uC/OS室温计

移植 µC/OS-II

下载源码

下载 µC/OS-II 源码，要注册账号和邮箱验证，下载后点击 .exe 文件进行解压。

新建工程

由于 µC/OS 需要标准库的支持，所以不再使用 Cube 库，而是直接建立 Keil 工程。

在 Keil 中新建工程，选择开发板信号 STM32F103XX，然后选择需要的环境，勾选 CMSIS->CORE、Device->Startup、Device->StdPeriph Drivers->(Framework | RCC | GPIO)

使用标准库

为了测试环境以及熟悉标准库的使用，我们编写一段裸机程序控制小灯闪烁。在工程中新建 app.c，测试代码如下

```
#include "stm32f10x.h"
#include "stm32f10x_conf.h"

void GPIO_Init(void){
    GPIO_InitTypeDef GPIO_InitStructure;
    RCC_DeInit();
    RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC | RCC_APB2Periph_GPIOA,
ENABLE);
    GPIO_InitStructure.GPIO_Pin = GPIO_Pin_13;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(GPIOC, &GPIO_InitStructure);
}

void Delay(int times){
    unsigned int i;
    for (i = 0; i < times; i++){
    }
}

int main() {
    GPIO_Init();
    while(1){
        GPIO_WriteBit(GPIOC, GPIO_Pin_13, Bit_SET);
        Delay(10000);
        GPIO_WriteBit(GPIOC, GPIO_Pin_13, Bit_RESET);
        Delay(10000);
    }
}
```

在 Keil 中编译代码，并利用 ST-LINK 将二进制代码烧录到开发板，可以看到开发板自带的 PC13 灯闪烁，测试成功。

添加 µC/OS-II

接下来我们需要把 µC/OS-II 源码中的部分代码添加到工程中。

Keil 工程中和实际目录中的文件夹并不对应，为了使结构清晰，我们在目录中分别建立几个文件夹，用于存放源代码：APP、UCOS、BSP、LIB、CPU，*当然，把所有文件放在一个文件夹下也是可以的，但是这样文件混杂在一起会很不清晰。*

在工程中右键 -> Add Group，功能与新建文件夹类似，重命名为与目录中文件夹相同的名字。

添加路径，Options->C/C++->Include Paths, 将上面几个文件夹添加为默认路径。

Options->C/C++->Defines 中添加 USE_STDPERIPH_DRIVER

之后我们需要将 µC/OS-II 源码中的文件分类复制到这几个文件夹下，对应关系如下：

1. UCOS -> Micrium\Software\uCOS-II\Source
2. APP -> Micrium\Software\EvalBoards\ST\STM3210B-EVAL\RVMDK\OS-Probe

3. CPU -> Micrium\Software\uCOS-II\Ports\arm-cortex-m3\Generic\RealView，Micrium\Software\uC-CPU\ARM-Cortex-M3\RealView，Micrium\Software\uC-CPU
4. BSP -> Micrium\Software\EvalBoards\ST\STM3210B-EVAL\RVMDK\BSP
5. LIB -> Micrium\Software\uC-LIB

然后修改一些文件：

1. os_cfg.h，修改 `#define OS_APP_HOOKS_EN 0`
2. bsp.h，注释掉 `#include <stm32f10x_lib.h>` 和 `#include <lcd.h>`
3. app_cfg.h，修改 `#define APP_OS_PROBE_EN 0` 和 `#define APP_PROBE_COM_EN 0`
4. includes.h，注释掉 `#include <stm32f10x_lib.h>` 和 `#include <lcd.h>`

BSP 目录下新建 bsp.c，添加内容：

```
#include <bsp.h>
CPU_INT32U  BSP_CPU_ClkFreq (void) {
    RCC_ClocksTypeDef  rcc_clocks;
    RCC_GetClocksFreq(&rcc_clocks);
    return ((CPU_INT32U)rcc_clocks.HCLK_Frequency);
}
INT32U  OS_CPU_SysTickClkFreq (void) {
    INT32U  freq;
    freq = BSP_CPU_ClkFreq();
    return (freq);
}
```

编写一个函数，接收中断并向 μC/OS-II 内核传递消息：

```
   void SysTick_Handler(void){
       OS_CPU_SR  cpu_sr;
       OS_ENTER_CRITICAL();    // Tell uC/OS-II that we are starting an ISR
       OSIntNesting++;
       OS_EXIT_CRITICAL();
       OSTimeTick();  // Call uC/OS-II's OSTimeTick()
       OSIntExit();   // Tell uC/OS-II that we are leaving the ISR
   }
```

延时函数

由于 while 循环实现的延时函数在 μC/OS-II 中会产生阻塞，任务无法在这里切换。

所以使用 μC/OS-II 的延时函数 `OSTimeDly (INT16U ticks)`，ticks 代表延时的系统节拍数，宏 `OS_TICKS_PER_SEC` 定义了系统节拍的频率，也就是每秒的节拍数，所以 `OSTimeDly(OS_TICKS_PER_SEC)` 会产生一秒的延时。

```
void Delay_s(int times) {
    OSTimeDly(OS_TICKS_PER_SEC * times);
}
void Delay_ms(int times) {
    OSTimeDly(OS_TICKS_PER_SEC * times);
}
```

微妙级别的延时，在 μC/OS 上基本是不可能实现的，如果你不介意高频率切换任务带来的 CPU 浪费，你当然可以把它的节拍调到微妙级别以下，然后通过OSTimeDly来获得这个延时。

若要获得微秒级的延时，采用循环阻塞的方式是可行的，因为如此短的时间不会对任务的切换造成影响。

μC/OS-II 多任务

上述步骤将所需的 μC/OS-II 代码添加到了工程中，我们可以用两个灯测试一下 μC/OS-II 的多任务调度。

```
void LED0_task(void* pdata){
    while(1){
        GPIO_WriteBit(GPIOA, GPIO_Pin_11, Bit_SET);
        Delay_s(1);
        GPIO_WriteBit(GPIOA, GPIO_Pin_11, Bit_RESET);
        Delay_s(1);
    }
}

void LED1_task(void* pdata){
    while(1){
        GPIO_WriteBit(GPIOC, GPIO_Pin_13, Bit_SET);
        Delay_s(2);
        GPIO_WriteBit(GPIOC, GPIO_Pin_13, Bit_RESET);
        Delay_s(2);
    }
}

#define STK_Size 100
int LED0_Task_STK[STK_Size];
int LED1_Task_STK[STK_Size];
int Task_STK[STK_Size];

int main() {
    GPIO_Init();
    OSInit();
    OS_CPU_SysTickInit();
    OSTaskCreate(LED0_task, (void *)0, (OS_STK *)&LED0_Task_STK[STK_Size-1],
1);
    OSTaskCreate(LED1_task, (void *)0, (OS_STK *)&LED1_Task_STK[STK_Size-1],
2);
    OSStart();
    return 0;
}
```

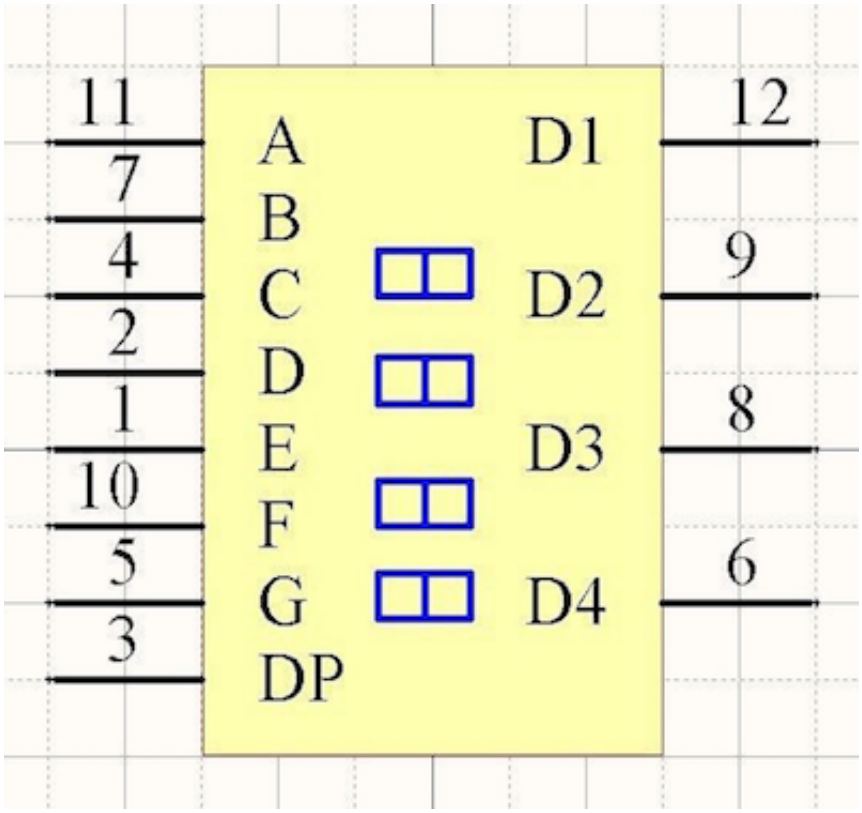
烧录之后，看到两个小灯交替闪烁，说明实现了多任务的调度。

数码管显示

数码管原理

LG3641BH 数码管是共阳极的，每个数码段都有8个 LED 条并连在了一起，有8个引脚对应。

并有 4 个位选信号，输出高电平为选择对应数码管，引脚如下图所示：



分配 PA11, PA12, PC13, PC14 引脚为位选信号，PA0~PA7 为段选信号。

时分复用

为了使 4 个数码管同时点亮，我们采用时分复用的方式，即以高频切换选择数码管，利用视觉暂留的原理，看上去全部点亮。

每 5 毫秒切换位选信号，实现时分复用。对于每个数码管的显示，需要有字库的支持。

```
void digit_select(int index){
    BitAction v[4];
    int i;
    for (i=0; i<4; i++){
        if (index == i){
            v[i] = Bit_SET;
        }else{
            v[i] = Bit_RESET;
        }
    }
    GPIO_WriteBit(GPIOA, GPIO_Pin_11, v[0]);
    GPIO_WriteBit(GPIOA, GPIO_Pin_12, v[1]);
    GPIO_WriteBit(GPIOC, GPIO_Pin_13, v[2]);
    GPIO_WriteBit(GPIOC, GPIO_Pin_14, v[3]);
}

void digit_show(int dight, int point){
    int segment, i, base;
    BitAction v[8];
    switch (dight){
```

```
        case    0    :    segment = 0xee; break; // 0b11101110
        case    1    :    segment = 0x24; break; // 0b00100100
        case    2    :    segment = 0xba; break; // 0b10111010
        case    3    :    segment = 0xb6; break; // 0b10110110
        case    4    :    segment = 0x74; break; // 0b01110100
        case    5    :    segment = 0xd6; break; // 0b11010110
        case    6    :    segment = 0xde; break; // 0b11011110
        case    7    :    segment = 0xa4; break; // 0b10100100
        case    8    :    segment = 0xfe; break; // 0b11111110
        case    9    :    segment = 0xf6; break; // 0b11110110
        default      :    segment = 0xda; break; // 0b11011010 error state
    }
    segment |= point != 0;
    base = 1 << 8;
    for (i=0; i<8; i++){
        base >>= 1;
        if ((segment & base )== 0){
            v[i] = Bit_SET;
        }else{
            v[i] = Bit_RESET;
        }
    }
    GPIO_WriteBit(GPIOA, GPIO_Pin_0, v[0]);
    GPIO_WriteBit(GPIOA, GPIO_Pin_1, v[1]);
    GPIO_WriteBit(GPIOA, GPIO_Pin_2, v[2]);
    GPIO_WriteBit(GPIOA, GPIO_Pin_3, v[3]);
    GPIO_WriteBit(GPIOA, GPIO_Pin_4, v[4]);
    GPIO_WriteBit(GPIOA, GPIO_Pin_5, v[5]);
    GPIO_WriteBit(GPIOA, GPIO_Pin_6, v[6]);
    GPIO_WriteBit(GPIOA, GPIO_Pin_7, v[7]);
}

void led_show(int digit){
    static int index = -1;
    int i;
    int base = 1000;
    index = (index + 1) % 4;
    for (i=0; i<index; i++){
        base /= 10;
    }
    digit = (digit / base) % 10;
    digit_select(index);
    digit_show(digit, 0);
}
```

运行一个任务 `LED_task` 来控制数码管的显示及刷新。

```
void LED_task(void* pdata) {
    while(1) {
        led_show(8888);
        Delay_ms(5);
    }
}
```

DHT-11 传感器

DHT11 数字温湿度传感器是一款含有已校准数字信号输出的温湿度复合传感器，技术手册中说明了引脚功能以及通讯协议。

引脚说明

Pin	名称	注释
1	VDD	供电 3-5.5VDC
2	DAT	串行数据，单总线
3	NC	空脚，请悬空
4	GND	接地，电源负极

数据格式

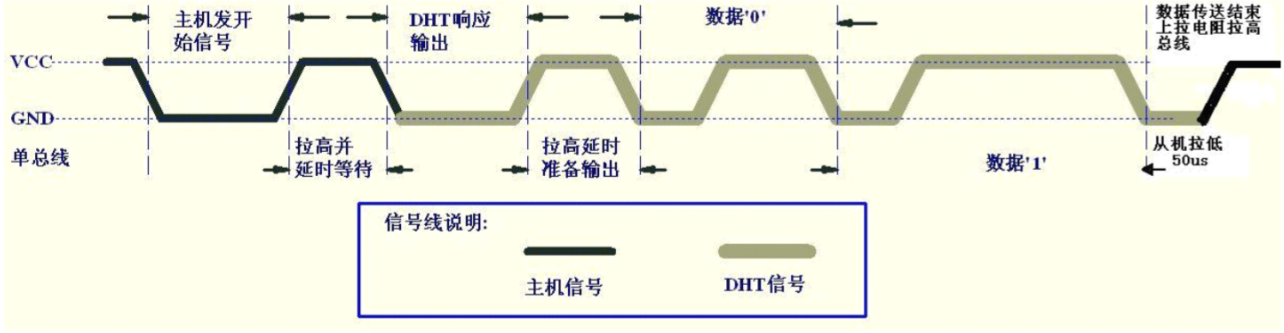
模块采用单线双向的串行通讯，一次通讯时间 4ms 左右，数据分小数部分和整数部分，具体格式如下:

> 一次完整的数据传输为 40bit，高位先出。
>
> 数据格式：8bit湿度整数数据 + 8bit湿度小数数据 + 8bit温度整数数据 + 8bit温度小数数据 + 8bit校验和
>
> 数据传送正确时校验和数据等于前四个数据相加所得结果的末8位。
>
> 用户 MCU 发送一次开始信号后，DHT11 从低功耗模式转换到高速模式，等待主机开始信号结束后，DHT11发送响应信号，送出 40bit 的数据，并触发一次信号采集，用户可选择读取部分数据。

握手协议

数据通讯前要进行设备间的握手，握手协议如下:

1. 总线空闲状态为高电平，主机把总线拉低等待DHT11响应，拉低时间必须大于 18 毫秒，保证DHT11能检测到起始信号。
2. 主机发送开始信号结束后，拉高并延时等待 20~40us
3. DHT11 接收到主机的开始信号后，等待主机开始信号结束，然后发送 80us 低电平响应信号。
4. DHT11 发送响应信号后，再把总线拉高 80us，准备发送数据。



数据接收

握手完成后，开始发送数据，每一 bit 数据都以 50us 低电平时隙开始，高电平的长短定了数据位是 0 还是 1，

高电平持续时间为 26~28μs的为 0，高电平持续时间为 70μs 的表示 1。

DHT11 通讯

根据 DHT11 的通讯协议，编写函数，实现温湿度的读取。

dht11.h 中给出了与 DHT11 通讯的函数接口，调用 `read_dht11()` 即可与传感器模块进行一次通讯，并通过 `get_humidity()` 和 `get_temperature()` 来分别获取温湿度数值。

```
#ifndef dht11_h
#define dht11_h

#include "stm32f10x.h"
#include "stm32f10x_conf.h"

#define DHTLIB_OK                0    // 采集成功
#define DHTLIB_ERROR_CHECKSUM    1    // 校验失败
#define DHTLIB_ERROR_TIMEOUT     2    // 等待超时

#define DHT11_PORT  GPIOC
#define DHT11_PIN   GPIO_Pin_15

int read_dht11();         // 进行一次数据采集
int get_humidity();       // 获得最新采集的湿度值
int get_temperature();    // 获得最新采集的温度值

#endif
```

dht11.c 定义了具体的函数实现

```
#include "dht11.h"

int humidity;
int temperature;

int get_humidity() {
```

```
    return humidity;
}
int get_temperature() {
    return temperature;
}

void DHT11_Set(int state) {
    BitAction s;
    if (state) {
        s = Bit_SET;
    }else {
        s = Bit_RESET;
    }
    GPIO_WriteBit(DHT11_PORT, DHT11_PIN, s);
}

void DHT11_Pin_OUT(){
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = DHT11_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(DHT11_PORT, &GPIO_InitStructure);
    DHT11_Set(1);
}

void DHT11_Pin_IN(){
    GPIO_InitTypeDef GPIO_InitStructure;
    GPIO_InitStructure.GPIO_Pin = DHT11_PIN;
    GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
    GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
    GPIO_Init(DHT11_PORT, &GPIO_InitStructure);
    DHT11_Set(1);
}

void Delay_us(int times){
    unsigned int i;
    for (i=0; i<times; i++){}
}

// Return values:
// DHTLIB_OK
// DHTLIB_ERROR_CHECKSUM
// DHTLIB_ERROR_TIMEOUT
int read_dht11() {
    uint8_t bits[5];    // BUFFER TO RECEIVE
    uint8_t cnt = 7;
    uint8_t idx = 0;
    int i;
```

```
    unsigned int loopCnt = 10000;

    uint8_t sum;

    // EMPTY BUFFER
    for (i=0; i< 5; i++) bits[i] = 0;

    // REQUEST SAMPLE
    DHT11_Pin_OUT();
    DHT11_Set(0);
    Delay_us(30000);
    DHT11_Set(1);
    Delay_us(25);
    DHT11_Set(0);
    DHT11_Pin_IN();


    // ACKNOWLEDGE or TIMEOUT
    loopCnt = 10000;
    while(GPIO_ReadInputDataBit(DHT11_PORT, DHT11_PIN) == LOW)
        if (loopCnt-- == 0) return DHTLIB_ERROR_TIMEOUT;

    loopCnt = 10000;
    while(GPIO_ReadInputDataBit(DHT11_PORT, DHT11_PIN) == HIGH)
        if (loopCnt-- == 0) return DHTLIB_ERROR_TIMEOUT;

    // READ OUTPUT - 40 BITS => 5 BYTES or TIMEOUT
    for (i=0; i<40; i++) {
        loopCnt = 10000;
        while(GPIO_ReadInputDataBit(DHT11_PORT, DHT11_PIN) == LOW)
            if (loopCnt-- == 0) return DHTLIB_ERROR_TIMEOUT;
        loopCnt = 10000;
        while(GPIO_ReadInputDataBit(DHT11_PORT, DHT11_PIN) == HIGH) {
            if (loopCnt-- == 0) return DHTLIB_ERROR_TIMEOUT;
            Delay_us(1);
        }

        if (loopCnt < 9997) bits[idx] |= (1 << cnt);

        if (cnt == 0) {   // next byte?
            cnt = 7;    // restart at MSB
            idx++;      // next byte
        }
        else cnt--;
    }

    // WRITE TO RIGHT VARS
        // as bits[1] and bits[3] are allways zero they are omitted in
formulas.
```

```
    humidity    = bits[0];
    temperature = bits[2];

    // Check sum
    sum = bits[0] + bits[2];
    if (bits[4] != sum) return DHTLIB_ERROR_CHECKSUM;

    return DHTLIB_OK;
}
```

室温计

数码管和 DHT11 分别测试成功后，就可以将它们组合在一起，实现温湿度监测及显示的功能。

运行两个任务，一个任务负责数码管的显示，另一个任务定时请求 DHT11 数据，并在两个任务间共享数据。

需要注意的是，在与 DHT11 的一次通讯过程中，不能有任务的切换。所以在进入临界区代码块之前要关闭中断，以避免任务的切换，通讯结束后再开启中断。

程序定时将温度和湿度交替显示，并以 5s 的周期请求传感器数据。

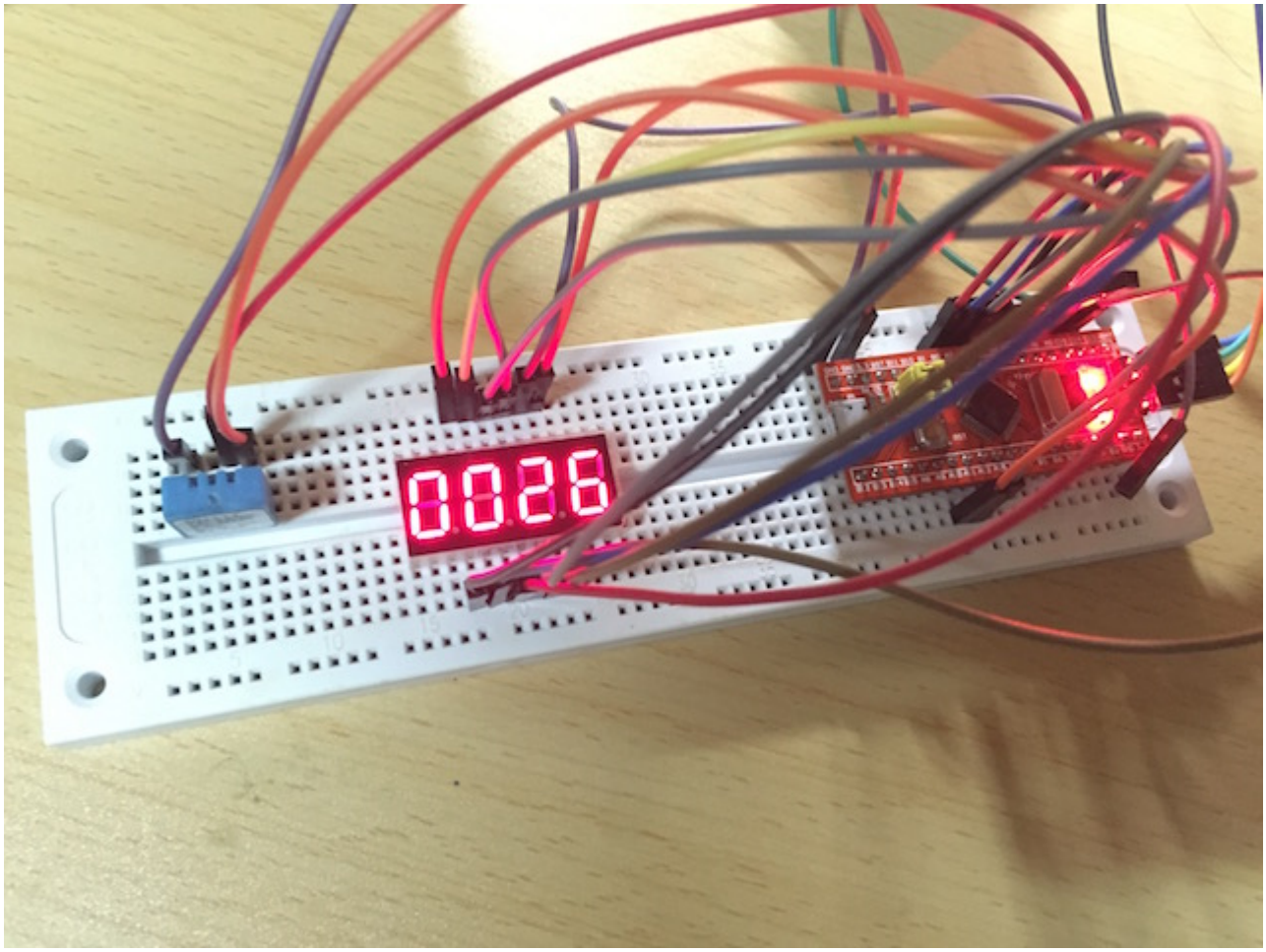
```c
int ledValue;

void LED_task(void* pdata) {
    while(1) {
        led_show(ledValue);
        Delay_ms(5);
    }
}

void DHT11_task(void* pdata){
    int state;
    int cnt = 0;
    while (1){

        OS_ENTER_CRITICAL();     // 进入临界区，关闭中断
        state = read_dht11();    // 请求一次通讯
        OS_EXIT_CRITICAL();      // 离开临界区，开启中断

        switch (state) {
            case DHTLIB_OK:
                if (cnt) ledValue = get_temperature();
                else ledValue = get_humidity();
                break;
            case DHTLIB_ERROR_CHECKSUM:
                ledValue = 9998;
                break;
            case DHTLIB_ERROR_TIMEOUT:
                ledValue = 9999;
                break;
            }
        cnt = 1 - cnt;
        Delay_ms(5000);
    }
}
```


0026