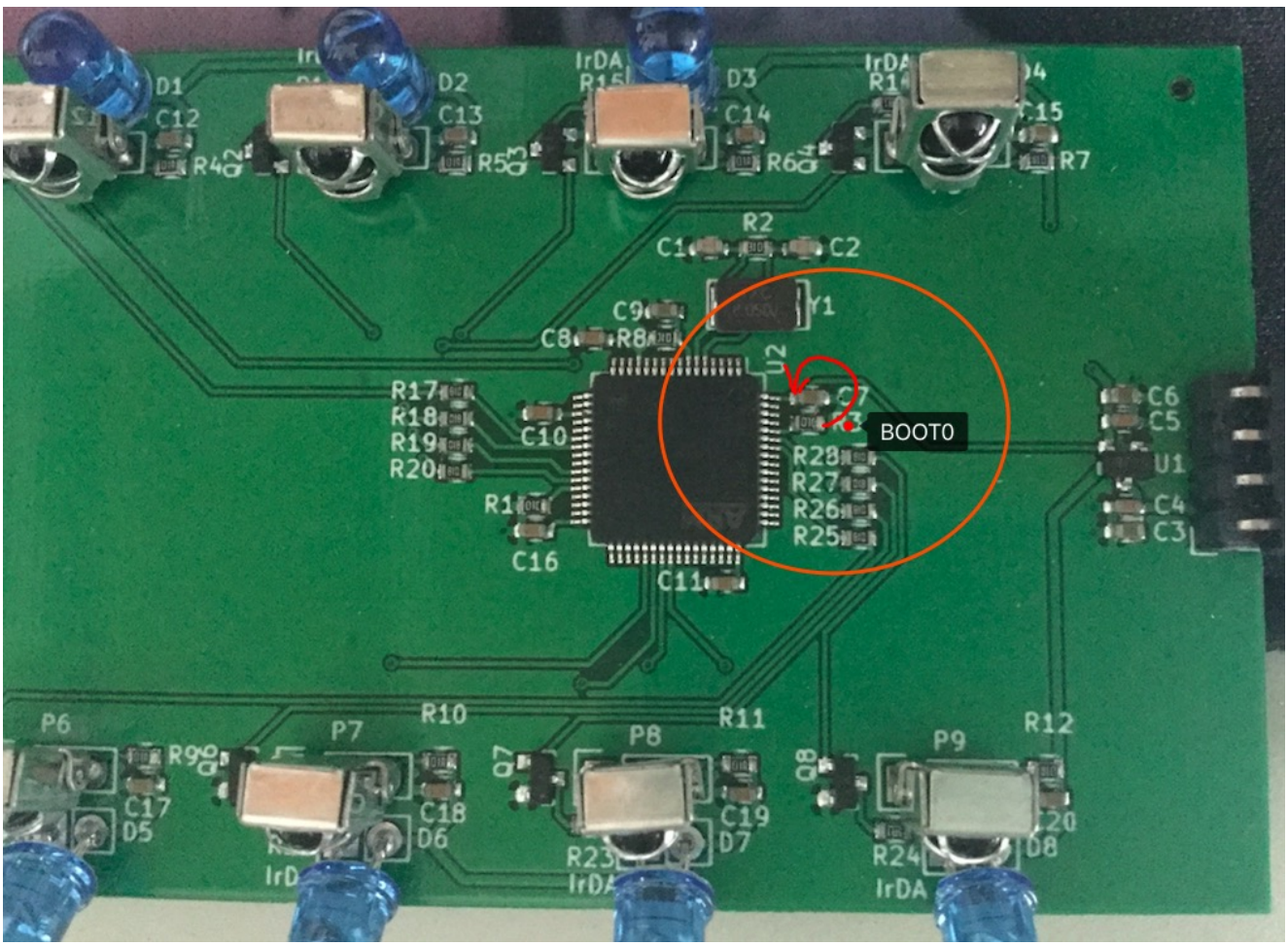
- 从系统存储器启动: 系统存储器被映射到启动空间 (0x0000 0000), 但仍然能够在它原有的地址(互联型产品原有地址为 0x1FFF B000, 其它产品原有地址为0x1FFF F000)访问它。

  > 以这种模式启动会运行单片机出厂自带的 ISP 程序(In System Program 在系统编程), 这时我们可以通过串口连接单片机来上传程序。

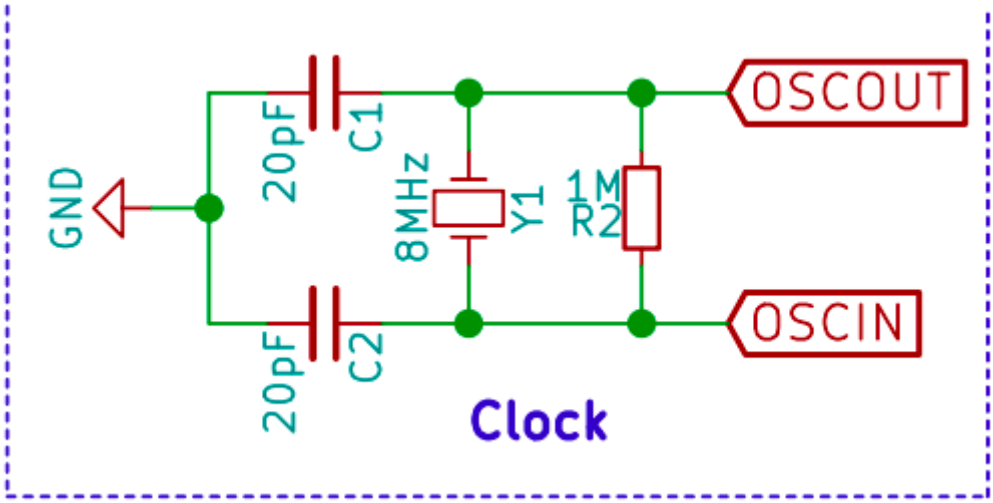
在系统复位后, SYSCLK 的第4个上升沿, BOOT 引脚的值将被锁存。用户可以通过设置 BOOT1 和 BOOT0 引脚的状态, 来选择在复位后的启动模式。



在这次设计中, 我们将 BOOT1 通过一个 10K 的限流电阻直接接地, 通过 BOOT0 来控制单片机的启动模式。在实际使用的过程中, 单片机默认会以第一种模式启动 (BOOT0 悬空时为低电平), 也就是运行用户的程序, 如果我们要烧写程序, 只需要在单片机供电时确保 BOOT0 为高电平, 如下图, 将电阻 R3 右侧短接上方电容 C7 左侧即可。



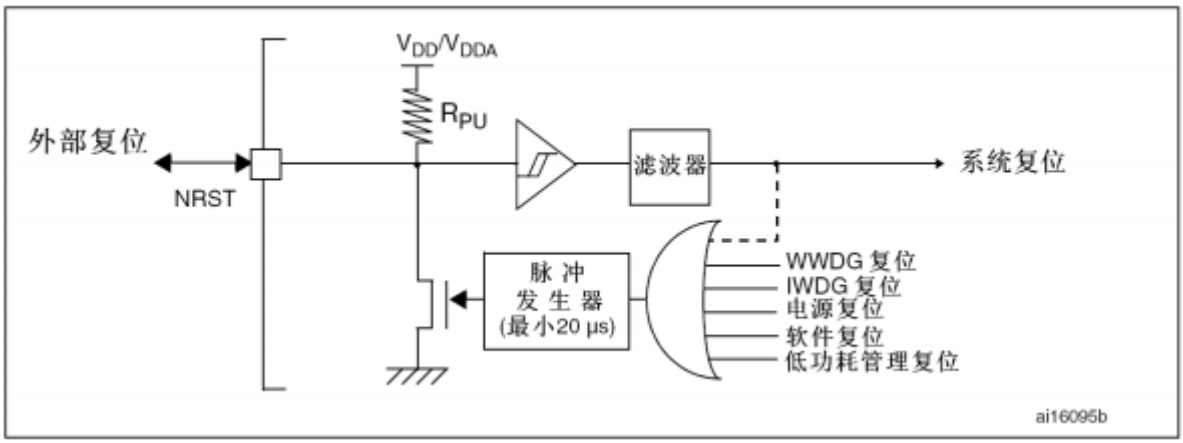
2.3 单片机外部时钟晶振电路



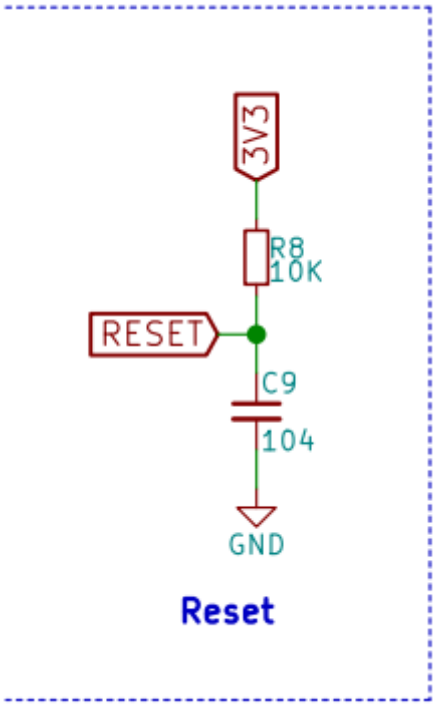
4~16Mz外部振荡器可为系统提供更为精确的主时钟, 在这里单片机的高速外部时钟信号 (HSE) 是由以下一个8Mhz晶振产生的。 晶振两个引脚分别连接单片机的外部时钟引脚, 并且连接了两个 20pF 的负载电容, 作用是对晶体和振荡电路的补偿和匹配, 使电路易于起振并处于合理的激励态下, 同时对振荡频率也有一定的微调作用; 同时晶振也并联了一个 1M 欧的电阻 R2, 起到分流作用, 使晶振的工作电流在一个合理的范围内。

2.4 单片机上电复位电路

STM32 支持三种复位形式, 分别为系统复位、上电复位和备份区域复位, 这里的电路图属于上电复位。

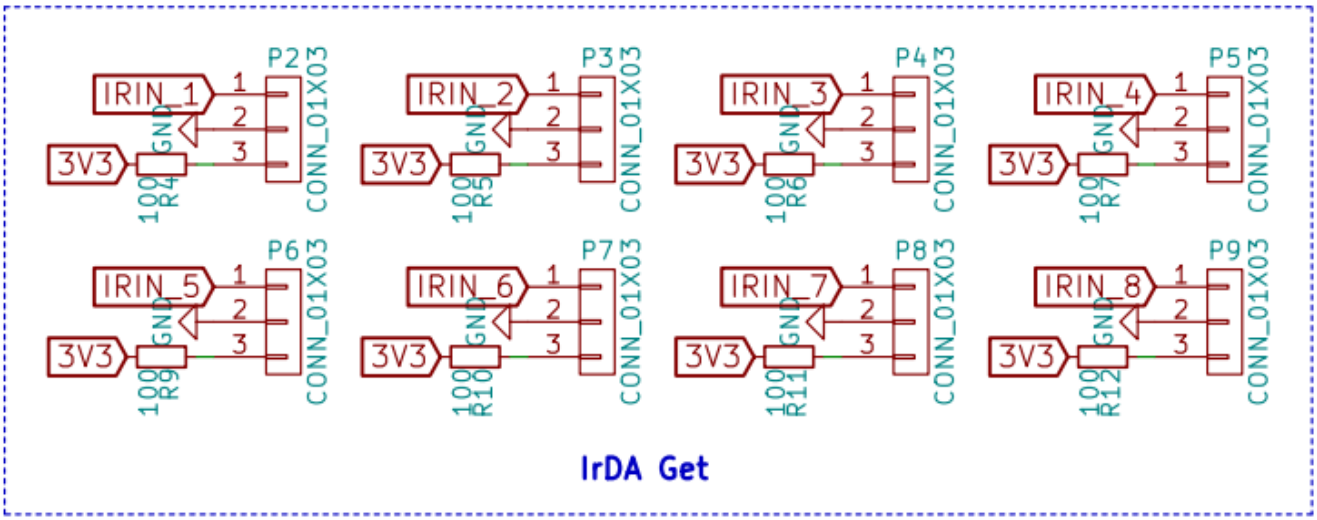


芯片内部的复位信号源是单片机的 NRST 引脚, 脉冲发生器保证每一个(外部或内部)复位源都能有 至少20μs的脉冲延时; 当 NRST 引脚被拉低产生外部复位时, 它将产生复位脉冲, 然后单片机将会复位除了备份区域外的所有寄存器, 也就是清除上次运行后单片机配置可能发生的改变, 目的是让系统回到一个固定的初始状态, 这时所有寄存器的值都是已知的, 所以在这之后对单片机的每一次操作的结果都是可以判断的。

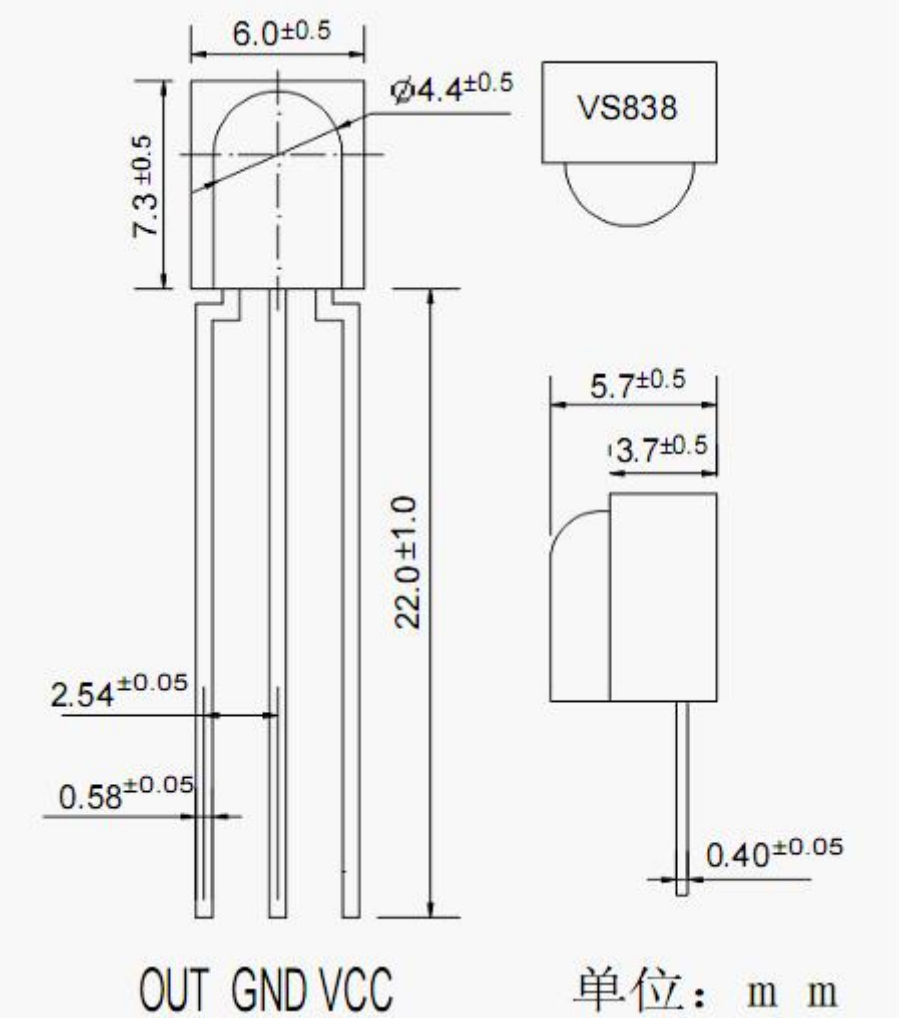


上图是单片机的上电复位电路, RESET 连接在单片机的 NRST 引脚, 在电路板开始供电时电容 C9 两侧是短路的, 也就是说这时 RESET 是直接接地的, 为低电平, 随着时间的推移, 电容将会被上拉的限流电阻 R8 充电, 随后 RESET 到 GND 之间就会形成断路, 这时 RESET 会变为高电平, 在整个过程也就中产生了低电平的复位信号。

2.5 8路学码部分电路

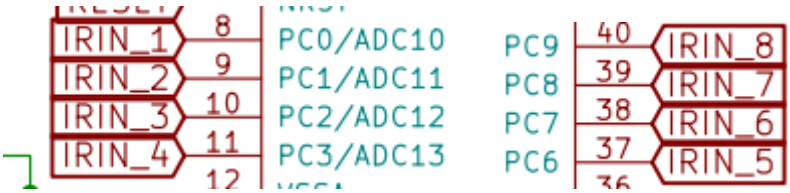


上图为 8 路的学码电路, 每一路都连接了一个 VS838 红外接收管:



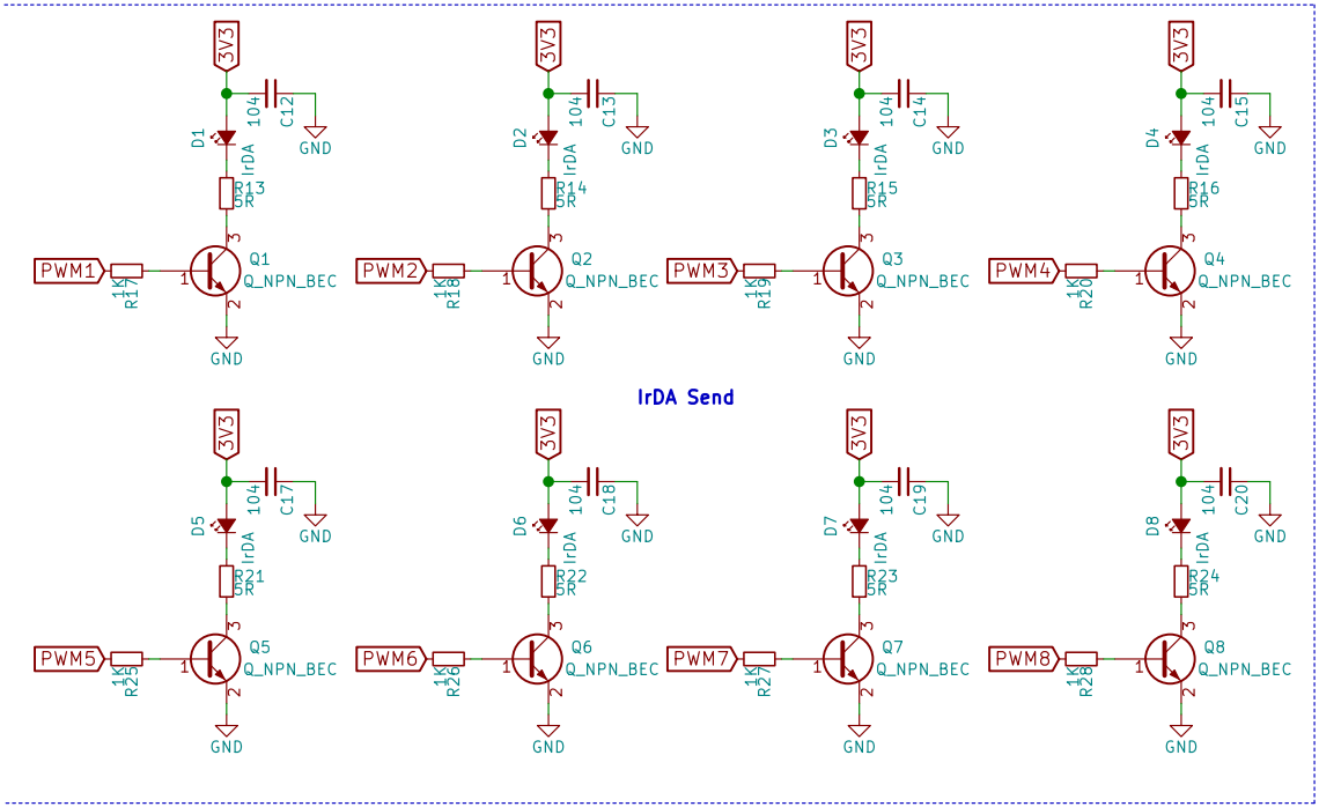
VS838 的电源管脚 VCC 通过一个限流电阻接电源。

VS838 的 OUT 管脚是信号输出管脚, 当 VS838 接受到红外信号时 OUT 管脚会由高电平变为低电平, 直到红外信号消失。(未收到红外信号时 OUT 脚一直为高电平)



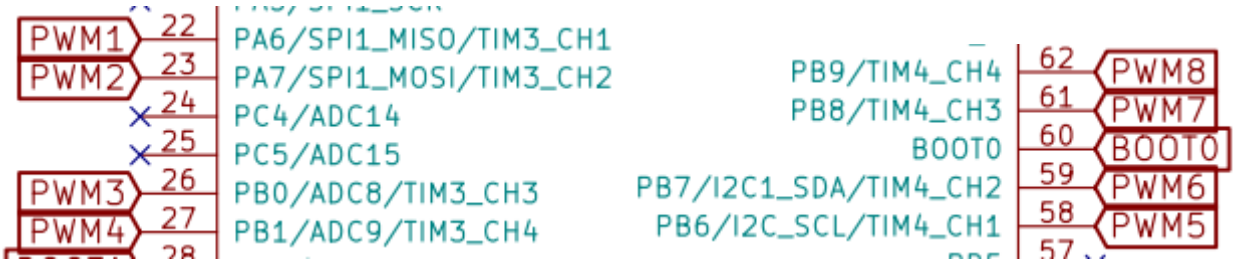
红外信号输出管脚是单片机获取红外信号的唯一来源, 所以该管脚连接在单片机的 GPIO 上, 当收到红外信号时该管脚产生的电平下降沿会被单片机的 GPIO 捕获并产生外部中断, 在中断函数中我们会记录该波形也就是红外信号的波形长度, 从而进行学码。

2.6 8路发码部分电路



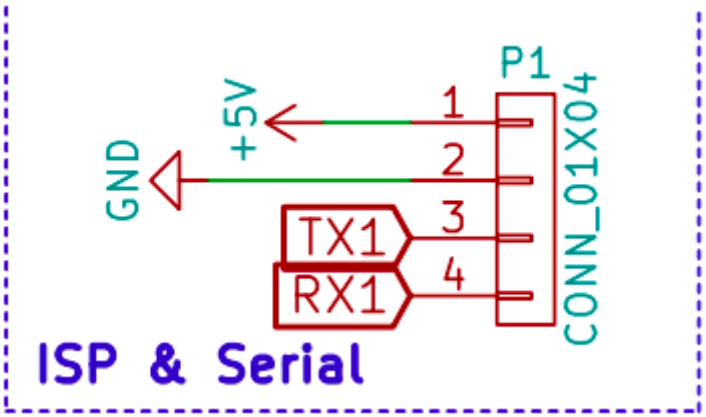
上图为 8 路的发码电路, NEC标准规定: 红外通信的载波频率为38KHz, 占空比为1: 3,

这里我们使用单片机定时器的 PWM 输出功能产生所需的 38KHz 1: 3 波形, 通过对 PWM 输出的打开和关闭来实现载波。PWM 信号管脚通过一个限流电阻连接一个 NPN 三极管的, 通过控制三极管来打开和关闭红外 LED 进行红外信号的发送。



PWM 信号分别由单片机 TIM3 的四个通道和 TIM4 的四个通道产生。

2.7 电路板接口电路



该电路板的接口共有 4 个针脚:

- 外部电源供电正极
- 外部电源供电负极
- 串口信号线 TX
- 串口信号线 RX

其中串口有两个作用:

- 串口控制: 电脑通过串口与电路板连接, 发送指令以控制单片机进行学码和发码操作
- ISP: 电脑通过串口连接单片机来烧写程序。

3. 串口控制的8路学码系统的软件原理

3.1 主函数

```
int main() {
    // 配置中断优先级分组
    NVIC_SetPriorityGrouping(0x07 - NVIC_GROUPING);

    uart_init(72, 115200);     // 初始化串口, 波特率为115200
    irda_init();              // irda初始化
    while(1) {
        // uart_sendStr("Alive~");
        // UART_CR();
        // delay_ms(1000);
    }
    return 0;
}
```

在系统供电之后单片机会进行下面这些操作:

- 配置中断优先级分组
- 串口功能初始化
- 8路红外发码收码系统的初始化
- 死循环, 等待用户的进一步操作

3.2 配置中断优先级分组

该系统的主要功能如串口控制、红外信号的捕获都依赖于单片机中的中断功能, 所以在程序启动后我们首先要做的就是单片机中断相关的配置。

STM32单片机使用的 Cortex-M3 内核中搭载了一个中断响应系统, 支持为数众多的系统异常和外部中断。当系统存在多个中断时, 优先级对于中断来说就很关键了, 它会决定一个中断是否能被掩蔽, 以及在未掩蔽的 情况下何时可以响应。优先级的数值越小, 则优先级越高。CM3 支持中断嵌套, 使得高优先级中断 会抢占 (preempt) 低优先级中断。有 3 个系统异常: 复位, NMI

以及硬 fault, 它们有固定的优先 级, 并且它们的优先级号是负数, 从而高于所有其它异常。所有其它异常的优先级则都是可编程的。

为了使抢占机能变得更可控, CM3 还把 256 级优先级按位分成高低两段, 分别称 为抢占优先级和子优先级。

并且在 NVIC 中有一个寄存器是 “应用程序中断及复位控制寄存器” (AIRCR), 它里面有一个位段名为 “优先级组”。该位段的值对每一个优先级可配置的异常都有影响 —— 把其优先级分为 2 个 位段: MSB 所在的位段（左边的）对应抢占优先级, 而 LSB 所在的位段（右边的）对应子优先级。

分组位置	表达抢占优先级的位段	表达子优先级的位段
0	[7:1]	[0:0]
1	[7:2]	[1:0]
2	[7:3]	[2:0]
3	[7:4]	[3:0]
4	[7:5]	[4:0]
5	[7:6]	[5:0]
6	[7:7]	[6:0]
7	无	[7:0]（所有位）

[10:8]	PRIGROUP	中断优先级分组域： PRIGROUP 从子优先级中拆分强占式优先级 0 7.1 表示 7 位抢占式优先级，1 位子优先级 1 6.2 表示 6 位抢占式优先级，2 位子优先级 2 5.3 表示 5 位抢占式优先级，3 位子优先级 3 4.4 表示 4 位抢占式优先级，4 位子优先级 4 3.5 表示 3 位抢占式优先级，5 位子优先级 5 2.6 表示 2 位抢占式优先级，6 位子优先级 6 1.7 表示 1 位抢占式优先级，7 位子优先级 7 0.8 表示 0 位抢占式优先级，8 位子优先级 PRIGROUP 域是一个二进制小数点定位指示器，用于为共用同一抢占级别的异常创建优先级。它将中断优先级的 PRI_n 域分成抢占式优先级和子优先级。二进制小数点是一个偏左值。即 PRIGROUP 值代表一个从 LSB 左边开始的小数值。这是 7:0 的位 0。 最低的值不能为 0，这取决于为优先级分配的位数以及设备的选择

下图是应用程序中断及复位控制寄存器 (AIRCR) 地址: 0xE000_ED0C 的详细说明

位段	名称	类型	复位值	描述
31:16	VECTKEY	RW	-	访问钥匙：任何对该寄存器的写操作，都必须同时把 0x05FA 写入此段，否则写操作被忽略。若读取此半字，则 0xFA05
15	ENDIANESS	R	-	指示端设置。1=大端(BE8)，0=小端。此值是在复位时确定的，不能更改。
10:8	PRIGROUP	R/W	0	优先级分组
2	SYSRESETREQ	W	-	请求芯片控制逻辑产生一次复位
1	VECTCLRACTIVE	W	-	清零所有异常的活动状态信息。通常只在调试时用，或者在 OS 从错误中恢复时用。
0	VECTRESET	W	-	复位 CM3 处理器内核（调试逻辑除外），但是此复位不影响芯片上在内核以外的电路

我们所使用的 STM32 单片机的 PRIGROUP 只是用了4bit来表示:

Table 45. Priority grouping

PRIGROUP [2:0]	Interrupt priority level value, PRI_N[7:4]			Number of	
	Binary point(1)	Group priority bits	Subpriority bits	Group priorities	Sub priorities
0b011	0bxxxx	[7:4]	None	16	None
0b100	0bxxx.y	[7:5]	[4]	8	2
0b101	0bxx.yy	[7:6]	[5:4]	4	4
0b110	0bx.yyy	[7]	[6:4]	2	8
0b111	0b.yyyy	None	[7:4]	None	16

1. PRI_n[7:4] field showing the binary point. x denotes a group priority field bit, and y denotes a subpriority field bit.

通过上图我们可以得到一个规律, 即 PRIGROUP 的值等于 0x07 - 优先级组别。

在主函数中我们会通过下面这行语句将中断优先级分组设置为3, 在寄存器中: xxx.y (x 指抢占式优先级, y 指响应式优先级):

```
#define NVIC_GROUPING    3

NVIC_SetPriorityGrouping(0x07 - NVIC_GROUPING);
```

其中调用了 NVIC_SetPriorityGrouping 函数, 该函数用来设置 AIRCR 寄存器的 PPRIGROUP 位, 该函数的定义位于 STM32 官方的头文件 core_cm3.h 中:

```
/* \brief  Set Priority Grouping

  The function sets the priority grouping field using the required unlock seque
  The parameter PriorityGroup is assigned to the field SCB->AIRCR [10: 8] PRIGR
  Only values from 0..7 are used.
  In case of a conflict between priority grouping and available
  priority bits (__NVIC_PRIO_BITS), the smallest possible priority group is set

    \param [in]      PriorityGroup  Priority grouping field.
 */
__STATIC_INLINE void NVIC_SetPriorityGrouping(uint32_t PriorityGroup)
{
    uint32_t reg_value;
    uint32_t PriorityGroupTmp = (PriorityGroup & (uint32_t)0x07);
    // only values 0..7 are used

    reg_value  =  SCB->AIRCR;
    // read old register configuration
    reg_value &= ~(SCB_AIRCR_VECTKEY_Msk | SCB_AIRCR_PRIGROUP_Msk);
    // clear bits to change
    reg_value  =  (reg_value |
        ((uint32_t)0x5FA << SCB_AIRCR_VECTKEY_Pos) |
        (PriorityGroupTmp << 8));
    // Insert write key and priorty group
    SCB->AIRCR =  reg_value;
}
```

3.3 串口功能初始化

在配置好中断优先级分组后, 会初始化单片机的 USART1 串口, 配置其波特率为 115200, 并打开串口中断

```
uart_init(72, 115200);    // 初始化串口, 波特率为115200
```

下面是该函数的定义:

```
void uart_init(unsigned int pclk2, unsigned int bound) {
    float temp;
    unsigned short mantissa;
    unsigned short fraction;
    temp = (float)(pclk2*1000000)/(bound*16);
    mantissa = temp;
    fraction = (temp - mantissa) * 16;
    mantissa <<= 4;
    mantissa += fraction;
    RCC->APB2ENR |= RCC_APB2ENR_IOPAEN;         // 打开 GPIOA 时钟
    RCC->APB2ENR |= RCC_APB2ENR_USART1EN;     // 打开 USART1 时钟

    GPIOA->CRH &= 0xFFFFF00F;                  // USART GPIO 引脚配置
    GPIOA->CRH |= 0x000008B0;

    RCC->APB2RSTR |= RCC_APB2RSTR_USART1RST;
```

```
    RCC->APB2RSTR &= ~RCC_APB2RSTR_USART1RST;

    USART1->BRR = mantissa;
    USART1->CR1 |= 0x200C;    //设置 UE, TE and RE

    USART1->CR1 |= 1<<8;    //PE 中断使能
    USART1->CR1 |= 1<<5;    //RX 非空中断使能


    USART1->SR;     // 读取 SR 寄存器以将 TXE 和 TE 位清零,(复位值:  0x00C0)

    // SCB->AIRCR &= 0x05FAF8FF;    // AIRCE Key:  0x05FA
    // SCB->AIRCR |= 0x05FA0400;    // Set up group value
    NVIC_EnableIRQ(USART1_IRQn);    // 使能串口中断
    NVIC_SetPriority(USART1_IRQn, 0b1001);//设置其优先级
}
```

该函数使能了单片机的串口中断, 当单片机收到用户从电脑发送的字符后, 会触发串口中断并执行下面的中断处理函数 USART1_IRQHandler :

```
void USART1_IRQHandler(void) {
    if(USART1->SR & USART_SR_RXNE) {
        const char cmd = USART1->DR;    // 读取串口接收寄存器来清除 RXNE 标志
        switch (cmd) {
            case 0x0D:      //回车键
            case 0x0A:
                uart_sendStr("\n\r当前命令: \t");
                uart_sendStr(gCmdCache);
                UART_CR();
                uart_decode(gCmdCache);
                clrCache();
                break;
            case 0x08:      //退格键
            case 0x7F:
                if(top>=0) {        // 防止指针跨域操作, 否则多次按退格键会出现bug
                    pop = '\0';
                    uart_sendData(0x7F);
                    uart_sendData(0x08);
                }
                break;
            case TOKEN_START:     //$ - 命令起始标志
                clrCache();
            default:      //其它按键
                if(STACK_OVERFLOW)    //如果指令缓存将要溢出, 则不会入栈当前字符
                    break;
                push(cmd);            //保存当前字符
                uart_sendData(cmd);    //在终端回显, 反馈用户输入的字符
                break;
        }
    }
}
```

我们将用户的输入分为四种:

- 串口命令的起始标志: $
- 普通字符, 比如字母和数字
- 回车键
- 删除键

在上面的源码中可以清楚的看到, 当单片机收到第一种输入也就是 '$' 时, 单片机会清空用于记录串口控制指令的数组。

当单片机收到第二种输入时, 单片机会记录并保存用户当前输入的字符到用于记录串口指令的数组中, 并把当前字符通过串口发送给电脑, 也就是在终端中反馈给用户。

当单片机收到第三种输入即回车键时, 单片机会通过调用 uart_decode 函数来解析当前指令并执行相应的操作。

当单片机收到第四种输入即删除键时, 单片机会在串口控制指令数组中删除最后的一个字符, 并且通过串口输出删除键的键值, 以删除电脑终端上的字符, 这样就实现了删除字符的功能。

3.3.1 串口控制指令的规则定义

我们在头文件 uart.h 中实现了串口指令的规则的宏定义:

```
#define TOKEN_START         '$'
#define TOKEN_SEND         'S'
#define TOKEN_LEARN         'L'
#define TOKEN_OFFSET         0x01
#define CMD_NUM_MAX         '8'
#define CMD_NUM_MIN         '1'
#define ISLEGAL_NUM(k)     (((k) >= CMD_NUM_MIN) && ((k) <= CMD_NUM_MAX))
```

在规则中定义了下面三种特殊标志:

- $ 字符作为串口指令的起始标志, 当用户输入 $ 时意味着一条指令的开始, 在 $ 之前输入的所有字符都将被删除, 在 $ 之后输入的所有字符都将作为串口指令被记录;
- **S** (Send)字符作为红外发码指令的标志, 该标志应当紧随着命令起始标志 $, 例如:
  - $S1 - 1号红外外设执行发码操作
  - $S123 - 1号、2号、3号外设执行发码操作
  - $S85 - 8号、5号外设执行发码操作
- **L** (Learn)字符作为红外收码状态切换 (Toggle) 指令, 状态切换指的是如果被操作外设的收码状态是关闭的则执行该指令后会被打开, 如果是打开的则会被关闭, 同样该标志也要紧随着命令起始标志 $, 例如:
  - $L2 - 2号红外外设收码状态切换
  - $L345 - 3号、4号、5号红外外设收码状态切换

在规则中定义了操作标志 S 和 L 在指令中的便宜值, 当前的规则是操作标志的位置必须在指令的第二个字符, 也就是紧随起始标志之后。

在规则中定义了指令中可能出现的数字的最大值以及最小值，以限制数字的合法范围，防止非法指令造成的越界操作，提高该系统的稳健性。该范围取决于本系统的红外外设数量，在这里我们共有8路外设，所以其合法范围是 1 到 8。

3.3.2 串口控制指令的解析

当用户在终端中键入回车键意味着一条指令的结束，这时该指令会通过 uart_decode 函数来进行解析，该函数的定义在 uart.c 文件中，下面是该函数的源码：

```
void uart_decode(char *token) {
    if(*token == 0)
        return;    //如果发生越界则结束 decode
    if(ISLEGAL_NUM(*token)) {//如果当前操作符为效数字

        uart_sendStr(" - ");
        uart_sendData(*token);//在终端显示当前处理的学码电路通道号
        // 如果这条指令是学码命令
        if(gCmdCache[TOKEN_OFFSET] == TOKEN_LEARN) {
            // (Toggle between enable and disable) 如果该路学码中断是关闭的则使能，反
            *g_IrDA_Device[*token - '1'].IrInterrup ^= 1;
            uart_sendStr("号: 学码");
            uart_sendStr(*g_IrDA_Device[*token - '1'].IrInterrup?"开启\n\r": "关
        }
        else if(gCmdCache[TOKEN_OFFSET] == TOKEN_SEND) {// 如果这条指令是发码命令
            uart_sendStr("号: 发码开启\n\r");
            // 在发码的过程中为了防止自发自收，得确保所有的接收中断都是关闭的，否则中断会干
            *g_IrDA_Device[0].IrInterrup =
                *g_IrDA_Device[1].IrInterrup =
                *g_IrDA_Device[2].IrInterrup =
                *g_IrDA_Device[3].IrInterrup =
                *g_IrDA_Device[4].IrInterrup =
                *g_IrDA_Device[5].IrInterrup =
                *g_IrDA_Device[6].IrInterrup =
                *g_IrDA_Device[7].IrInterrup = 0;
            irda_encode(&g_IrDA_Device[*token - '1']);//发码
        }
    }
    uart_decode(++token);
}
```

从上面的代码中我们不难看出，该函数一次只判断一次字符，当该字符为合法数字时根据当前指令中的操作标志对该数字代表的外设执行相应的操作，然后再次调用该函数，处理下一个字符，以实现对当前串口指令中每一个字符的遍历。

3.4 八路红外发码收码系统的初始化

```
void irda_init() {
    irda_PWM_Init();     // 发送功能初始化
    irda_EXTI_Init();     // 接收功能初始化
```

```
    //实例化红外外设对象 - 第 1 路
    g_IrDA_Device[0].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 0);
    g_IrDA_Device[0].IrPWM        = BIT_ADDRP(&(TIM3->CCER), 0);
    g_IrDA_Device[0].signal        = BIT_ADDRP(&(GPIOC->IDR), 0);

    //实例化红外外设对象 - 第 2 路
    g_IrDA_Device[1].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 1);
    g_IrDA_Device[1].IrPWM        = BIT_ADDRP(&(TIM3->CCER), 4);
    g_IrDA_Device[1].signal        = BIT_ADDRP(&(GPIOC->IDR), 1);

    //实例化红外外设对象 - 第 3 路
    g_IrDA_Device[2].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 2);
    g_IrDA_Device[2].IrPWM        = BIT_ADDRP(&(TIM3->CCER), 8);
    g_IrDA_Device[2].signal        = BIT_ADDRP(&(GPIOC->IDR), 2);

    //实例化红外外设对象 - 第 4 路
    g_IrDA_Device[3].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 3);
    g_IrDA_Device[3].IrPWM        = BIT_ADDRP(&(TIM3->CCER), 12);
    g_IrDA_Device[3].signal        = BIT_ADDRP(&(GPIOC->IDR), 3);

    //实例化红外外设对象 - 第 5 路
    g_IrDA_Device[4].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 6);
    g_IrDA_Device[4].IrPWM        = BIT_ADDRP(&(TIM4->CCER), 0);
    g_IrDA_Device[4].signal        = BIT_ADDRP(&(GPIOC->IDR), 6);

    //实例化红外外设对象 - 第 6 路
    g_IrDA_Device[5].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 7);
    g_IrDA_Device[5].IrPWM        = BIT_ADDRP(&(TIM4->CCER), 4);
    g_IrDA_Device[5].signal        = BIT_ADDRP(&(GPIOC->IDR), 7);

    //实例化红外外设对象 - 第 7 路
    g_IrDA_Device[6].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 8);
    g_IrDA_Device[6].IrPWM        = BIT_ADDRP(&(TIM4->CCER), 8);
    g_IrDA_Device[6].signal        = BIT_ADDRP(&(GPIOC->IDR), 8);

    //实例化红外外设对象 - 第 8 路
    g_IrDA_Device[7].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 9);
    g_IrDA_Device[7].IrPWM        = BIT_ADDRP(&(TIM4->CCER), 12);
    g_IrDA_Device[7].signal        = BIT_ADDRP(&(GPIOC->IDR), 9);
}
```

上面是红外外设初始化函数, 该函数执行了下面几个操作:

- 发码功能初始化, 也就是 PWM 初始化
- 收码功能初始化, 也就是外部中断初始化
- 分别实例化 8 路红外外设对象

3.4.1 发码功能初始化

```
//红外学码电路发码部分初始化函数
void irda_PWM_Init() {
    // 红外发射管分别依次连接在单片机的 TIM3_CH1, TIM3_CH2, TIM3_CH3, TIM3_CH4,
    //         TIM4_CH1, TIM4_CH2, TIM4_CH3, TIM4_CH4
    //     所以在这个函数中需要初始化 TIM3 和 TIM4
```

```
    RCC->APB1ENR |= RCC_APB1ENR_TIM3EN | RCC_APB1ENR_TIM4EN;    //TIM3 Enable
    RCC->APB2ENR |= RCC_APB2ENR_IOPAEN | RCC_APB2ENR_IOPBEN;    //IO Port A and

    // GPIOA GPIOB 寄存器配置
    GPIOA->CRL &= 0x00FFFFFF;
    GPIOA->CRL |= 0xBB000000;   //将 6, 7 脚配置为复用推挽输出
    GPIOA->ODR |= 1<<7;            //TIM3_CH2 GPIO 配置
    GPIOA->ODR |= 1<<6;            //TIM3_CH1 GPIO 配置

    GPIOB->CRL &= 0x00FFFF00;
    GPIOB->CRL |= 0xBB0000BB;    //将 0, 1, 6, 7 脚配置为复用推挽输出

    GPIOB->CRH &= 0xFFFFFF00;
    GPIOB->CRH |= 0x000000BB;    //将 8, 9 脚配置为复用推挽输出

    GPIOB->ODR |= 1<<0 | 1<<1 | 1<<6 | 1<<7 | 1<<8 | 1<<9;

    // TIM3 寄存器配置
    TIM3->ARR = IR_PWM_ARR - 1;    //配置计数器最大值
    TIM3->PSC = IR_PWM_PSC - 1;    //配置计数器分频

    TIM3->CCMR1 |= 6<<4;            //CH1 设置为 OC1M[2: 0]:  PWM 模式
    TIM3->CCMR1 |= TIM_CCMR1_OC1PE;    //CH1 设置为 OC1PE:  使能
    TIM3->CCMR1 |= 6<<12;           //CH2 设置为 OC2M[2: 0]:  PWM 模式
    TIM3->CCMR1 |= TIM_CCMR1_OC2PE;    //CH2 设置为 OC2PE:  使能
    TIM3->CCMR2 |= 6<<4;            //CH3 设置为 OC3M[2: 0]:  PWM 模式
    TIM3->CCMR2 |= TIM_CCMR2_OC3PE;    //CH3
    TIM3->CCMR2 |= 6<<12;            //CH4
    TIM3->CCMR2 |= TIM_CCMR2_OC4PE;    //CH4

    TIM3->CCR1 = TIM3->CCR2 = TIM3->CCR3 = TIM3->CCR4 = IR_PWM_CCR_DEF;
    //CH1 ~ CH4 计数器使能
    // TIM3->CCER |= TIM_CCER_CC1E | TIM_CCER_CC2E | TIM_CCER_CC3E | TIM_CCER_C

    TIM3->CR1 |= TIM_CR1_ARPE | TIM_CR1_CEN;    //配置APRE位并使能计数器

    // TIM4 寄存器配置
    TIM4->ARR = IR_PWM_ARR - 1;    //配置计数器最大值
    TIM4->PSC = IR_PWM_PSC - 1;    //配置计数器分频

    TIM4->CCMR1 |= 6<<4;            //CH1 设置为 OC1M[2: 0]:  PWM 模式
    TIM4->CCMR1 |= TIM_CCMR1_OC1PE;    //CH1 设置为 OC1PE:  使能
    TIM4->CCMR1 |= 6<<12;           //CH2 设置为 OC2M[2: 0]:  PWM 模式
    TIM4->CCMR1 |= TIM_CCMR1_OC2PE;    //CH2 设置为 OC2PE:  使能
    TIM4->CCMR2 |= 6<<4;            //CH3 设置为 OC3M[2: 0]:  PWM 模式
    TIM4->CCMR2 |= TIM_CCMR2_OC3PE;    //CH3
    TIM4->CCMR2 |= 6<<12;            //CH4
    TIM4->CCMR2 |= TIM_CCMR2_OC4PE;    //CH4

    TIM4->CCR1 = TIM4->CCR2 = TIM4->CCR3 = TIM4->CCR4 = IR_PWM_CCR_DEF;
    //CH1 ~ CH4 计数器使能
    // TIM4->CCER |= TIM_CCER_CC1E | TIM_CCER_CC2E | TIM_CCER_CC3E | TIM_CCER_C

    TIM4->CR1 |= TIM_CR1_ARPE | TIM_CR1_CEN;    //配置APRE位并使能计数器
}
```

红外发射管分别依次连接在单片机的 TIM3_CH1、TIM3_CH2、TIM3_CH3、TIM3_CH4、TIM4_CH1、TIM4_CH2、TIM4_CH3、TIM4_CH4, 所以在这个函数中初始化了 TIM3 和 TIM4 的 PWM 输出功能, 并且按照 NEC 协议标准将其 PWM 配置为 38KHz 1: 3占空比。

3.4.2 收码功能初始化

```
//红外学码电路收码部分初始化函数
void irda_EXTI_Init() {
    //使能外部中断
    NVIC_EnableIRQ(EXTI0_IRQn);
    NVIC_EnableIRQ(EXTI1_IRQn);
    NVIC_EnableIRQ(EXTI2_IRQn);
    NVIC_EnableIRQ(EXTI3_IRQn);
    NVIC_EnableIRQ(EXTI9_5_IRQn);

    NVIC_SetPriority(EXTI0_IRQn, 0b0011);     //设置中断优先级
    NVIC_SetPriority(EXTI1_IRQn, 0b0011);     //设置中断优先级
    NVIC_SetPriority(EXTI2_IRQn, 0b0011);     //设置中断优先级
    NVIC_SetPriority(EXTI3_IRQn, 0b0011);     //设置中断优先级
    NVIC_SetPriority(EXTI9_5_IRQn, 0b0011);     //设置中断优先级



    // 红外接收管的数据输出分别连接在单片机GPIO端口 C 的0, 1, 2, 3, 6, 7, 8, 9 引脚

    RCC->APB2ENR |= RCC_APB2ENR_AFIOEN;     //使能 AFIO 时钟, 因为中断属于复用功能
    RCC->APB2ENR |= RCC_APB2ENR_IOPCEN;     //使能 IO Port C 时钟
//                      76543210
    GPIOC->CRL &= 0x00FF0000;     //清空 0, 1, 2, 3, 6, 7
    GPIOC->CRL |= 0x88008888;     //配置为输入模式

    GPIOC->CRH &= 0xFFFFFF00;     //清空 8, 9
    GPIOC->CRH |= 0x00000088;     //配置为输入模式

    GPIOC->ODR |= 1 | 1<<1 | 1<<2 | 1<<3 | 1<<6 | 1<<7 | 1<<8 | 1<<9;     //上拉

    AFIO->EXTICR[0] = 0x2222;     //使能C端口 0, 1, 2, 3 引脚的中断复用
    AFIO->EXTICR[1] = 0x2200;     //使能C端口 6, 7 引脚的中断复用
    AFIO->EXTICR[2] = 0x0022;     //使能C端口 8, 9 引脚的中断复用

    EXTI->FTSR |= 1 | 1<<1 | 1<<2 | 1<<3 | 1<<6 | 1<<7 | 1<<8 | 1<<9;
    //下降沿触发
    EXTI->RTSR |= 1 | 1<<1 | 1<<2 | 1<<3 | 1<<6 | 1<<7 | 1<<8 | 1<<9;
    //上升沿触发
}
```

红外接收管的数据输出分别连接在单片机GPIO端口 C 的0, 1, 2, 3, 6, 7, 8, 9 引脚, 该函数配置了相应管脚的外部中断, 当收码状态被用户通过串口指令设置为打开后, 如果该路的红外接收管收到红外信号则会产生下降沿然后会被单片机捕获并执行相应的中断处理函数。

下面的源码是各路中断处理函数的定义:

```
//各路的中断处理函数
void EXTI0_IRQHandler(void) {
    IRQ_HANDLE_CORE(0);
    EXTI->PR |= 1<<0;
}

void EXTI1_IRQHandler(void) {
    IRQ_HANDLE_CORE(1);
    EXTI->PR |= 1<<1;
}

void EXTI2_IRQHandler(void) {
    IRQ_HANDLE_CORE(2);
    EXTI->PR |= 1<<2;
}

void EXTI3_IRQHandler(void) {
    IRQ_HANDLE_CORE(3);
    EXTI->PR |= 1<<3;
}

void EXTI9_5_IRQHandler(void) {
    unsigned long origin = EXTI->PR;

    if(origin & EXTI_PR_PR6) {            //如果来源为 6
        IRQ_HANDLE_CORE(4);
        EXTI->PR |= 1<<6;
    } else if(origin & EXTI_PR_PR7) {    //如果来源为 7
        IRQ_HANDLE_CORE(5);
        EXTI->PR |= 1<<7;
    } else if(origin & EXTI_PR_PR8) {    //如果来源为 8
        IRQ_HANDLE_CORE(6);
        EXTI->PR |= 1<<8;
    } else if(origin & EXTI_PR_PR9) {    //如果来源为 9
        IRQ_HANDLE_CORE(7);
        EXTI->PR |= 1<<9;
    }
}
```

可以看出当每一路都调用了一个名为 IRQ_HANDLE_CORE() 的宏, 也可以称作内联函数, 是在头文件 irda.h 中声明的:

```
#define IRQ_HANDLE_CORE(i) do {\
    *g_IrDA_Device[(i)].IrInterrup = 0;\
    irda_decode(&g_IrDA_Device[(i)]);\
    \
    IR_WAVE_FEEDBACK((i));\
    \
    UART_CR();\
} while(0)
```

i为被控制的外设号码

在上面的定义中，该内联函数先关闭被操作外设的外部中断，防止在学码过程中被打断，然后执行 irda_decode 函数来执行学码操作，最后执行 IR_WAVE_FEEDBACK 在终端将解析到的红外波形数据在终端中反馈给用户。

3.4.3 红外外设对象初始化

本次设计中使用面对对象的思维方式，通过C语言的结构体 (struct) 来实现红外外设对象的封装，通过这种方法可以增加代码可读性，提高执行效率。

```
#define IR_DEVICES_NUM            8         //共有8路学码发码外设
#define WAVE_SEGMEENT_NUM        400         //红外波形共有200次电平翻转

typedef struct {
    unsigned short token[WAVE_SEGMEENT_NUM];
    //保存红外波形的数组
    volatile unsigned long *IrInterrup;
    //指针指向中断使能寄存器的某一位（位带操作），用来打开和关闭当前中断
    volatile unsigned long *IrPWM;
    //指针指向 PWM 使能寄存器的某一位（位带操作），用来打开和关闭 PWM
    volatile unsigned long *signal;
    //指针指向 GPIO ODR 的某一位，用来检测输入红外波形的高低
} ir_st, *ir_pst;
ir_st g_IrDA_Device[IR_DEVICES_NUM];
```

这里定义了一个全局变量，用来存储 8 红外外设对象，并在红外发码收码系统初始化函数中对其赋值:

```
//实例化红外外设对象 - 第 1 路
g_IrDA_Device[0].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 0);
g_IrDA_Device[0].IrPWM         = BIT_ADDRP(&(TIM3->CCER), 0);
g_IrDA_Device[0].signal        = BIT_ADDRP(&(GPIOC->IDR), 0);

//实例化红外外设对象 - 第 2 路
g_IrDA_Device[1].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 1);
g_IrDA_Device[1].IrPWM         = BIT_ADDRP(&(TIM3->CCER), 4);
g_IrDA_Device[1].signal        = BIT_ADDRP(&(GPIOC->IDR), 1);

//实例化红外外设对象 - 第 3 路
g_IrDA_Device[2].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 2);
g_IrDA_Device[2].IrPWM         = BIT_ADDRP(&(TIM3->CCER), 8);
g_IrDA_Device[2].signal        = BIT_ADDRP(&(GPIOC->IDR), 2);

//实例化红外外设对象 - 第 4 路
g_IrDA_Device[3].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 3);
g_IrDA_Device[3].IrPWM         = BIT_ADDRP(&(TIM3->CCER), 12);
g_IrDA_Device[3].signal        = BIT_ADDRP(&(GPIOC->IDR), 3);

//实例化红外外设对象 - 第 5 路
g_IrDA_Device[4].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 6);
g_IrDA_Device[4].IrPWM         = BIT_ADDRP(&(TIM4->CCER), 0);
g_IrDA_Device[4].signal        = BIT_ADDRP(&(GPIOC->IDR), 6);
```

```
//实例化红外外设对象 - 第 6 路
g_IrDA_Device[5].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 7);
g_IrDA_Device[5].IrPWM        = BIT_ADDRP(&(TIM4->CCER), 4);
g_IrDA_Device[5].signal        = BIT_ADDRP(&(GPIOC->IDR), 7);

//实例化红外外设对象 - 第 7 路
g_IrDA_Device[6].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 8);
g_IrDA_Device[6].IrPWM        = BIT_ADDRP(&(TIM4->CCER), 8);
g_IrDA_Device[6].signal        = BIT_ADDRP(&(GPIOC->IDR), 8);

//实例化红外外设对象 - 第 8 路
g_IrDA_Device[7].IrInterrup    = BIT_ADDRP(&(EXTI->IMR), 9);
g_IrDA_Device[7].IrPWM        = BIT_ADDRP(&(TIM4->CCER), 12);
g_IrDA_Device[7].signal        = BIT_ADDRP(&(GPIOC->IDR), 9);
```

3.5 学码函数

```
//红外信号学码函数
void irda_decode(ir_pst ir) {
    unsigned char lastStatus = *ir->signal;
    //用来保存上一次电平状态, 以判断电平是否发生翻转
    unsigned short *wave = ir->token;
    //指向用来存储波形长度数据的数组

    for(unsigned int cnt = 0; cnt < WAVE_SEGEMENT_LENGTH; cnt++) {
        //计数器递增, 计数器的数值代表波形长度
        if(lastStatus != *ir->signal) {
            *wave++ = cnt;//当发生电平跳转时保存当前计数并将指针指向波形数组的下一个元素
            if(IR_ISOVERFLOW(wave, ir->token)) {
                //如果遥控器按键未松开会导致多次发送红外信号
                uart_sendStr("\n\r波形数据溢出, 请松开按键!\n\r");
                return;
            }
            cnt = 0;    //清空计数器以便于测量下一段波形长度
        }
        lastStatus = *ir->signal;
        delay_us(26);    // 跳过38KHz载波信号的电平反转
    }
    *wave = 0;              // 数据结束标志
    unsigned short len = wave - ir->token;
    //计算波形的高电平和低电平共有多少段
    uart_sendStr("\n\r波形数组长度: \t");
    uart_short2char(len);//在终端反馈数据
    UART_CR();
}
```

本次设计采用的学码方式是波形拷贝式, 也就是通过记录接收到的红外波形的低电平和高电平的电平宽度, 对红外信号进行完全的复制。

3.6 发码函数

```
//红外信号发码函数
void irda_encode(ir_pst ir) {
    unsigned short *wave = ir->token;//定义一个指针指向被发送的波形数据
    unsigned short cnt;//计数器
    while((cnt = *wave++)) {//提取当前波形长度, 如果长度值有效则将发送波形, 并且将 wave
        *ir->IrPWM ^= 1;    //电平翻转,切换 PWM 输出状态,如果之前输出是打开的则关闭, 如
        while(cnt--)
            delay_us(26);    //计数器单位为 20us, 因为波形有38KHz载波
    }
    *ir->IrPWM = 0;//发码结束后关闭 PWM 输出
}
```

发码函数中通过学码函数记录的数据, 对 PWM 进行打开和关闭来实现载波的发送。由于这里的计数器和学码电路中保存数据的时间单位同为 26 微妙, 所以发送的红外信号和接收到的红外信号几乎是完全相同的, 从而实现了红外信号的学码功能。

4. 参考资料

- Cortex-M3 权威指南
- STM32F10xxx参考手册
- 4.23 STM32 外设篇-红外线接收工作原理及程序设计 此博文包含图片 (2015-06-10 15: 38: 33)
- 浅谈38K红外发射接受编码
- 基于STM32的学习型通用红外遥控设备的设计实现(1~3)
- STM32学习之路-AIRCR寄存器PRIGROUP位的配置
- STM32学习笔记: 外部中断EXTI的使用
- STM32学习笔记之EXTI（外部中断）
- STM32中断优先级彻底讲解