# 目录


- DS18B20数字温度计 (一) 电气特性, 寄生供电模式和远距离接线
- DS18B20数字温度计 (二) 测温, ROM和CRC算法
- DS18B20数字温度计 (三) 1-WIRE总线 ROM搜索算法和实际测试


## DS18B20 搜索算法

以下说明当总线上存在多个 DS18B20 芯片时, 识别各个 DS18B20 的编号并进行通信的算法.

其实这是 1-Wire 总线的搜索算法, 当 1-Wire 总线上挂接了多个设备时, 总线控制端需要通过 ROM Search 命令来判断总线上存在的设备以及获取他们的8字节唯一ROM.

## 1-WIRE SEARCH ALGORITHM 算法规则和实现机制

ROM搜索算法的核心规则, 是在搜索中重复进行一个简单的三步操作

### 步骤1: 读一次: 得到一位的值

总控读取1个bit. 这时每个设备都会将ROM当前这一位的bit值放到总线上, 如果这位是0, 就会对总线写**0**(拉低总线), 如果这位是1, 则会对总线写**1**, 允许总线保持高电平. 如果两者都存在, 总控读取的是**0**(低电平).

### 步骤2: 再读一次: 得到这位的补码

总控继续读一个bit, 这时候每个设备会将ROM当前这一位的bit的补码放到总线上, 如果这位是0就会写**1**, 如果这位是1则会写**0**, 如果两者都存在, 总控会读到一个**0**, 这样总控就会知道存在多个设备, 并且它们的ROM在这一位上的值不同.

### 步骤3: 写一次: 指定这一位的目标值

总控写入一个bit, 比如写入**0**, 表示在后面的搜索中选择这一位为**0**的设备, 屏蔽掉这一位为**1**的设备

## 循环

总线控制端在8字节ROM的每一位上执行这个三步操作后, 就能知道一个 DS18B20 的 8字节 ROM 值, 如果总线上有多个 DS18B20, 则需要重复多次.

## 搜索示例

### 示例数据

下面的例子假设总线上有4个设备, 对应的ROM值分别为

- ROM1 00110101...
- ROM2 10101010...
- ROM3 11110101...
- ROM4 00010001...

### 示例搜索过程

搜索步骤如下

1. 单线总线控制端(以下简称总控)执行 RESET, 所有的 DS18B20设备(以下简称设备)响应这个RESET
2. 总控执行 Search ROM 命令
3. 总控读取1个bit. 这时每个设备都会将自己的ROM的第一个bit放到总线上, ROM1 和 ROM4 会对总线写**0**(拉低总线), 而 ROM2 和 ROM3 则会对总线写**1**, 允许总线保持高电平. 这时候总控读取的是**0**(低电平).
4. 总控继续读下一个bit, 每个设备会将第一个bit的补码放到总线上, 这时候 ROM1 和 ROM4 写**1**, 而 ROM2 和 ROM3 写**0**, 因此总控依然读到一个**0**, 这时候总控会知道存在多个设备, 并且它们的ROM在这一位上的值不同.
5. (说明)从每次的两步读取中观察到的值分别有以下的含义
   - *00* 有多个设备, 且在这一位上值不同
   - *01* 所有设备的 ROM在这一位上的值是**0**
   - *10* 所有设备的 ROM在这一位上的值是**1**
   - *11* 总线上没有设备
6. 总控写入一个bit, 比如写入**0**, 表示在后面的搜索中屏蔽 ROM2 和 ROM3, 仅留下 ROM1 和 ROM4
7. 总控再执行两次读操作, 读到的值为**0**,**1**, 这表示总线上所有设备在这一位上的值都是**0**
8. 总控写入一个bit, 因为值是确定的, 这次写入的是**0**
9. 总控再执行两次读操作, 读到的值为**0**,**0**, 这表示总线上还有多个设备, 在这一位上的值不同
10. 总控写入一个bit, 这次写入**0**, 这将屏蔽 ROM1, 仅留下 ROM4
11. 总控重复进行三步操作, 读出 ROM4 剩余的位, 完成第一次搜索
12. 总控再次重复之前的搜索直到第7位
13. 总控写入一个bit, 这次写入**1**, 将屏蔽 ROM4, 仅保留 ROM1

14. 总控通过重复三步操作, 读出 ROM1 剩余的位
15. 总控再次重复之前的搜索直到第3位
16. 总控写入一个bit, 这次写入**1**, 将屏蔽 ROM1 和 ROM4 仅保留 ROM2 和 ROM3
17. 重复之前的逻辑, 当所有**00**读数都被处理, 说明设备的ROM已经全部被读取.

总控通过单线总线读取所有设备, 每个设备需要的时间为 `960 µs + (8 + 3 x 64) 61 µs = 13.16 ms` , 识别速度为每秒钟75个设备.

# 代码逻辑

使用代码实现时, 整体的逻辑是按一个固定的方向(先0后1)深度优先遍历一个二叉树.

### 数据结构

- 预设一个8字节数组 Buff 用于记录路径(即ROM的读数)
- 预设一个8字节数组 Stack, 用于记录每一位的值是否确定, 如果确定就是1, 未确定就是0.
- 预设一个整数变量 Split_Point 用于记录每一轮搜索中得到的最深分叉点的位置, 下一次到这一位就用1进行分叉.

### 遍历逻辑

在每一轮遍历中

1. 从低位开始, 每一位进行两次读, 得到这一位的值和补码
2. 对前面的结果进行判断
    1. 如果为**11**, 说明没有设备, 直接退出
    2. 如果为**01**, 说明这一位都是**0**, 写入 Buff, 同时将 Stack 这一位设成 1, 表示这一位已确认
    3. 如果为**10**, 说明这一位都是**1**, 写入 Buff, 同时将 Stack 这一位设成 1, 表示这一位已确认
    4. 如果为**00**, 说明这一位产生了分叉, 需要继续判断
3. 对分叉的判断, 与 Split_Point 记录的值进行比较
    1. 如果当前位置比已知的分叉点更浅, 说明还没到该分叉的位置, 继续设置成 Buff 中上一次使用的值, Stack不变
    2. 如果当前位置等于分叉点, 说明已经到了上次定好的分叉位置, 上次已经用0分叉过了, 这次就用1进行分叉, 这一位就确认了, 将 Stack 这一位设成 1, 表示已确认
    3. 如果当前位置比已知的分叉点位置还要深, 说明发现了新的分叉点(例如用1分叉后, 进入了新的子树, 发现下面还有分叉), 更新 Split_Point 记录分叉点位置, 将 Stack 这一位设成 0 (未确认), 用默认的0继续往下走
4. 在这轮遍历结束后, Buff 就得到了一个新的地址
5. 检查 Split_Point 是否需要往上挪: 在 Stack 上找到 Split_Point 标识的位置, 如果值为1, 则将 Split_Point 设置到最浅的一个0的位置. (例如这次正好在分叉点使用1分叉, 当前点确认了, 而之后又全是确认的情况, 需要将分叉点往上移)
6. 结束条件: 和深度遍历一样, 每一轮遍历后分叉点可能会上下变化, 当分叉点的位置为0时, 说明遍历结束

# 代码实现

搜索逻辑的C语言代码实现

```c
/**
 * buff, stack 和 split_point 都是全局变量, 由外部传入
 *
 */
uint8_t DS18B20_Search(uint8_t *buff, uint8_t *stack, uint8_t split_point)
{
    uint8_t len = 64, pos = 0;
    /* 分叉点的初始值应该用0xFF, 如果输入参数为0, 将其设为0xFF */
    split_point = (split_point == 0x00)? 0xFF : split_point;
    /* Reset line */
    DS18B20_Reset();
    /* Start searching */
    DS18B20_WriteByte(ONEWIRE_CMD_SEARCHROM);

    // len 初始值为64, 对 8 字节 ROM 做一个遍历
    while (len--)
    {
        // 两次读, 读取这一位bit值和补码
        __BIT pb = DS18B20_ReadBit();
        __BIT cb = DS18B20_ReadBit();
        if (pb && cb) // 都是1, 表示没有设备
        {
            return 0;
        }
        else if (pb) // pb=1, cb=0, 说明这一位为1
        {
```

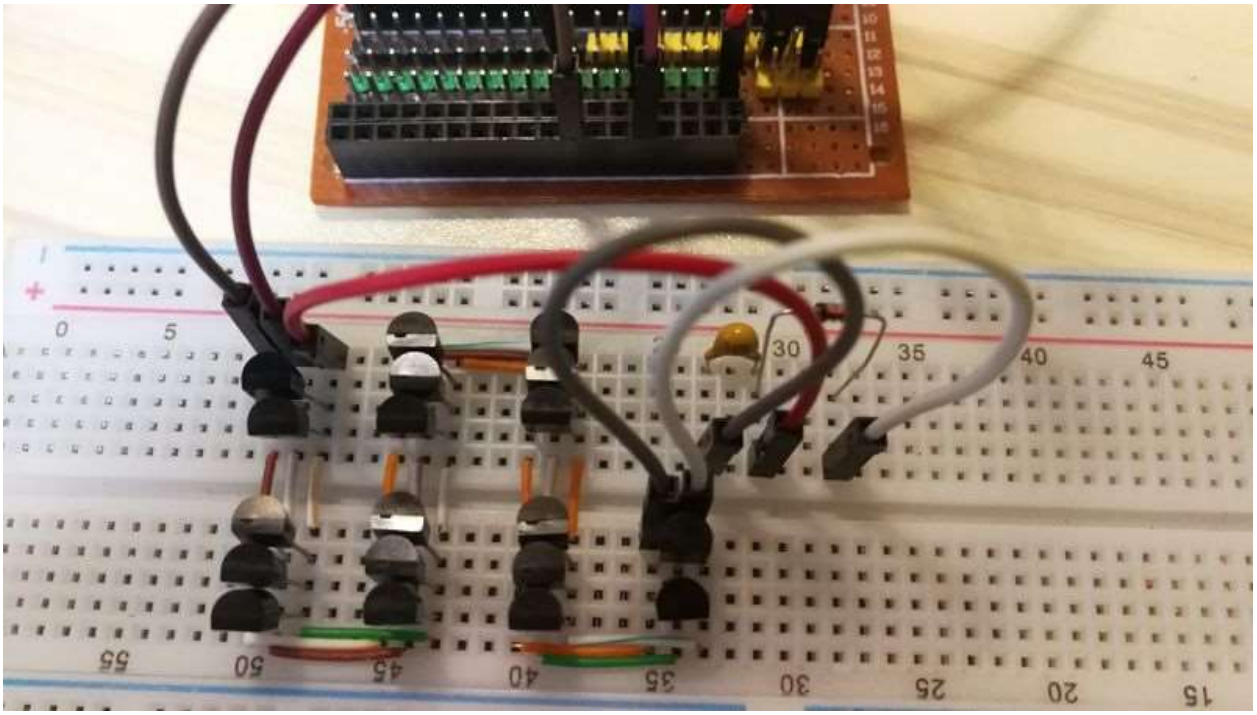
```
            // 在buff上记录这一位
            *(buff + pos / 8) |= 0x01 << (pos % 8);
            DS18B20_WriteBit(SET);
            // 在stack上将这一位记录为1，表示已确认
            *(stack + pos / 8) |= 0x01 << (pos % 8);
        }
        else if (cb) // pb=0, cb=1, 说明这一位为0
        {
            // 在buff上记录这一位
            *(buff + pos / 8) &= ~(0x01 << (pos % 8));
            DS18B20_WriteBit(RESET);
            // 在stack上将这一位记录为1，表示已确认
            *(stack + pos / 8) |= 0x01 << (pos % 8);
        }
        else // 出现分叉点
        {
            if (split_point == 0xFF || pos > split_point)
            {
                // 比上次记录的点更深，出现了新的分叉点
                *(buff + pos / 8) &= ~(0x01 << (pos % 8));
                DS18B20_WriteBit(RESET);
                // 在stack上将这一位记录为0，表示未确认
                *(stack + pos / 8) &= ~(0x01 << (pos % 8));
                // 记录新的分叉点位置
                split_point = pos;
            }
            else if (pos == split_point)
            {
                // 到达了上次记录的分叉点位置，这次使用1继续往下走
                *(buff + pos / 8) |= 0x01 << (pos % 8);
                DS18B20_WriteBit(SET);
                // 在stack上将这一位记录为1，表示已确认
                *(stack + pos / 8) |= 0x01 << (pos % 8);
            }
            else
            {
                // 这个分叉点处于中间位置，还没到处理时间，继续使用上次记录的值
                DS18B20_WriteBit(*(buff + pos / 8) >> (pos % 8) & 0x01);
            }
        }
        pos++;
    }
    // 重新定位分叉点，将其指向到stack上最后一个未确认的位置
    while (split_point > 0 && *(stack + split_point / 8) >> (split_point % 8) & 0x01 ==
0x01) split_point--;
    return split_point;
}
```

调用方法

```
sp = 0;
do
{
    // ROM search and store ROM bytes to addr
    sp = DS18B20_Detect(addr, Search_Stack, sp);
    // Print the new split point and address
    UART1_TxHex(sp);
    UART1_TxChar(' ');
    PrintArray(addr, 0, 8);
    UART1_TxString("\r\n");
} while (sp);
```

## 运行实测

对一个挂载了19个 DS18B20 的 1-Wire 总线进行实际测试, 用1uF电容和1N4148模拟寄生供电电路, 与上位机只连了两根线.



实际的测试输出如下, 第一列输出的是Split_Point的值, 表示当前的分叉深度, 后半部分是这个DS18B20采样的温度值和CRC

```
0F 2854FD96F0013C1A........B20155057FA5A5669A CRC:9A CR LF
0D 28D44496F0013C4C........BD0155057FA5A56660 CRC:60 CR LF
0B 28744196F0013CC2........B50155057FA5A5664A CRC:4A CR LF
09 280CCB96F0013C8D........B20155057FA5A5669A CRC:9A CR LF
0B 28D2A396F0013C75........B50155057FA5A5664A CRC:4A CR LF
0D 288AFB48F6973CFD.......BE0155057FA581665F CRC:5F CR LF
0C 28AA8196F0013C37........B40155057FA5A56609 CRC:09 CR LF
0A 283A9096F0013C37........B80155057FA5A56636 CRC:36 CR LF
08 283E5996F0013C3A........B80155057FA5A56636 CRC:36 CR LF
0B 2811E896F0013C2A........B70155057FA5816636 CRC:36 CR LF
0C 28C90196F0013C66........B40155057FA5A56609 CRC:09 CR LF
0D 28597196F0013CBA........B80155057FA5A56636 CRC:36 CR LF
0A 28794648F65D3C26........B60155057FA5A5668F CRC:8F CR LF
0B 2865BB96F0013CB5........BD0155057FA5A56660 CRC:60 CR LF
0C 28ADCB96F0013CE6........BA0155057FA581664A CRC:4A CR LF
09 281D1648F64B3CEA.......BD0155057FA5A56660 CRC:60 CR LF
0B 2843E896F0013C6A........BB0155057FA5A566F3 CRC:F3 CR LF
0A 289B0896F0013CD5........B70155057FA5816636 CRC:36 CR LF
00 28EF5C96F0013C1B........BE0155057FA5A566A5 CRC:A5 CR LF
```

# 参考


- 单线总线搜索算法 1-WIRE SEARCH ALGORITHM https://www.maximintegrated.com/en/design/technical-documents/app-notes/1/187.html