武汉大学计算机学院

本科生实验报告

基于 51 单片机的智能交通灯设计

专 业 名 称 ：计算机科学与技术

课 程 名 称 ：嵌入式系统

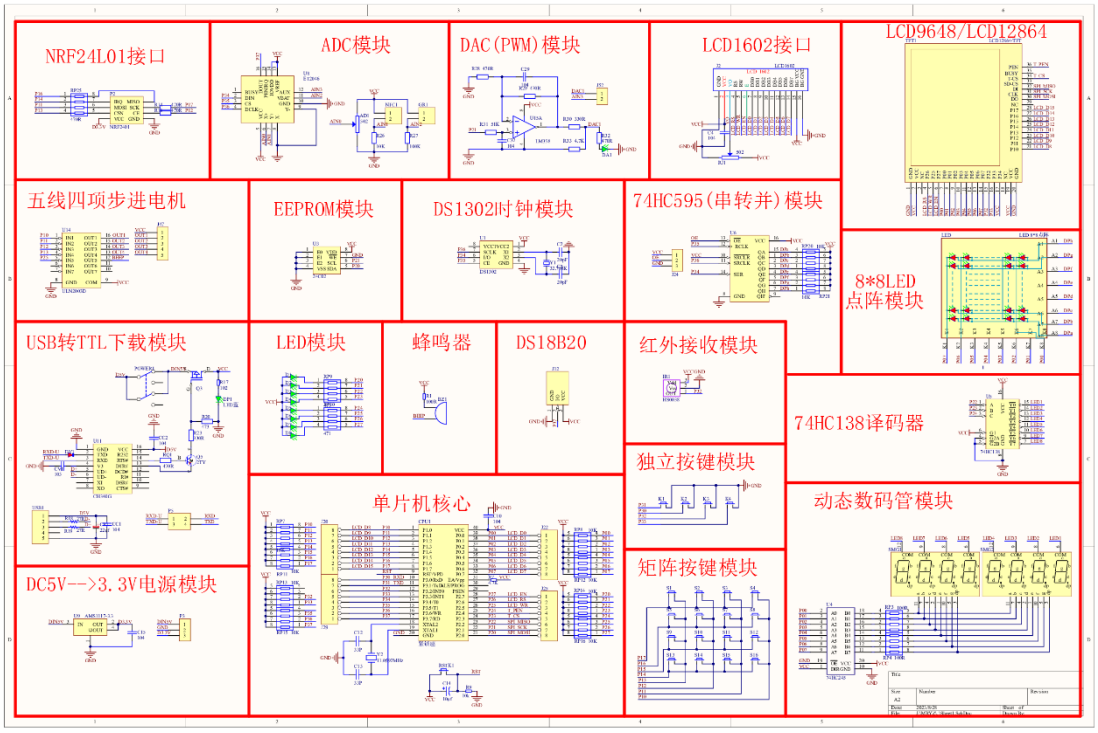
指 导 教 师 ：

学 生 学 号 ：

学 生 姓 名 ：

二〇二四年十月

1 开发板原理图



2 实验器材简介

1. STC89C52RC：一款基于 8051 内核的 8 位单片机，适用于各种嵌入式应用。
2. 11.0592MHz 晶振：提供稳定时钟信号的晶体振荡器，用于微控制器精确计时。
3. LED：发光二极管，用于显示或指示状态。
4. 动态数码管：通过动态扫描技术控制显示的多位数码显示器。
5. 74HC245：用于总线驱动的八路双向总线收发器。
6. 独立按键：单个按钮开关，用于用户输入或触发特定功能。
7. 矩阵键盘：由行列交叉连接的按键阵列，用于多键输入。
8. 蜂鸣器：一种声响设备，通过电信号发出声音用于报警或提示。
9. ULN2003D：高电流达林顿晶体管阵列，用于驱动继电器或步进电机。
10. HS0038 红外接收：用于接收红外遥控信号的接收器模块。
11. DS18B20 温度传感器：一种一线制数字温度传感器，提供高精度的温度测量。
12. 74HC138:一个 3-8 线译码器，将 3 位二进制输入转换为 8 条的独立输出，即动态数码管位选输入。

3 硬件内置定时器 0 实现定时(加分点)

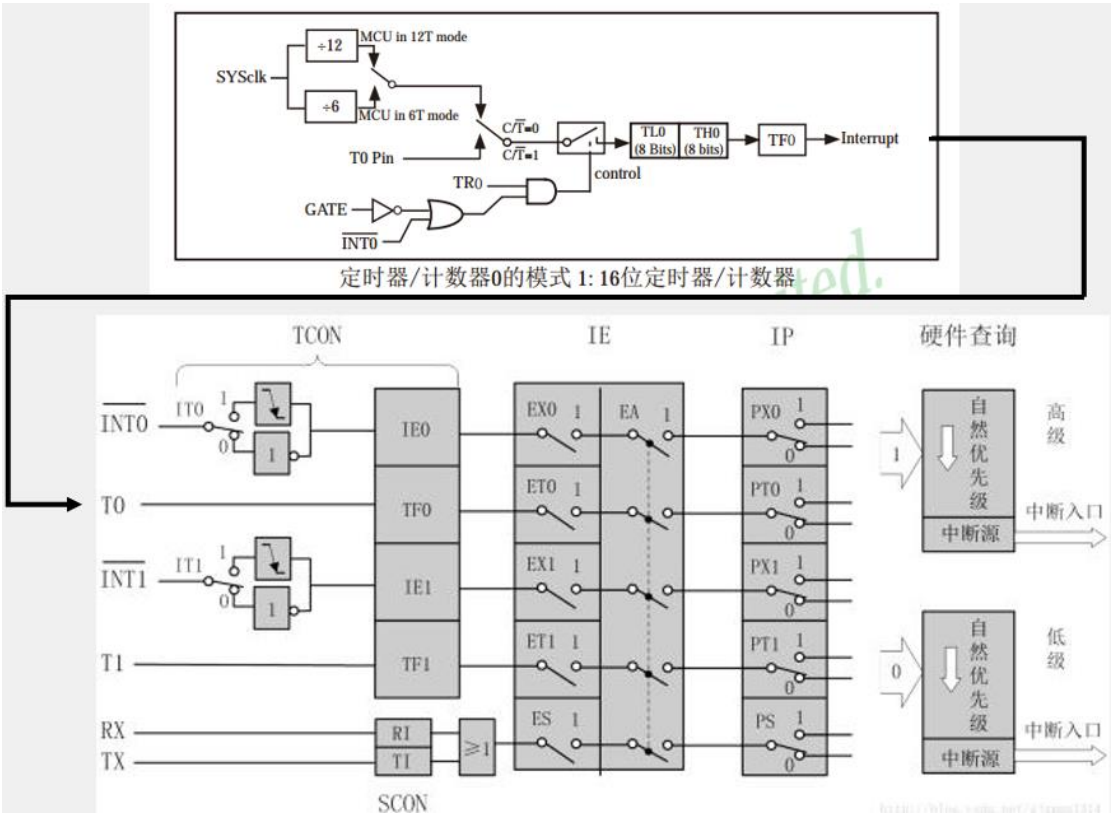
后面许多模块都需要进行定时，这里我们使用定时器 0 进行定时。下面介绍具体逻辑。

3.1 定时器 0 工作模式设定

TMOD	定时器模式寄存器	89H	GATE	C/$\overline{T}$	M1	M0	GATE	C/$\overline{T}$	M1	M0	0000 0000B

首先，STC89C52RC 定时器/计数器 0 有 4 种工作模式：模式 0(13 位定时器/计数器),模式 1(16 位定时器/计数器模式),模式 2(8 位自动重装模式)，模式 3(两个 8 位定时器/计数器)。我们通过设置定时器的 TMOD 寄存器的 M0 位(**M0=1**)，让它以模式 1(16 位定时器/计数器模式)工作。

3.2 定时器 0 定时 500 微秒实现



接下来，我们希望它能够定时 500 微秒。由于我们的晶振是 11.0592MHz，每个机器周期需要 12 个时钟周期（对于标准的 8051 微控制器），从而机器周期 = 1 / (11.0592 MHz / 12) = 1.085069 微秒。因此，为了实现 500 微秒的延迟，我们需要定时器计数约 500 / 1.085069 ≈ 461 个机器周期。8051 定时器是 16 位的，最大计数值为 65536，因此我们需要从 65536 - 461 = 65075 开始

计数。将 65075 转换为十六进制是 0xFE33。因此，我们需要将 **TH0 设置为 0xFE**，将 **TL0 设置为 0x33**，以便计时开始时定时器从 65075 开始计数.接下来我们设置 **TR0=1**，表示启动定时，接着清除定时器 0 的溢出标志位，设置 **TF0=0**。经过 461 个机器周期后溢出，触发中断，TF0 变成 1，实现 500 微秒的定时，同时，我们需要再次将 TH0 设置为 0xFE，将 TL0 设置为 0x33，以便重复计时。

3.3 定时器 0 中断使能

根据流程图，我们知道我们需要设置 **ET0=1** 使能定时器 0 中断，**EA=1** 使能所有中断，同时，将 **PT0=0** 表示低优先级中断。这样，当 500 微秒后，触发中断，就可以执行定时器 0 的中断函数，实现了每 500 微秒精确定时执行指定逻辑。

4 软件延时实现

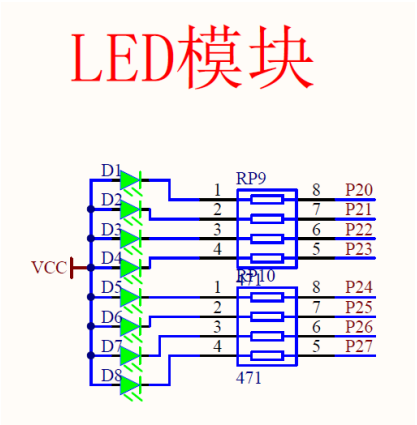
有的时候我们可能希望阻塞主循环，所以这里对软件延时单独介绍。

我写了一个单独的延时函数，如果希望延时 n 毫秒，那就输入 n，这个延时函数会死循环 n 毫秒后再返回。上面硬件定时器说了一个机器周期为 1.085069 微秒，下面这个 while 内部的循环大概有 849 个机器指令，那么就大概延时 1ms，外部循环 n 次，可以近似这个 delay 可以延迟 n 毫秒。

```
void delay(unsigned int n)      //@11.0592MHz
{
    while (n)
    {
        unsigned char i, j;

        _nop_();
        i = 2;
        j = 199;!
        do
        {
            while (--j);
        } while (--i);
        --n;
    }
}
```

5 交通灯基本设计

5.1 LED 交通灯设计



5.1.1 状态设定解释

首先对南北和东西方向的红、绿、黄三种状态进行定义。这里，开发板的 LED 一共是 8 个，分别连接到 STC89C52RC 端口 P2.0-P2.7。由于动态数码管显示占据了端口 P2.2-P2.4 实现位选，所以实际上能用的 LED 灯只有 5 个，同时，这里的 LED 灯只能发红光。基于此，考虑到南北方向，东西方向这两组他们组内的交通灯对应的状态和时间应当是一样的，所以，这里我们 D1,D2/D7,D8 来表示南北/东西方向的交通灯。当 D1/D7 亮时，表示南北/东西方向为绿灯，当 D2/D8 亮时，表示南北/东西方向为绿灯，当 D1,D2/D7,D8 同时亮时，表示南北/东西方向为黄灯。

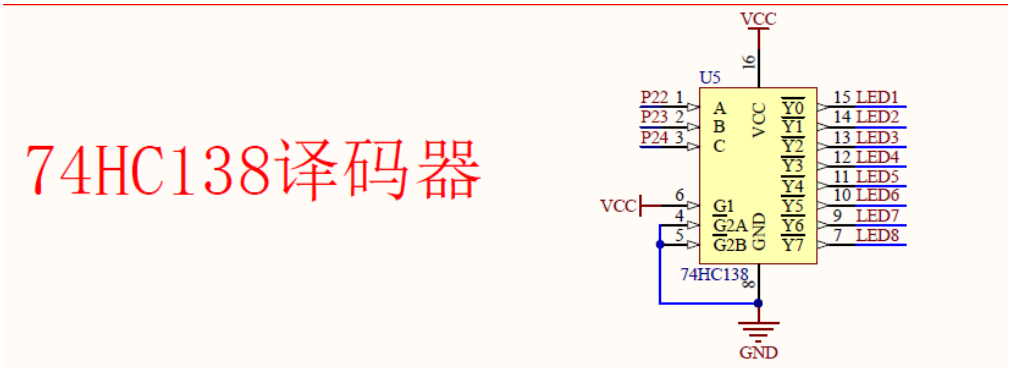
方向	绿灯	红灯	黄灯
南北	D1 亮	D2 亮	D1,D2 同时亮
东西	D7 亮	D8 亮	D7,D8 同时亮

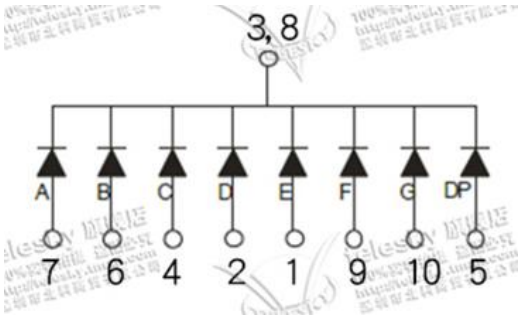
然后，设定红灯默认时长为 16 秒，绿灯默认时长为 13 秒，黄灯默认时长为 3 秒。

5.1.2 实现逻辑

最后，是实际实现方式。通过原理图，我们可以知道该 LED 模块采用的时共阳极结构，所以当对应连接的端口为低电平(即置 0)时，LED 灯导通，发光。初始状态我们设置南北方向为绿灯(P2.0=0)，东西方向为红灯(P2.7=0)。13s 后设置南北为黄灯(P2.1=1)。3s 后设置南北方向为红灯(P2.0=1)，东西方向为绿灯(P2.6=0，P2.7=1)。循环往复，就构成了基本的交通灯红绿灯逻辑。

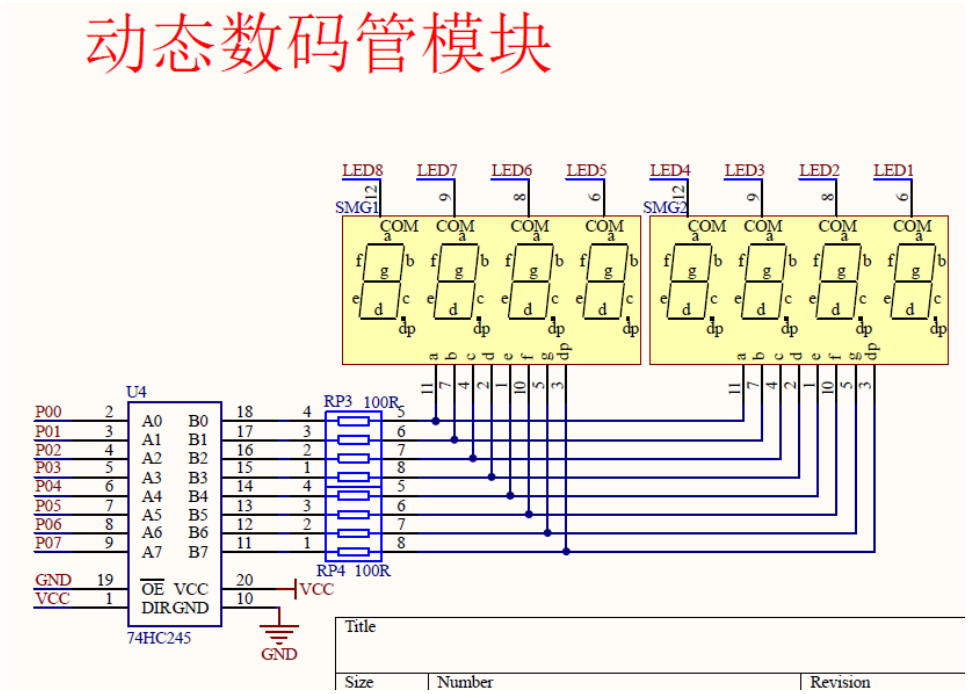
5.2 动态数码管倒计时





5.2.1 位选和段选

首先，介绍一下数码管显示的逻辑，数码管首先通过设定位 P2.2~P2.4，然后通过 74HC138 译码器，输出 8 位信号，进行位选。位选信号接在了二极管的阴极，所以 0 为有效，表示选择那一组 a~g+dp 的 LED 导通。



然后，P0.0 到 P0.7 的信号用于段选，通过 74HC245 芯片提供更高的电流输

出，然后经过保护电阻，分别连接到 a~g 和 dp 这 8 个段。信号接在二极管的阳极，因此信号为 1 时导通二极管，实现发光。根据需要显示的数字，确定要导通的 LED 段，通过编码实现显示。

5.2.2 清零消影动态显示

在倒计时过程中，由于需要同时显示多组数码管（每组表示 a~f 和 dp），我们需要快速切换显示这些数码管。只要切换刷新频率高于人眼的感知能力，就可以让用户觉得数码管是同时显示的。为了避免重影，我们需要在输入下一组数码管的信号之前先清零当前显示。因此，流程为：先输入一组信号，延迟 1 毫秒，然后输入清零信号，再输入下一组信号。这样就能让人眼看起来像是同时点亮了多个数码管。

5.2.3 计时器 0 实现倒计时计时

最后，关于倒计时，我们使用标志变量 T0Count+T0 定时器，来计算 1s。T0 定时器每 500 微秒中断一次，我们在中断函数内自加这个标志变量，当它到 2000 时，就经过了 1s，我们把标志变量清零，就可以再次计算 1s。我们通过这样的方式来计算一秒的时长。每秒结束后，倒计时减少一秒。需要注意的是，当计时减少到 0 时，从绿灯变为黄灯、黄灯变为红灯的默认时长，以及红灯变为绿灯的默认时长。这样我们就实现了对应红绿灯的数码管倒计时效果。

6 交通灯倒计 3 秒时闪烁功能的实现(加分点)

6.1 闪烁频率确定

首先我们确定闪烁的频率，我们认为一次亮->暗->亮为一次闪烁，然后我们在最后 3 秒每一秒就一次闪烁，那么就需要每 500ms 改变动态数码管+LED 的状态。

6.2 闪烁实现

具体状态的改变是这样的：还是使用标志变量 T0Count+T0 定时器，来计算 500ms。T0 定时器每 500 微秒中断一次，我们在中断函数内自加这个标志变量，当它到 1000 或者 2000 时，就经过了 500ms。但是我们在 2000 的时候，才把标志变量清零，就可以再次计算 500ms。然后我们再使用另外一个标志变量 isBlink。它初始状态为 0，每经过 500ms，它就改变一次状态(0->1 或者 1->0)，当他每改变一次状态且倒计时小于 3s 时，LED 红绿灯+倒计时就改变一次亮暗状态，当他为 0 或者倒计时多于 3s 的时候，动态数码管正常显示倒计时，LED 红绿灯恒亮。这样在最后 3s 数码管就实现了闪烁效果。

另外，需要注意的是，只有红灯或绿灯状态最后 3s 会闪烁，黄灯不闪烁。
对应的代码部分：
中断处理+交通灯闪烁部分：

```
// 最后3s每1s闪一次(一次暗亮)，黄灯除外
if (T0Count == 10*100 || T0Count == 20*100) {
    // 南北方向闪烁处理
    if (SN.time <= BLINKTIME && SN.flag != YELLOW) {
        SN.isBlink = !SN.isBlink;
        if (SN.time == 0) SN.isBlink = 0;    // 0s应该让Blink=0，防止后面不显示
        blink(&SN);
    }
    // 东西方向闪烁处理
    if (EW.time <= BLINKTIME && EW.flag != YELLOW) {
        EW.isBlink = !EW.isBlink;
        if (EW.time == 0)    EW.isBlink = 0; // 0s应该让Blink=0，防止后面不显示
        blink(&EW);
    }
}
```

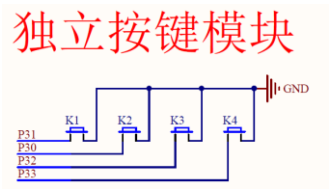
```
// 闪烁
void blink(TrafficLight* self) {
    if (self->flag==RED) {
        P2 ^= 1 << self->redLightPin;
    } else {
        P2 ^= 1 << self->greenLightPin;
    }
}
```

倒计时闪烁部分：

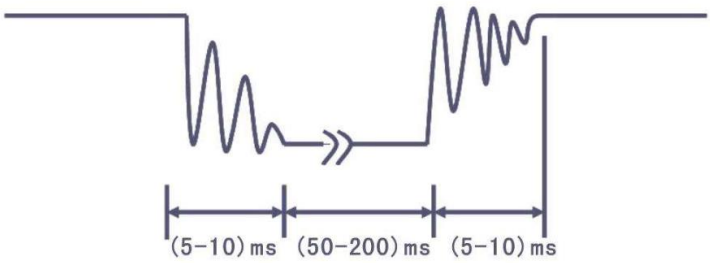
```
/*************************低于3s闪烁，否则正常(黄灯除外)*************************/
if (SN.isBlink) {
    showNum(1,666);
} else {
    showNum(1, SN.time);
}
if (EW.isBlink) {
    showNum(2,666);
} else {
    showNum(2, EW.time);
}
```

7 独立按键让红灯时间暂时延长 3s(加分点)

7.1 独立按键读取



这里我们可以看到，独立按键一边全部接地，另外一边分别接入 P3.0~P3.1，当按下按键时，这几个端口被强下拉置 0，因此，当对应端口为 0 时，可以认为按下了对应的按键。



同时，如上图，对于机械开关，当机械触点断开、闭合时，由于机械触点的弹性作用，一个开关在闭合时不会马上稳定地接通，在断开时也不会一下子断开，所以在开关闭合及断开的瞬间会伴随一连串的抖动。因此，当读到端口为 0 后，我们需要 delay 20ms，然后再看是否端口为 0，才能判断是否按下，同时再 delay 20ms，才能判断是否放开。如果都满足了，我们认为按下且松开了按键。

7.2 紧急状态加 3s 实现

基于这个逻辑，这里我写了函数 Key()，放入 while 主循环中，这个函数返回按下按键的键码，范围是 1~4，无按键按下时返回值为 0。当得到了 1 后，我们认为按下了 K1，发生了紧急状况，然后将南北方向和东西方向的红绿灯时间延长 3 秒。这样就实现了该功能。

```
/*************紧急情况，延长红灯时间3秒(对应另一个路口它的状态也延长三秒)****************/
keyNum = Key();
if (keyNum==1) {
    SN.time += 3;
    EW.time += 3;
}
```

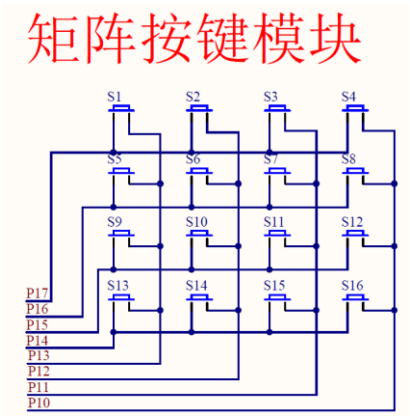
```
unsigned char Key()
{
    unsigned char KeyNumber=0;

    if(P3_1==0){delay(20);while(P3_1==0);delay(20);KeyNumber=1;}
    if(P3_0==0){delay(20);while(P3_0==0);delay(20);KeyNumber=2;}
    if(P3_2==0){delay(20);while(P3_2==0);delay(20);KeyNumber=3;}
    if(P3_3==0){delay(20);while(P3_3==0);delay(20);KeyNumber=4;}

    return KeyNumber;
}
```

8 独立按键+矩阵键盘自定义倒计时时间(加分点)

8.1 设置状态的设定

首先，基于独立按键，我们设置一个标志 isSet，初始为 0，当按下独立按键 K2 后，isSet=1，代表进入设置模式，此时停止倒计时，动态数码管实时显示我们设定的时间。对于时间的设定是使用矩阵键盘实现的。

8.2 矩阵按键的读取



这里矩阵键盘只用了 8 个端口但是需要识别 16 个按键，所以需要使用扫描的方式来实现对按键的读取。同时，因为 P1.0 也用作步进电机的一个端口，所以 P1.0 对应的那一列我们不会扫描，总共读取 12 个按键。

读取按键按下的具体逻辑是这样的，首先让 P1.1~P1.7 全部置 1，接下来分别让 P1.1~P1.3 置 0，假设现在 P1.1 置 0，我们再分别扫描 P1.4~P1.7，如果按键按下，那么线路导通，对应端口会被下拉至 0。比如如果 P1.7 变成了 0，我们就认为现在 S3 被按下。基于这个逻辑我们就可以识别到底是哪个矩阵按键被按下了。需要注意的是，和上面独立按键一样，可能出现机械键盘按下的抖动问题，所以按下和松开的判断也要 delay 20ms。

8.3 矩阵按键的编码

然后是具体的编码，这里，S1-S11(不包括第四列)共 9 个按键，按下后，我让他分别返回 1-9，代表对应数字，然后 S13 按下代表返回 0，S14 按下代表清空目前设定的计时，重新设定，然后 S15 按下表示确定这个计时。

按键	含义
S1	1
S2	2
S3	3

按键	含义
S5	4
S6	5
S7	6

按键	含义
S9	7
S10	8
S11	9

按键	含义
S13	0
S14	清空
S15	确认

8.4 红绿灯状态和时间的设定基本逻辑

接着是红绿灯状态和时间的设定，默认设定的是南北方向红绿灯的设定，东西方向红绿灯时间会自动计算。然后南北方向红绿灯状态是在当前红绿灯状态进行设定的，比如现在南北方向是绿灯，那么我认为就是在设定南北方向绿灯倒计时，然后东西方向就是红灯，时间为设定时间加上黄灯默认时长。现在南北方向是红灯，那么我认为就是在设定南北方向红灯倒计时，然后东西方向

就是绿灯/黄灯，如果设定时长多余默认黄灯时长，那东西方向是绿灯，时间就是设定时间减去黄灯默认时长，否则是黄灯，时间和设定时间一样。南北方向不允许在黄灯状态对时长进行设定，因为我认为黄灯默认时长是在一开始就设定了的，黄灯时长应当是恒为默认时长，而且事实上也基本是从红灯或者绿灯开始倒计时，很少从黄灯开始倒计时。

8.5 红绿灯时间的设定的实现逻辑

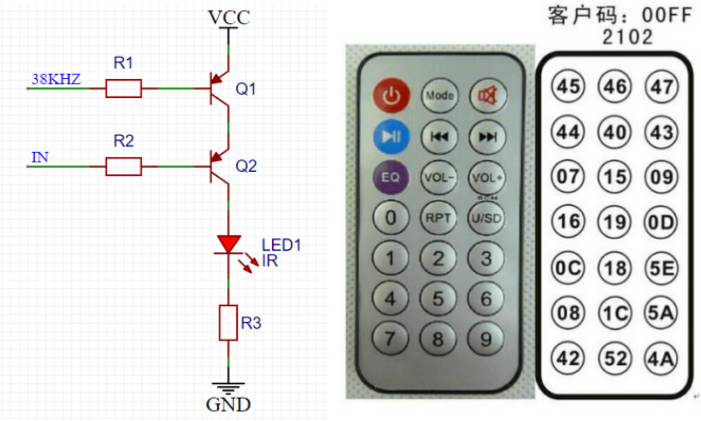
在设定时间的时候，动态数码管实时显示我设定的时间。然后同时我用到了 3 层循环，前两次循环内会一直尝试读取矩阵按键的值，默认读取到矩阵按键的值为常量 INVALID，只要没有读取到上述的 12 个按键的值，那么他的值就是 INVALID，就会死循环。如果读到数字按键，那么就把当前倒计时乘以 10 之后加上这个数字按键的值，然后显示；如果读到清空的话，那就把外层循环变量给清 0，然后时间也清 0，表示从头开始循环三次；如果读到确认按键，那就表示就设定当前这个值。同时第三次循环按键只允许得到清空或者确认按键，保证设定时长在 0-99 之间。

这样我们就实现了独立按键+矩阵键盘自定义倒计时时长。

9 利用红外遥控自定义倒计时时间(加分点)

考虑到可能设定时长的时候我们能够希望远程设定倒计时时长，所以添加了红外遥控设定倒计时时长功能。

9.1 红外发送装置

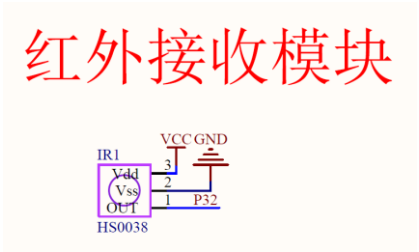


首先介绍一下红外发送装置，在这里，使用的是红外遥控技术，采用了 38kHz 的载波信号进行调制。具体来说，发送装置会在 38kHz 的频率下，将数据编码成高低电平脉冲，这样的脉冲序列通过红外发射管转换为红外光信号。

在发送过程中，首先将待传输的数据经过编码处理，然后将编码后的信号叠加到 38kHz 的载波上。发射器会在每个周期内发出一定时间的高信号（对应

数据的“1”）和低信号（对应数据的“0”），形成调制后的红外信号。这种调制方式确保了接收端能够清晰地识别信号的状态变化，从而正确解读传输的信息。

9.2 红外接受装置

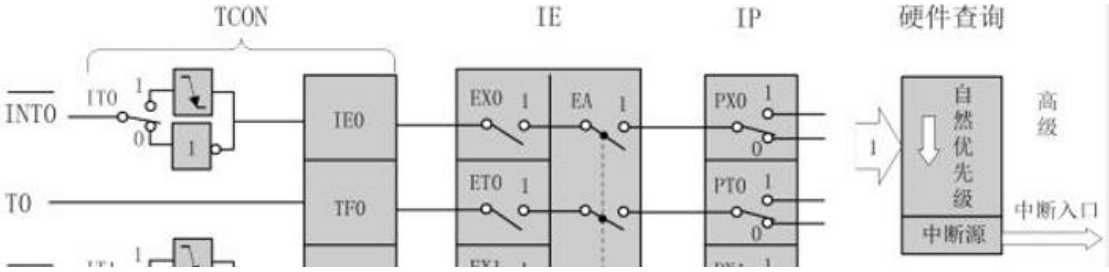


接着介绍一下接受装置，这里使用的是 HS0038，当 HS0038 接收到来自发射器的调制信号时，它首先会利用内部的光电二极管将红外光信号转换为电信号。接着，模块内部的解调电路会自动识别并处理该信号。通过对 38kHz 载波的解调，HS0038 可以提取出原始的数据脉冲序列，这一过程不需要外部干预，因而实现了自动解调的功能。

解调后的信号被转换为标准的高低电平脉冲，以便后续的微控制器或其他处理器进行分析和处理。

9.3 外部中断 0 实现信号读取

这里，对于外部信号的读取时通过外部中断 0 实现的。



具体是实现是这样的，首先，HS0038 的 OUT 端口连接到端口 P3.2，这个端口也恰好是外部中断 0 的输入端口。

这里我们配置，IT0 = 1，表示下降沿触发，因为红外线的编码都是以下降沿为开始的，IE0 = 0，设中断标志位起始为 0，EX0=1，使能外部 0 中断，EA=1，使能所有中断，同时，这个信号应当是优先级最高的中断，随时可以打断其他中断，不然没有办法及时读取到信号变化，所以设置 PX0=1，表示是最高级别的中断。

这样，当下降沿触发中断后，就进入中断处理函数，开始读取识别信号。

9.4 计时器 1 实现精确计时

因为红外信号读取对于时间精度的要求很高，所以需要用计时器 1 实现精

确计时，具体来说，应当是从读取到下降沿之后开始计时，判断多少时间，解码完信号后清零计时器，直到下一次信号来临，再次进行上述流程。所以和计时器 0 的逻辑又有些许不同，因此这里讲解一下逻辑。

具体来说，这里需要有四个函数：

9.4.1 Timer1_Init()

这个函数用于在上电时初始化定时器 1。首先它设置定时器 1 的工作模式，通过修改 TMOD 寄存器实现。TMOD &= 0x0F;和 TMOD |= 0x10;组合使用来确保仅修改定时器 1 的模式，而不影响定时器 0。

接着，它将 TL1 和 TH1 寄存器清零，设置计时初值为 0。

然后，TF1 标志位被清除，确保没有遗留的溢出标志。

TR1 被设为 0，意味着计时器 1 在初始化后是停止的。

最后，通过设置 ET1 和 EA 来使能定时器 1 的中断和总中断，这样当定时器溢出时能够触发中断服务。

9.4.2 Timer1_SetCounter(unsigned int Value):

这个函数用于设置定时器 1 的计数起始值。它接受一个无符号整型参数 Value，将其高 8 位赋给 TH1，低 8 位赋给 TL1。

具体实现上，TH1 = Value/256;和 TL1 = Value%256;分别用于提取和设置高低位。

9.4.3 Timer1_GetCounter():

该函数用于获取定时器 1 的当前计数值。它通过将 TH1 的值左移 8 位并与 TL1 进行按位或操作，合并高低位得到完整的计数值。

返回值是一个无符号整数，表示当前计时器的计数。

9.4.4 Timer1_Run(unsigned char Flag):

这个函数用于控制定时器 1 的启动和停止。它接受一个无符号字符作为参数，其中 1 表示启动定时器，0 表示停止定时器。

实现上，通过直接将 Flag 赋值给 TR1 来控制定时器状态。

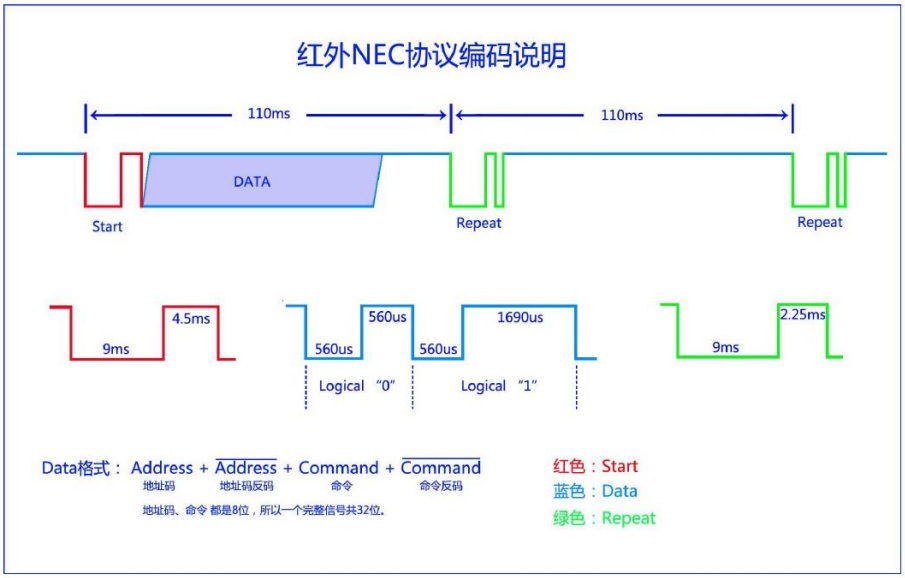
9.4.5 具体实现逻辑

具体说来，首先我们需要 Timer1_Init()来对计时器 1 初始化，然后外部中断下降沿触发后，我们需要 Timer1_SetCounter 计时器初始值为 0，然后调用 Timer1_Run 启动计时器 1，然后直到下一次下降沿来临，调用 Timer1_GetCounter()读取 Counter 计数，通过计时器 0 里面介绍的频率逻辑计算时间经过了多少，从而得到这个信号的具体时间，然后清零后，判断是否有必

要关闭计时器。重复上述过程，就实现了对每个信号具体时长的读取。

这里可以一直使用下降沿来开始以及结束前提的逻辑是我们使用了 NEC 协议，这个协议每个信号往往以下降沿开始，以下降沿结束，和我们外部中断 0 相契合，和我们上述的计时器 1 中断处理也就可以配合了。所以，下面详细介绍这个 NEC 协议以及对应的解码逻辑。

9.5 NEC 协议以及对应解码实现



这里，通信使用的编码是红外 NEC 编码，主要的信号有 start 信号，逻辑 0，逻辑 1 和 repeat 信号，信号的具体编码以及格式查看上图，这里不再赘述。详细介绍我的解码的实现。

这里解码的实现我用到了状态自动机来实现

9.5.1 状态自动机全局变量定义

IR_Time 用于记录时间，IR_State 用于记录当前解码的状态，IR_Data 数组用于存储接收到的红外数据，IR_Address 表示红外线信号发出的地址(也就是遥控器的地址)，IR_Command 表示我们的指令，IR_pData 表示当前处理到了数据的第几位，IR_DataFlag 表示当前是否有有效数据，IR_RepeatFlag 表示当前信号是否是重复输入。

9.5.2 状态自动机状态转换逻辑

这里解释起来比较麻烦，所以对于每个状态机进行解释，同时指出转换逻辑，然后计时器计时得到的数字和时间对应的关系需要用到的频率、机器周期和时钟周期转换关系在计时器 0 模块里面讲解过了，这里不赘述，需要注意的是，我这里前后对于算出来的计时器数字都前后加减了 500，这样就能把误差考虑进去。

具体说来：

状态 0：初始状态
功能：在初始状态下准备开始计时，用于捕捉起始信号。
处理：
调用 Timer1_SetCounter(0)将定时器计数器清零。
调用 Timer1_Run(1)启动定时器 1。
将 IR_State 设置为 1，以便进入下一个状态进行起始信号的检测。

状态 1：等待起始信号或重复信号
功能：检测红外信号以识别是起始信号还是重复信号。
处理：
获取到上一次中断到此次中断的时间间隔 IR_Time。
将计数器清零准备下次计时。
判断 IR_Time 以决定是否接收到起始信号或重复信号：
如果 IR_Time 在范围 12442±500 内(对应时间 13.5ms)，表示接收到起始信号，IR_State 被置为 2，进入数据接收状态。
如果 IR_Time 在范围 10368±500 内(对应时间 11.25ms)，表示接收到重复信号，置 IR_RepeatFlag 为 1，停止定时器，将 IR_State 重置为 0。
如果时间不在以上范围内，表示接收出错，IR_State 保持为 1，继续等待下一个信号。

状态 2：接收数据
功能：解析并获取红外信号的数据位。
处理：
获取当前中断到上次中断的时间间隔 IR_Time。
将计数器清零准备下次计时。
根据 IR_Time 判断当前接收到的数据位是 0 还是 1：
如果 IR_Time 在范围 1032±500(对应时间 1120us)内，表示接收到数据位 0，将对应位置的数据位清 0，并递增数据位置指针 IR_pData。
如果 IR_Time 在范围 2074±500(对应时间 2250us)内，表示接收到数据位 1，将对应位置的数据位置 1，并递增数据位置指针 IR_pData。
如果时间不在以上范围内，表示接收出错，将 IR_pData 清零，IR_State 重置为 1，准备重新接收。
当接收到 32 位数据时，清零数据位置指针 IR_pData 并验证数据：
检查数据完整性，如果数据有效，将 IR_Data 中的地址和命令转存到 IR_Address 和 IR_Command 中，并置 IR_DataFlag 为 1 表示数据接收完成。
停止定时器，将状态 IR_State 重置为 0，准备接收新的信号。

通过 3 个状态的转换，我们就能对红外信号进行解码和解读。

9.5.2 红绿灯状态和时间设定的实现

红绿灯状态和时间设定的实现的基本逻辑思路和实际实现思路和矩阵键盘是类似的，也是根据按键按下后得到解码结果，将其返回到主循环中，然后在主循环中进行各种逻辑操作。与矩阵键盘不同处只有编码，所以这里对于编码

进行补充说明。

这里以表的形式给出：

按键	命令	结果
Mode	0x46	进入设置模式
Mute	0x47	清零
EQ	0x07	确认

按键	命令	结果
VOL-	0x15	倒计时加 1
VOL+	0x09	倒计时减 1
0	0x16	0

按键	命令	结果
1	0x0c	1
2	0x18	2
3	0x5E	3

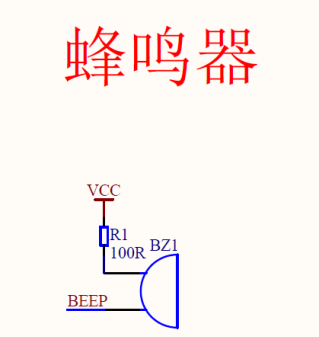
按键	命令	结果
4	0x08	4
5	0x1C	5
6	0x5A	6

按键	命令	结果
7	0x42	7
8	0x52	8
9	0x4A	9

需要注意的是，由于这里按键充足，所以我补充了按钮 VOL+和 VOL-，实现对倒计时的加 1 减 1 操作。

11 为行人增加语音提示(加分点)

当绿灯倒计时只剩 6 秒时，我们需要提醒行人快速通过，这里介绍实现逻辑。这里是通过蜂鸣器鸣叫实现语音提醒的，我们这里使用的蜂鸣器是无源蜂鸣器，它的内部不带振荡源，需要控制器提供振荡脉冲才可发声，调整提供振荡脉冲的频率，可发出不同频率的声音。



首先，是蜂鸣器的频率设计，经过查阅，我们知道常用的蜂鸣器的频率是 1KHz，换算成周期就是 1ms，也就是说，只要每 500 微秒转换一次蜂鸣器的 BEEP 口状态，就可以实现蜂鸣器以 1KHz 发出声音提醒行人通过了，蜂鸣器这里的 BEEP 口接的是 P2.5 端口。

而我们计时器 0 基本的中断触发的时间就是 500 微秒。所以，我们可以设置一个标志变量 buzzerFlag，在计时器 0 的中断处理函数中，每 500 微秒将其置 1，然后当绿灯时间小于 6 秒且 buzzerFlag 为 1 的时候，主函数开始调用函数，被调用的函数将 P2.5 端口状态取反就可以了，同时在这个函数中，最后也应当置 buzzerFlag=0。这样，绿灯最后 6 秒，每 500 微秒蜂鸣器 BEEP 口状态就会发生改变，蜂鸣器就会以 1KHz 震动，发出声音，提醒行人通过。

12 温度传感器+风扇实现自动降温(加分点)

考虑到夏天温度很高，MCU 可能工作负载大，温度过高，可能会烧毁电路，所以这里添加了一个温度传感器识别温度，当超过设定阈值的时候，风扇就会自动开始转动，直到温度低于阈值，风扇停止转到，下面介绍具体实现逻辑。

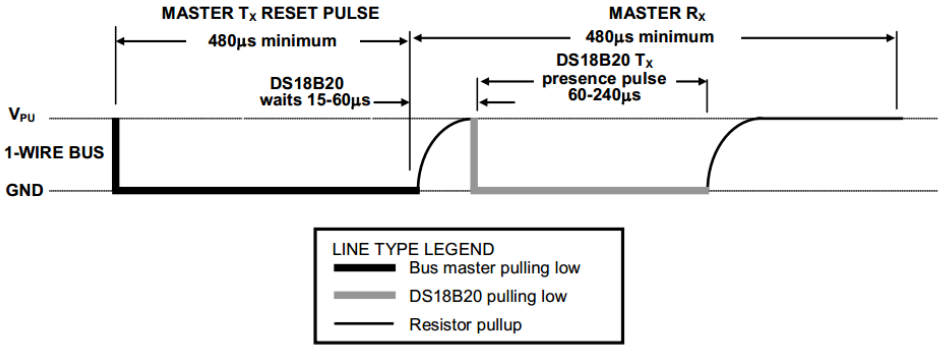
12.1 温度传感器的单总线实现

这里，由于我们使用的温度传感器是 DS18B20，它的通信接口是 1-Wire（单总线），所以我们需要单独对单总线的信号进行解码。

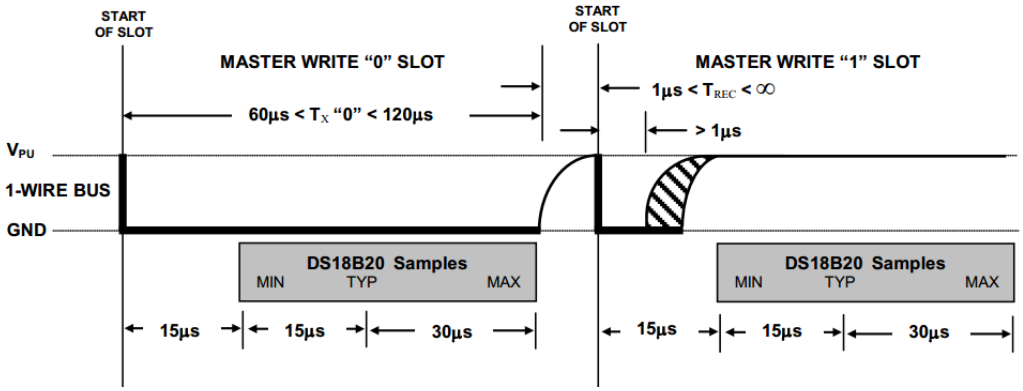
单总线（1-Wire BUS）是由 Dallas 公司开发的一种通用数据总线，单总线只需要一根通信线 DQ 即可实现数据的双向传输，通信模式是异步、半双工的。

12.1.1 单总线时序结构

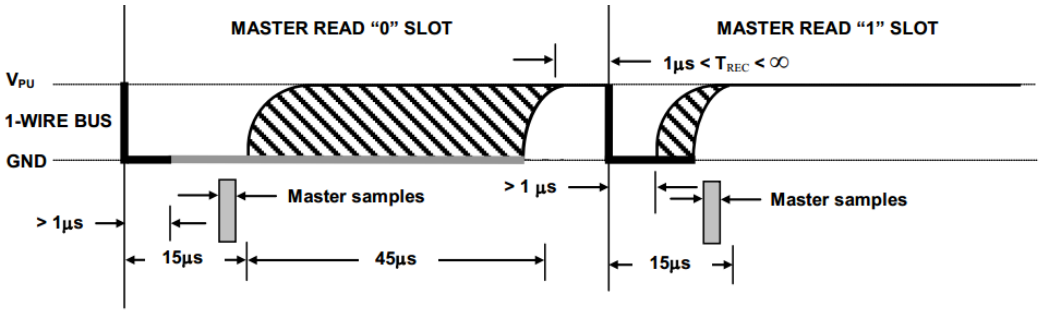
下面介绍单总线的时序结构



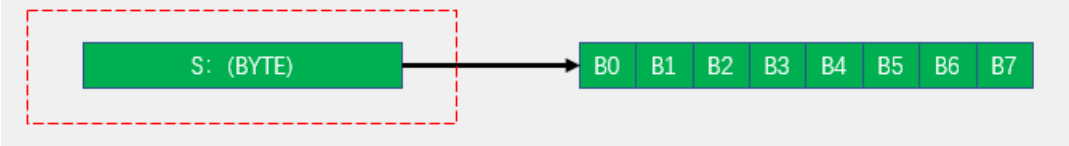
初始化：主机将总线拉低至少 480us，然后释放总线，等待 15~60us 后，存在的从机会拉低总线 60~240us 以响应主机，之后从机将释放总线



发送一位：主机将总线拉低 60~120us，然后释放总线，表示发送 0；主机将总线拉低 1~15us，然后释放总线，表示发送 1。从机将在总线拉低 30us 后（典型值）读取电平，整个时间片应大于 60us。



接收一位：主机将总线拉低 1~15us，然后释放总线，并在拉低后 15us 内读取总线电平（尽量贴近 15us 的末尾），读取为低电平则为接收 0，读取为高电平则为接收 1 ，整个时间片应大于 60us。



发送一个字节：连续调用 8 次发送一位的时序，依次发送一个字节的 8 位（低位在前）。

接收一个字节：连续调用 8 次接收一位的时序，依次接收一个字节的 8 位（低位在前）。

基于上述逻辑，我们给出单总线信号的编码和解码实现逻辑。

12.1.2 单总线信号的编码和解码实现

这里根据不同的时序分别介绍对应函数的实现。

OneWire_Init()

功能：初始化单总线通信并检查从设备的响应。

实现：

将 OneWire_DQ 引脚设置为高电平然后拉低，产生一个复位脉冲。

延时 500 微秒（通过 i = 247; while (--i);实现），保证从设备能检测到复位。

将引脚拉高，并延时 70 微秒，等待从设备的响应。

读取 OneWire_DQ 的状态作为从设备的响应位：如果为 0 表示响应正常，否则未响应。

再延时 500 微秒，确保复位周期结束。

返回响应位 AckBit。

OneWire_SendBit(unsigned char Bit)

功能：发送一个位到单总线设备。

实现：

将 OneWire_DQ 拉低，开始写时隙。
延时 10 微秒，确保从设备进入接收状态。
根据 Bit 值决定 OneWire_DQ 的状态：为 0 保持低电平，为 1 拉高。
延时 50 微秒，保持时隙稳定。
最后将引脚拉高，结束时隙。

OneWire_ReceiveBit()
功能：从单总线设备接收一个位。
实现：
将 OneWire_DQ 拉低，开始读时隙。
延时 5 微秒，然后拉高以释放总线。
再延时 5 微秒后，立即读取 OneWire_DQ 的状态以获取数据位。
延时 50 微秒，等待时隙结束。
返回读取到的位 Bit。

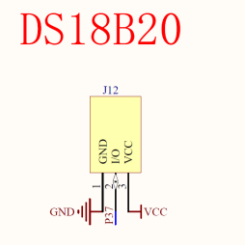
OneWire_SendByte(unsigned char Byte)
功能：发送一个字节到单总线设备。
实现：
通过循环，逐位调用 OneWire_SendBit()发送字节的每一位。
使用位与操作 Byte & (0x01 << i)逐位提取字节中的每一位。

OneWire_ReceiveByte()
功能：从单总线设备接收一个字节。
实现：
初始化字节 Byte 为 0。
通过循环，逐位调用 OneWire_ReceiveBit()接收数据。
使用位或操作 Byte |= (0x01 << i)将接收到的每一位整合到字节的相应位置。
返回完整的字节 Byte。

这样就分别实现了单总线的几个时序结构，实现了对单总线信号的编码和解码。

12.2 温度传感器的温度读取实现

12.2.1 温度传感器读取的基本逻辑

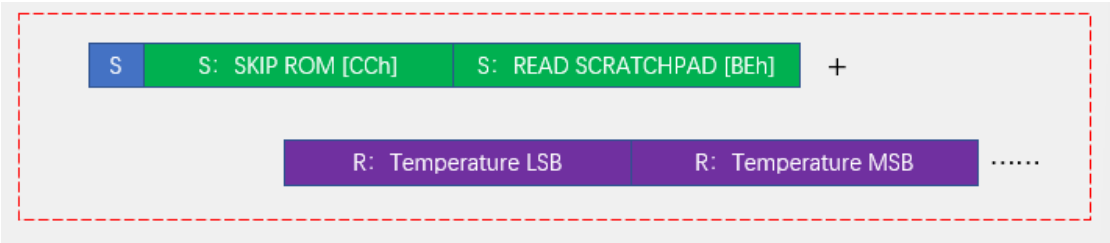


ROM指令	功能指令
SEARCH ROM [F0h]	CONVERT T [44h]
READ ROM [33h]	WRITE SCRATCHPAD [4Eh]
MATCH ROM [55h]	READ SCRATCHPAD [BEh]
SKIP ROM [CCh]	COPY SCRATCHPAD [48h]
ALARM SEARCH [ECh]	RECALL E2 [B8h]
	READ POWER SUPPLY [B4h]

我们温度传感器使用的组件是 DS18B20，他主要是有三个操作，第一个是初始化，实现从机复位，主机判断从机是否响应。第二个是 ROM 操作，实现 ROM 指令+本指令需要的读写操作。最后是功能操作，实现功能指令+本指令需要的读写操作。

这里我们只需要读取温度并将结果返回，所以我们只需要实现两个功能：

第一个是温度变换：初始化→跳过 ROM →开始温度变换



第二个是温度读取：初始化→跳过 ROM →读暂存器→连续的读操作

所以，接下来给出两个操作的具体实现逻辑。

12.2.2 温度传感器读取的具体实现逻辑

DS18B20_ConvertT()

功能：启动 DS18B20 的温度转换。

实现：

调用 OneWire_Init()初始化单总线，准备开始与传感器通信。

发送 DS18B20_SKIP_ROM 命令，用于跳过 ROM 选择，直接对单一设备操作，这种情况下适合单个传感器。

发送 DS18B20_CONVERT_T 命令，启动温度转换过程。DS18B20 在接收到该命令后开始进行温度测量。

DS18B20_ReadT()

功能：读取 DS18B20 的温度数据。

实现：

调用 OneWire_Init()初始化单总线。

发送 DS18B20_SKIP_ROM 命令，跳过 ROM 步骤，直接与传感器交互。

发送 DS18B20_READ_SCRATCHPAD 命令，通知传感器准备发送温度数

据。

通过 OneWire_ReceiveByte()接收两个字节的数据，分别是温度的低字节（TLSB）和高字节（TMSB）。

将两个字节组合成一个 16 位整数 Temp，使用 Temp = (TMSB << 8) | TLSB 进行位运算。

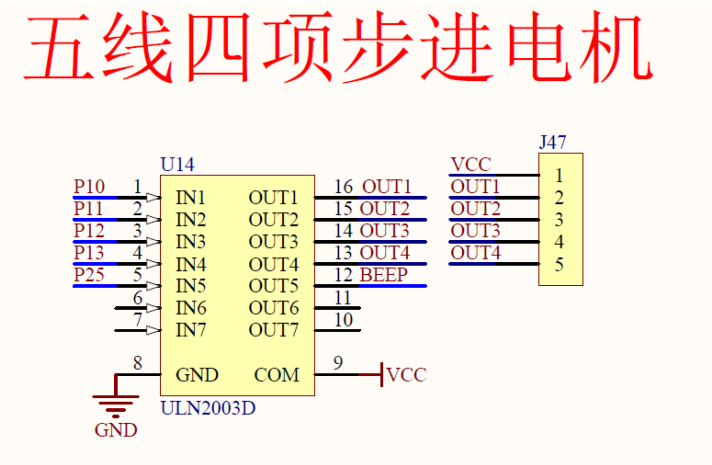
计算实际温度 T，使用公式 T = Temp / 16.0，因为 DS18B20 的温度分辨率为 0.0625° C，即传感器数据的低 4 位为小数部分。

返回计算的温度 T。

需要注意的是，经过我的实际测试，在主循环中，如果调用完 DS18B20_ConvertT()后直接调用 DS18B20_ReadT()，读取出来的温度起始的时候有点问题，经过查阅，可能是温度转换处理需要一些时间，因此需要延时 750ms 后再进行读取。这里我们不可能使用软件延时，不然 750ms 明显会影响整个红绿灯系统。所以还是使用计时器实现操作，具体说来，我在计时器 0 中，设置标志变量 waitForGetT，它每次中断加 1，当他为 1500 时，时间就经过了 750ms，然后我们调用 DS18B20_ReadT()，并且将结果赋给 t，然后清零标志变量 waitForGetT，后面我们实际读取温度的时候，是使用 t 的值来认为这就是当前温度。

这样，我们就实现了读取当前温度的操作。

12.3 温度过高启动风扇的实现逻辑



首先，我们使用的风扇是一个简单的马达+风扇的结构。因为本身端口提供的电流不足以驱动风扇转动，所以这里需要使用 ULN2003D 放大电流信号，提供足够的电压，变成步进电机来驱动风扇转动。

具体我们主要是将风扇一端连接这里的 VCC，另一端连接 OUT1，这样，我们只需要将 P1.0 置 1，那么他就会在 OUT1 放大电流输出提供足够的电压给风扇，风扇就会开始转动。这就是实现风扇转动的逻辑。

最后，在主循环中，我们首先需要读取 t 的值，看当前温度，如果温度大于阈值，那么我们置 P1.0=1，那么 ULN2003D 步进电机驱动风扇转动，直到低于阈值，风扇停止转动。这就是温度过高实现风扇自动转动降温的逻辑。