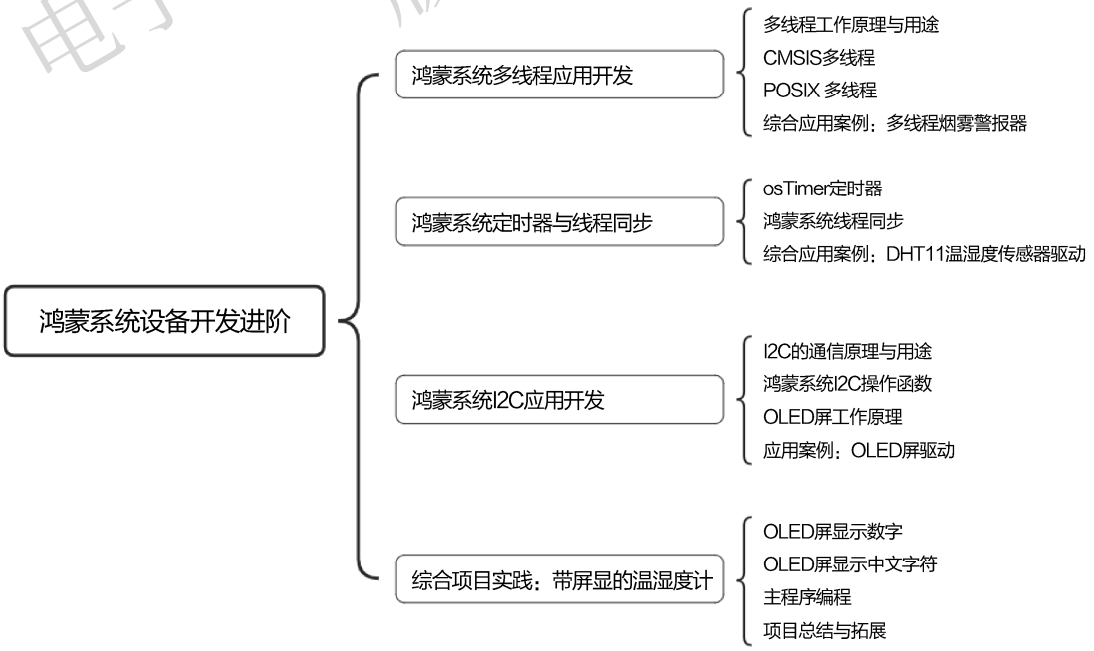
第 3 章

鸿蒙系统设备开发进阶

工作场景

经过第 2 章的学习，我的组员基本熟悉了鸿蒙系统设备的开发流程，已能够完成简单的开发任务，但对鸿蒙系统中的多线程开发技术仍仅处于听说过阶段，并且亦未能完全掌握嵌入式系统中应用广泛的 I2C 通信接口。为了将组员培养成能够独当一面的开发人员，本章将深入讲解 CMSIS 标准与 POSIX 标准下的多线程开发应用，并详细阐述如何利用 POSIX 信号量实现线程间的协调与同步。最后通过 OLED 屏模块的驱动深入讲述 I2C 通信接口的原理、图像与字符像素点阵数据的应用开发技术。

知识图谱



3.1 鸿蒙系统多线程应用开发

在第 2 章的项目实践中，每次烟雾浓度的检测都需要等待耗时比较久的呼吸灯完成后才能执行，这导致传感器数据处理工作的实时性较差，不能在第一时间发出警报。如果呼吸灯与烟雾采集的工作能并行执行，则会大大提高程序响应速率，而且有助于程序的模块化开发。多线程是一种简单、高效的并行执行技术，能够让多个任务同时执行。

3.1.1 多线程工作原理与用途

线程是操作系统能够进行任务调度的最小单位，也是一个程序中并行执行的分支。每个线程都有自己的工作函数与不同的执行路径，且每个线程都独自拥有在内存上分配局部变量的栈空间，但同属一个程序的多个线程共享此程序的全局变量等资源。在多核多线程的主控芯片上，通常由操作系统优先安排一个 CPU 硬件线程来完成线程的任务处理工作，如果硬件线程资源不足则与单核单线程的主控芯片一样，则采用时间片的调度方式来实施线程的并行执行。

多线程的用途广泛，能够充分利用多核多线程处理器的性能；在需要执行耗时操作的情况下，通过将任务分配给不同的线程，可以加速整体处理速度，提高程序的处理能力和性能；采用多线程技术可以将复杂的程序划分为多个相对独立的模块，每个模块负责完成特定的任务，从而简化程序设计，显著提高开发效率。鸿蒙系统既支持 CMSIS（Cortex Microcontroller Software Interface Standard）标准的多线程编程接口，也支持通用性更好的 POSIX（Portable Operating System Interface）标准多线程。

3.1.2 CMSIS 多线程

CMSIS 是 ARM 公司与众多芯片厂商和软件供应商一起紧密合作、联合制定的操作系统与硬件设备之间的通用接口标准。只要遵循 CMSIS 标准，应用程序就可以运行在不同的芯片或不同的操作系统上，而无须修改代码。CMSIS 多线程函数在 cmsis_os2.h 头文件中已声明，具体的编程步骤包括定义线程执行的函数、设置线程属性和线程管理，下面分别说明。

1. 定义线程执行的函数

CMSIS 线程函数原型为 void func(void *arg)，其中 arg 是在函数执行时附带的参数值，func 表示自定义的函数名。在调用 osThreadNew 函数创建线程时除指定 arg 参数值外，还需指定线程要执行的函数名。线程创建成功后自动执行线程函数，当线程函数执行结束，则表示线程退出，因此线程函数内通常会有一个循环的处理过程。循环实现呼吸灯的线程函数代码如下：

```
void myLedThreadFunc(void *arg)
{
    int v=0;
    while (1)
    {
        while (v < 100)
```

```
            {
                IoTPwmStart(LED_PWM, v, 2500);//设置 PWM 控制器的占空比与频率，并启动控制器
                v+=1;
                usleep(1000*10);                    //休眠 10ms
            }
            usleep(1000*500);                       //休眠 500ms
            while (v >= 0)
            {
                IoTPwmStart(LED_PWM, v, 2500);//设置 PWM 控制器的占空比与频率，并启动控制器
                v-=1;
                usleep(1000*10);                    //休眠 10ms
            }
            usleep(1000*500);                       //休眠 500ms
        }
    }
```

2. 设置线程属性

在 CMSIS 多线程编程中，使用 osThreadAttr_t 结构体变量描述线程的名称、栈大小、优先级别等属性值，参考代码如下：

```
    osThreadAttr_t attr;                        //声明描述线程属性的结构体变量
    memset(&attr, 0, sizeof(attr));             //将结构体变量的全部属性值清零
    attr.name = "myLed";                        //设置线程名
    attr.stack_size = 1024;                     //设置线程所用的栈大小
    attr.priority = osPriorityNormal1;          //设置线程优先级别
```

其中线程栈大小的设置非常关键，当线程函数中局部变量空间需求较大，或者函数调用嵌套层次较多时，应当加大栈的空间，否则会发生导致系统崩溃的栈溢出错误。

3. 线程管理

在 CMSIS 多线程编程中，使用 osThreadNew 函数创建线程，函数原型：

```
osThreadId_t osThreadNew(osThreadFunc_t func, void *arg, const osThreadAttr_t *attr)
```

其中 func 参数用于指定线程函数名，arg 参数用于指定当线程函数执行时得到的参数值，attr 参数用于指定使用的线程属性值。使用上一步骤设置的线程属性，创建一个线程执行呼吸灯处理函数的代码如下：

```
osThreadId_t tid = osThreadNew(myLedThreadFunc, NULL, &attr);
```

osThreadNew 函数执行成功会返回线程的 id，根据此线程 id，就可以通过以下函数管理线程：

```
osStatus_t osThreadSuspend(osThreadId_t tid);       //设置 tid 线程暂停执行
osStatus_t osThreadResume(osThreadId_t tid);        //设置 tid 线程恢复执行
osStatus_t osThreadTerminate(osThreadId_t tid);     //结束 tid 线程的执行
```

3.1.3　POSIX 多线程

与偏向嵌入式系统的 CMSIS 标准不同，POSIX 是一种通用的操作系统接口标准，旨在解决不同操作系统之间的兼容性问题，从而在不同操作系统上开发出可移植的通用程序。POSIX 标准得到了广泛的认可和应用，许多操作系统（如 UNIX、Linux）都支持或遵循这一标准，因此在后续章节的多线程开发中应用 POSIX 标准接口。POSIX 线程在 pthread.h 头文

件中提供相关操作的函数声明，具体的编程步骤也包括定义线程执行的函数、设置线程属性和线程管理。

1. 定义线程执行的函数

POSIX 线程与 CMSIS 线程的功能一致，它的线程函数原型为 void func(void *arg)，其中 arg 是在函数执行时附带的参数值，func 为自定义的函数名。线程函数执行结束返回一个地址。例如，循环检测烟雾浓度的线程函数代码：

```
void *myAdcThreadFunc(void *arg)
{
    int ret;
hi_u16 val;
    float vol;

    while (1)
    {
        //获取 ADC5 转换结果，使用转换 8 次平均算法，3.3V 基准电压，等待 3340ns 后开始转换
        ret = hi_adc_read(ADC_CHANNEL, &val, HI_ADC_EQU_MODEL_8, HI_ADC_CUR_BAIS_
3P3V, 10);
        if (HI_ERR_SUCCESS != ret)
        {
            printf("IotAdcRead failed\n");
            break;
        }
        //将 ADC 转换结果再转换成电压值
        vol = hi_adc_convert_to_voltage(val);
        printf("sensor value : %d, voltage: %.2fV\n", val, vol);

        usleep(1000*500);//休眠 500ms
    }
    return NULL;
}
```

2. 设置线程属性

在 POSIX 多线程编程过程中，使用 pthread_attr_t 结构体变量描述线程的属性，并通过函数设置线程的属性值，常用的函数如下：

```
    pthread_attr_t attr;                    //声明线程属性结构体变量

    int pthread_attr_init(pthread_attr_t *attr);//初始化线程结构体变量

    //在线程属性结构体变量中设置线程栈大小
    int pthread_attr_setstacksize(pthread_attr_t *attr, size_t size);

    //在线程属性结构体变量中设置线程的调度算法
    int pthread_attr_setschedpolicy(pthread_attr_t *attr, int policy);
    policy 参数可选：
        SCHED_OTHER（默认值）：分时调度算法，每个线程得到相同的执行时间
        SCHED_FIFO：工作队列调度算法，一个线程执行完才会执行下一个线程
        SCHED_RR：轮流调度算法，以分时为基础，结合线程优先级别进行调度
        除了 SCHED_OTHER 不支持，另外两种调度算法皆支持为线程设置优先级
```

```
//在线程属性结构体变量中设置线程的优先级别
int pthread_attr_setschedparam(pthread_attr_t *attr, const struct sched_param *param);
用法：先创建一个 sched_param 类型的结构体变量，并为其内部的 sched_priority 成员赋一个表示级别的 int 类型值(数值越大则级别越高)，然后执行 pthrerd_attr_setschedparam 函数并传递此 sched_param 变量的地址
```

在鸿蒙系统中，POSIX 线程的栈默认为 4096 字节，当线程函数局部变量空间需求大或函数嵌套调用层次过多时，应当相应加大线程的栈空间。

3. 线程管理

在 POSIX 多线程编程中，使用 pthread_create 函数创建线程，函数原型如下：

```
int pthread_create(pthread_t *tid, const pthread_attr_t *attr,void *(*func)(void *), void *arg);
```

其中，tid 参数用于存放线程 id 的 pthread_t 变量地址，attr 参数用于指定使用的线程属性值，func 参数指定线程执行的函数，arg 参数用于指定当线程函数执行时得到的参数值，函数执行成功返回 0，失败则返回小于 0 的错误码。创建一个线程循环检测烟雾浓度，并设置线程的栈空间为 8192 字节，代码如下：

```
pthread_attr_t attr;                          //声明线程属性结构体变量
pthread_t tid;                                //声明线程 id 变量

pthread_attr_init(&attr);                     //初始化线程属性结构体变量
pthread_attr_setstacksize(&attr, 1024*8);     //设置线程栈为 8KB

//创建线程，指定由 tid 变量存放线程 id，使用 attr 结构体变量的设置，指定线程执行 myAdcThreadFunc 函数，线程函数的 arg 参数值为 NULL
pthread_create(&tid, &attr, myAdcThreadFunc, NULL);
```

pthread_create 函数执行成功后 tid 变量中存放着线程 id。根据此线程 id，就可以通过以下函数管理线程：

```
//获取当前线程的 id
pthread_t pthread_self(void);

//当线程函数执行结束后，通过此函数回收指定 id 的线程资源，并通过 retval 参数接收线程函数的返回值
int pthread_join(pthread_t tid, void **retval);

//设置指定 id 的线程名称，name 为线程名称
int pthread_setname_np(pthread_t tid, const char *name);

//获取指定 id 的线程名称，获取的名称存入 buf 数组中，buflen 参数为 buf 数组的长度
int pthread_getname_np(pthread_t tid, char *buf, size_t buflen);
```

除了上述提到的函数，POSIX 多线程标准接口还包含了诸多其他功能函数。然而，在当前的基于 LiteOS-M 内核的鸿蒙系统中，部分接口函数的功能尚未得到具体实现。

3.1.4　综合应用案例：多线程烟雾警报器

通过多线程并行执行与模块化开发的技术，修改第 2 章的综合项目实践，由一个线程专门实现呼吸灯的功能，再由另一个线程循环检测烟雾传感器，当数据异常时就及时发出警报。

在工程源码的 myhello 目录下，创建 myledThread.c 源文件。在源程序中，由一个 CMSIS

标准的线程，循环定时调节 LED 的 PWM 占空比信号，从而实现呼吸灯的功能。源文件 myledThread.c 的具体代码如下：

```
#include <stdio.h>
#include <ohos_init.h>
#include <unistd.h>
#include <hi_io.h>
#include <iot_gpio.h>
#include <hi_adc.h>
#include <hi_pwm.h>
#include <iot_pwm.h>
#include <cmsis_os2.h>


//PWM 呼吸灯
#define LED_IO              HI_IO_NAME_GPIO_2
#define LED_IOFUNC          HI_IO_FUNC_GPIO_2_PWM2_OUT   //PWM2 的输出引脚功能
#define LED_PWM             HI_PWM_PORT_PWM2             //PWM2

void myLedThreadFunc(void *arg)
{
    int v=0;
    while (1)
    {
        while (v < 100)
        {
            IoTPwmStart(LED_PWM, v, 2500); //设置 PWM 控制器的占空比与频率，并启动控制器
            v+=1;
            usleep(1000*10);               //休眠 10ms
        }
        usleep(1000*500);                  //休眠 500ms
        while (v >= 0)
        {
            IoTPwmStart(LED_PWM, v, 2500); //设置 PWM 控制器的占空比与频率，并启动控制器
            v-=1;
            usleep(1000*10);               /休眠 10ms
        }
        usleep(1000*500);                  //休眠 500ms
    }
}


void myLedInit()
{
    //LED 的 PWM 设置
    IoTGpioInit(LED_IO);
    hi_io_set_func(LED_IO, LED_IOFUNC);    //设置 IO 用途
    IoTPwmInit(LED_PWM);                   //初始化 PWM 控制器

    osThreadAttr_t attr;                   //声明描述线程属性的结构体变量
    memset(&attr, 0, sizeof(attr));        //将结构体变量的全部属性值清零
    attr.name = "myLed";                   //设置线程名
```

```
    attr.stack_size = 1024;                    //设置线程所用的栈大小
    attr.priority = osPriorityNormal1;         //设置线程优先级别

    //创建线程，指定线程函数为 myLedThreadFunc，线程函数的 arg 参数值为 NULL，使用 attr 属性设置
    osThreadId_t tid = osThreadNew(myLedThreadFunc, NULL, &attr);
}

APP_FEATURE_INIT(myLedInit);
```

在工程源码的 myhello 目录下，再创建 myadcThread.c 源文件。在源程序中实现功能：由一个 POSIX 标准的线程，循环定时检测烟雾传感器数据，当数据异常时发生声光警报。myadcThread.c 具体代码如下：

```
#include <stdio.h>
#include <ohos_init.h>
#include <unistd.h>
#include <hi_io.h>
#include <iot_gpio.h>
#include <hi_adc.h>
#include <pthread.h>

//烟雾传感器
#define SENSOR_IO           HI_IO_NAME_GPIO_11
#define SENSOR_IOFUNC       HI_IO_FUNC_GPIO_11_GPIO
#define ADC_CHANNEL         HI_ADC_CHANNEL_5   //GPIO_11 -> ADC5

//继电器
#define RELAY_IO            HI_IO_NAME_GPIO_9
#define RELAY_IOFUNC        HI_IO_FUNC_GPIO_9_GPIO

void *myAdcThreadFunc(void *arg)
{
    int ret;
    hi_u16 val, val2;
    float vol;

    while (1)
    {
        //获取 ADC5 转换结果，使用转换 8 次平均算法，3.3V 基准电压，等待 3340ns 后开始转换
        ret = hi_adc_read(ADC_CHANNEL, &val, HI_ADC_EQU_MODEL_8, HI_ADC_CUR_BAIS_
3P3V, 10);
        if (HI_ERR_SUCCESS != ret)
        {
            printf("IotAdcRead failed\n");
            break;
        }
        //将 ADC 转换结果再转换成电压值
        vol = hi_adc_convert_to_voltage(val);
        printf("sensor value : %d, voltage: %.2fV\n", val, vol);
        //当烟雾传感器输出大于 2.0V 的电压时，表示当前环境可燃气体发生泄漏
        if (2.0f < vol)
        {
```

```
            IoTGpioSetOutputVal(RELAY_IO, IOT_GPIO_VALUE0);//继电器闭合，声光警报器工作
            sleep(3);//最少发出 3s 警报
        }
        else
        {
            //当烟雾传感器输出小于 2.0V 的电压时表示正常，如声光警报器已工作则停止
            IoTGpioGetOutputVal(RELAY_IO, &val2);
            if (IOT_GPIO_VALUE0 == val2)
                IoTGpioSetOutputVal(RELAY_IO, IOT_GPIO_VALUE1);
        }

        usleep(1000*500);                                   //休眠 500ms
    }
}
void myAdcInit()
{
    //烟雾传感器 IO 设置
    IoTGpioInit(SENSOR_IO);                                 //申请使用 IO
    hi_io_set_func(SENSOR_IO, SENSOR_IOFUNC);               //设置 IO 用途
    IoTGpioSetDir(SENSOR_IO, IOT_GPIO_DIR_IN);              //ADC 引脚配置为输入

    //继电器 IO 设置
    IoTGpioInit(RELAY_IO);
    hi_io_set_func(RELAY_IO, RELAY_IOFUNC);
    IoTGpioSetDir(RELAY_IO, IOT_GPIO_DIR_OUT);              //配置为输出
    IoTGpioSetOutputVal(RELAY_IO, IOT_GPIO_VALUE1);         //初始输出高电平

    //线程属性设置
    pthread_attr_t attr;                                    //声明线程属性结构体变量
    pthread_t tid;                                          //声明线程 id 变量

    pthread_attr_init(&attr);                               //初始化线程属性结构体变量
    pthread_attr_setstacksize(&attr, 1024*8);               //设置线程栈为 8KB

    //创建线程，指定由 tid 变量存放线程 id，使用 attr 结构体变量的设置，指定线程执行 myAdcThreadFunc
      函数，线程函数的 arg 参数值为 NULL
    pthread_create(&tid, &attr, myAdcThreadFunc, NULL);
}
APP_FEATURE_INIT(myAdcInit);
```

接着还需要修改 myhello 目录下的 BUILD.gn 编译文件，指定编译 myledThread.c 与 myadcThread.c 两个源文件。BUILD.gn 编译文件修改后的内容如下：

```
static_library("myhello_demo") {
    sources = [
        "myadcThread.c",
        "myledThread.c"
    ]

    include_dirs = [
        "//utils/native/lite/include",
        "//kernel/liteos_m/kal/cmsis",
        "//base/iot_hardware/peripheral/interfaces/kits",
```

```
        ]
}
```

完成上述源文件的修改后，进行工程源码编译、烧录，并重启开发板，即可观察到多线程并行执行的效果。

3.2　鸿蒙系统定时器与线程同步

在物联网设备开发中，经常利用定时器按照预定的间隔时间或时间点触发执行特定的任务，以实现对各种传感器的精确操作。另外，在多线程程序中，线程之间往往需要相互协作。当多个线程同时访问同一个共享资源时，应当实现互斥访问，确保数据的完整性与一致性；当多个线程需要按照一定的顺序或规则来执行时，应当通过合理的线程同步机制，使得线程之间可以按照预定的先后次序进行运行，从而实现了线程之间的协调。

3.2.1　osTimer 定时器

鸿蒙系统的 osTimer 定时器是基于一个 tick 硬件定时器封装出来的软件定时器。

1. osTimer 原理

系统启动后，tick 硬件定时器按系统配置的频率做加 1 计数，在当前版本的系统中，通过 osKernelGetTickFreq 函数获取的计数频率为 100Hz，这意味着每秒钟 tick 硬件定时器会增加 100 次，每次增加的间隔为 10ms。在鸿蒙系统中可以使用多个 osTimer 定时器，但每个 osTimer 定时器的时间应当以 tick 硬件定时器的计数间隔时间为单位（10ms），如创建 0.5s、1s、10s 三个定时器，则三个 osTimer 定时器的间隔时间设置如图 3-1 所示。

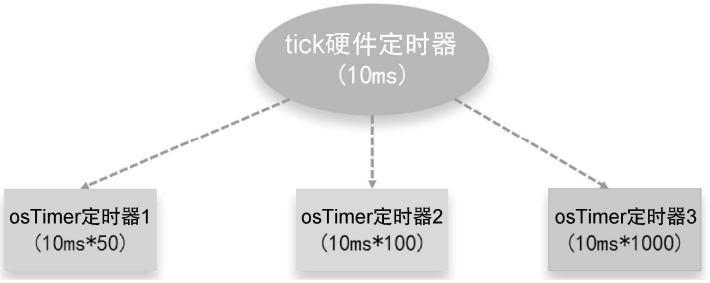


图 3-1　osTimer 定时器的间隔时间设置

2. osTimer 操作函数

在头文件 cmsis_os2.h 中，已声明了 osTimer 相关的操作函数，关键的如下：

```
//创建 osTimer 定时器，并指定超时后执行的函数 func，type 参数指定单次或重复工作，arg 参数指定
  定时器超时处理函数的参数，attr 参数指定定时器属性（可设置为 NULL，表示使用默认属性）
osTimerId_t osTimerNew(osTimerFunc_t func, osTimerType_t type, void *argument, const osTimerAttr_t *attr)
//其中定时器的超时函数原型为 void (*osTimerFunc_t) (void *arg);
//type 参数可选：osTimerOnce(一次性的定时器)，osTimerPeriodic(重复的定时器)
//函数执行成功后，返回 osTimer 定时器的 id，通过此 id 可启动或停止定时器工作
```

```
//启动指定 id 的定时器，并通过 ticks 参数设置定时器间隔时间，如 2s：ticks = 计数频率*2
  osTimerStart(osTimerId_t id, uint32_t ticks)
//注意定时器启动后，在定时器超时前再执行此函数，则定时器会重新开始计数

//停止指定 id 的定时器
osTimerStop(osTimerId_t timer_id)
```

3. 应用案例：按键的定时器驱动

通过 osTimer 定时器每隔 100ms 检查一次按键 IO 的电平，实现按键的驱动。具体代码如下：

```
#include <stdio.h>
#include <ohos_init.h>
#include <hi_io.h>
#include <iot_gpio.h>
#include <cmsis_os2.h>

#define KEY1_IO             HI_IO_NAME_GPIO_11
#define KEY1_FUNC           HI_IO_FUNC_GPIO_11_GPIO
#define KEY2_IO             HI_IO_NAME_GPIO_12
#define KEY2_FUNC           HI_IO_FUNC_GPIO_12_GPIO

osTimerId_t timerID;        //声明定时器 ID
int fKey1 = 0, fKey2 = 0;   //分别标注两个按键是否按下，0 表示放开，1 表示按下

void scanKey(char *name, int io, int *flag)
{
    int v;
    //获取按键 IO 电平
    IoTGpioGetInputVal(io, &v);
    if (v)                  //按键放开处理
    {
        if (*flag)          //如记录按键已按下，则认为按键刚放开
        {
            printf("%s up\n", name);
            *flag = 0;      //表示已放开
        }
    }
    else                    //按键按下处理
    {
        if (!*flag)         //如记录按键已放开则认为按键刚按下
        {
            printf("%s down\n", name);
            *flag = 1;      //表示已按下
        }
    }
}

void timerFunc(void *arg)   //定时器超时函数
{
    scanKey("key1", KEY1_IO, &fKey1);
```

```
    scanKey("key2", KEY2_IO, &fKey2);
}

void initKey(int io, int ioFunc)            //按键 IO 初始化
{
    IoTGpioInit(io);
    hi_io_set_func(io, ioFunc);
    hi_io_set_pull(io, HI_IO_PULL_UP);
    IoTGpioSetDir(io, IOT_GPIO_DIR_IN);
}
void myhello_test()
{
    initKey(KEY1_IO, KEY1_FUNC);
    initKey(KEY2_IO, KEY2_FUNC);

    //创建定时器，指定超时执行的函数、定时器类型、函数参数、属性配置
    timerID = osTimerNew(timerFunc, osTimerPeriodic, NULL, NULL);
    //启动定时器，100ms 扫描
    osTimerStart(timerID, 10);
}

SYS_RUN(myhello_test);
```

3.2.2　鸿蒙系统线程同步

线程同步在多线程编程中是实现线程间通信与协作的必要手段，常用的线程同步机制有互斥锁（Mutex）、读写锁（Read-Write Locks）、信号量（Semaphores）等。本书以 POSIX 信号量实现线程的同步。

1. 信号量原理

信号量（Semaphore）是操作系统中一种重要的线程同步机制，主要用于对共享资源的保护访问。信号量以一个非负整数值表示可用资源的数量，当信号量值大于 0 时，表示有资源可用。此时，若对信号量执行上锁操作，信号量值会减一，并允许线程继续执行。当信号量值等于 0 时表示已没有可用资源，此时上锁会让当前线程进入休眠阻塞状态，直到有可分配的资源才会唤醒并恢复执行。信号量的解锁操作会让信号量值加一，如果有等待此信号量资源的其他线程时，解锁操作会唤醒一个处于阻塞状态的线程。

2. 信号量操作函数

POSIX 信号量在 semaphore.h 头文件中声明了 sem_t 信号量类型及相关操作函数，关键的函数如下：

```
//信号量初始化
int sem_init(sem_t *sem, int pshared, unsigned int value);
//参数列表：
//sem 参数指定要初始化的变量
//pshared 参数值非零表示此信号量支持跨线程，零值则表示只支持线程内访问
//value 参数设置信号量初始时的可用资源数
```

```
//对 sem 信号量做上锁操作，如信号量值已为 0，调用的线程则进入休眠阻塞状态，直到有可用资源为止；如信号量值大于 0，则信号量值减一，上锁成功，接着执行
int sem_wait(sem_t *sem);

//对 sem 信号量做解锁操作，让信号量值加一，唤醒一个等待此信号量可用资源的阻塞线程
int sem_post(sem_t *sem);
```

3. 应用案例：线程与按键的同步

以一个线程循环对信号量做上锁操作并进入休眠阻塞状态，在按键按下时对信号量做解锁操作，唤醒与恢复线程的执行，具体代码如下：

```
#include <stdio.h>
#include <unistd.h>
#include <ohos_init.h>
#include <cmsis_os2.h>
#include <hi_time.h>
#include <hi_io.h>
#include <iot_gpio.h>
#include <pthread.h>
#include <semaphore.h>

sem_t sem;                    //声明信号量变量

//按键 IO 与用途
#define KEY_IO              HI_IO_NAME_GPIO_11
#define KEY_IOFUNC          HI_IO_FUNC_GPIO_11_GPIO

void keyIrqFunc(void *arg)  //按键中断处理函数
{
    sem_post(&sem);         //信号量解锁，唤醒休眠的线程
}

void *threadFunc(void *arg)
{
    int n = 0;              //用于记录线程休眠次数
    while (1)
    {
        printf("before lock %d\n", n);
        sem_wait(&sem); //信号量上锁，没有资源则进入休眠状态
        printf("after lock\n");
        n++;
    }
    return NULL;
}

void myhelloInit()
{
    //按键 IO 初始化与中断配置
    IoTGpioInit(KEY_IO);
    hi_io_set_func(KEY_IO, KEY_IOFUNC);
    IoTGpioSetDir(KEY_IO, IOT_GPIO_DIR_IN);
```

```
        hi_io_set_pull(KEY_IO, HI_IO_PULL_UP);
        hi_io_set_schmitt(KEY_IO, HI_TRUE);//中断信号过滤
        //注册按键 IO 中断，设置边沿中断类型，下降沿触发中断，中断处理函数及函数的参数
        IoTGpioRegisterIsrFunc(KEY_IO, IOT_INT_TYPE_EDGE, IOT_GPIO_EDGE_FALL_LEVEL_LOW,
keyIrqFunc, NULL);

        //初始化信号量，支持跨线程访问信号量，可用资源为 0
        sem_init(&sem, 1, 0);

        pthread_t tid;//存放线程的 id
        //创建线程
        pthread_create(&tid, NULL, threadFunc, NULL);
    }
    APP_FEATURE_INIT(myhelloInit);
```

3.2.3　综合应用案例：DHT11 温湿度传感器驱动

DHT11 是一款输出高/低电平数字信号的温湿度传感器。DHT11 模块如图 3-2 所示。

1. DHT11 通信过程

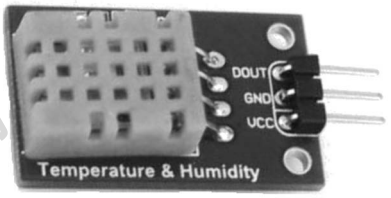


图 3-2　DHT11 模块

DHT11 模块通过 DOUT 引脚接收开始采集温湿度信号，并通过 DOUT 引脚输出温湿度与数据校验和。一次完整的数据传输为 40bit，高位先出。先后输出：8 位湿度整数数据、8 位湿度小数数据、8 位温度整数数据、8 位温度小数数据、8 位校验和。其中，校验和的值应当是前面 4 字节温湿数据累加后所得结果的末 8 位。

DHT11 只有接收到开始信号后才会进行温湿度数据的采集工作，DHT11 开始信号如图 3-3 所示。图中的黑色部分就是开始信号的时序。首先，连接模块 DOUT 引脚的 GPIO 应当接上拉，默认处于高电平状态，GPIO 应当输出 18ms 以上的低电平，再接着输出 20～40μs 的高电平信号。开始信号发出后，GPIO 应当改为输入功能，由 DHT11 模块控制 GPIO 的电平。DHT11 接收到开始信号后，输出 80μs 的低电平的响应信号，正常情况下，通过 GPIO 即获取到此低电平信号；如果在 DHT11 未接入等不正常情况下，因 GPIO 的上拉功能，GPIO 获取到高电平信号。

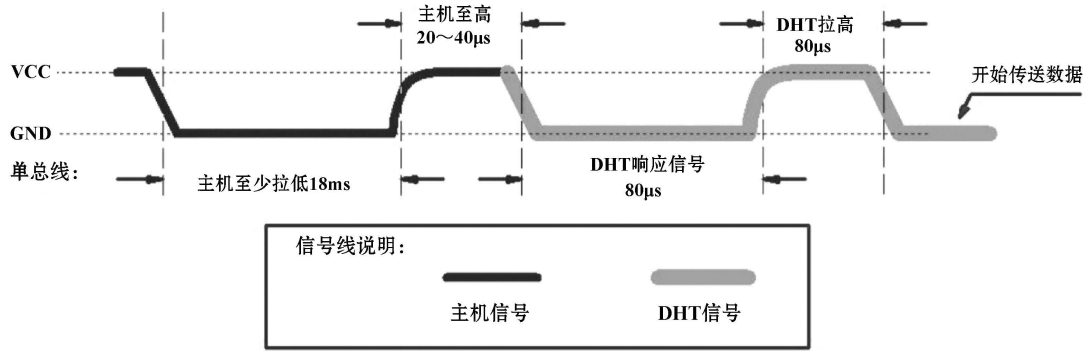


图 3-3　DHT11 开始信号

DHT11 输出响应信号后，就会发出表示二进制数值 0 或 1 的电平信号。二进制数值 0 的电平信号如图 3-4 所示。DOUT 引脚先输出 50μs 低电平信号，接着输出 26～28μs 的高电平信号。

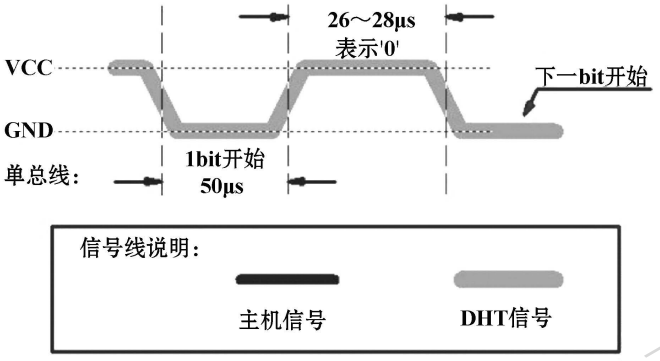


图 3-4　二进制数值 0 的电平信号

DHT11 输出的二进制数值 1 的电平信号如图 3-5 所示。DOUT 引脚先输出 50μs 低电平信号，接着输出 70μs 的高电平信号。

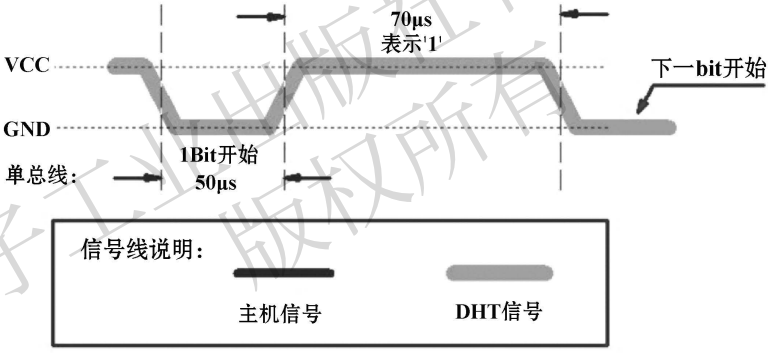


图 3-5　二进制数值 1 的电平信号

综上所述，当 DHT11 接收到开始信号后，就会输出二进制数值 0 与 1 的电平信号。二进制数值 0 与 1 的电平信号区别在于高电平持续时间。从电平信号的下降沿到下一个下降沿，二进制数值 0 的周期为 76～78μs，而二进制数值 1 的周期为 120μs。因此，只要以下降沿开始计算每个电平信号的周期时间，就可获取温湿度数据的每位二进制值，然后以 8 位为单位组合数据，即可获取温度与湿度的具体数值。

2. DHT11 驱动实现

DHT11 模块的 VCC、GND、DOUT 引脚通过杜邦线分别连接开发板的 3.3V、GND、IO10 引脚。初始化 GPIO 并根据图 3-3 向 DHT11 发出开始信号，功能实现代码如下：

```
#define DHT11IO              HI_IO_NAME_GPIO_10
#define DHT11IO_FUNC     HI_IO_FUNC_GPIO_10_GPIO
void dht11Start()
{
    IoTGpioInit(DHT11IO);
    hi_io_set_func(DHT11IO, DHT11IO_FUNC);
```

```
    hi_io_set_pull(DHT11IO, HI_IO_PULL_UP);        //配置上拉，IO 口默认高电平状态
    IoTGpioSetDir(DHT11IO, IOT_GPIO_DIR_OUT); //配置输出功能，可控制输出电平

    IoTGpioSetOutputVal(DHT11IO, 0);              //输出低电平
    usleep(20000);                                //根据图 3-3,最少需要 18ms,因此这里保守设置 20ms
    IoTGpioSetOutputVal(DHT11IO, 1);              //输出高电平
    hi_udelay(50);//注意这里使用的是不会休眠的函数延时 50μs。根据图 3-3，主机需要拉高 20～40μs，
                  延时 50μs 时间，这样是为了确保输出的电平信号满足 DHT11 的响应信号周期
    IoTGpioSetDir(DHT11IO, IOT_GPIO_DIR_IN);//配置为输入，获取 DHT11 模块 DOUT 输出的电平
}
```

DHT11 接收到开始信号后，采集温湿度数据，并通过 DOUT 引脚向 IO10 输出如图 3-4 与图 3-5 所示的电平信号，通过计算 IO10 下降沿中断的间隔时间即可获取二进制数值 0 与 1。注册中断代码如下：

```
IoTGpioRegisterIsrFunc(DHT11IO, IOT_INT_TYPE_EDGE, IOT_GPIO_EDGE_FALL_LEVEL_LOW,
    dht11Isr, NULL);
```

在中断处理函数中获取并记录当前系统的微秒时间，功能实现代码如下：

```
int n = 0; //记录当前第几次中断
int times[41], //记录中断时系统的微秒时间，5 字节 40 位需要 41 次中断(每位一次中断，还有最初开
            始的一次)
    vals[5];                  //存放温度、湿度数据与校验和
void dht11Isr(void *arg)
{
    times[n] = hi_get_us(); //保存当前系统的微秒时间，基于相邻的间隔时间，从而确定 0 或 1
    n++;
}
```

在完成 41 次中断处理后，将每次中断的间隔时间转换成二进制数值 0 或 1，并组合成分别对应湿度整数、湿度小数、温度整数、温度小数、校验和等的字节数据，功能实现的代码如下：

```
for (int i = 0; i < 5; i++)       //i 表示第几字节
{
    int v = 0;                    //暂存第 i 字节数据，初始为 0
    for (int j = 0; j < 8; j++)//第 i 字节的第 j 位数据
    {   //中断间隔时间大于 100μs 则是二进制数值 1，否则是二进制数值 0
        if ((times[i*8+j+1] - times[i*8+j]) > 100)
            v |= 1<<(7-j);//从高位顺序(第 7 位开始)存入，数据 1 需要对应第 7−j 位，以此类推。数
                          据 0 不用移位，因前面 v = 0，所有数据位已设 0，只要相应位设 1 即可
    }
  vals[i] = v;                    //存放数组
}
```

3. 编程实现

以一个线程循环每 3s 向 DHT11 发送启动信号，待接收 DHT11 的响应后，注册 IO 中断，并上锁信号量使线程进入休眠状态，直至温湿度数据接收并处理完毕，再解锁信号量唤醒线程，最终打印输出数据。同时，为了避免受到干扰，从而导致丢失每次采集操作的中断信号，程序中使用 osTimer 定时进行数据转换处理。具体功能在 mydht11Thread.c 源文件中实现，源文件代码如下：

```
#include <stdio.h>
```

```
#include <unistd.h>
#include <ohos_init.h>
#include <cmsis_os2.h>
#include <hi_time.h>
#include <hi_io.h>
#include <iot_gpio.h>
#include <pthread.h>
#include <semaphore.h>
/*
DHT11 vcc --> 3V3
DHT11 GND --> GND
DHT11 DATA --> GPIO07
*/
#define DHT11IO             HI_IO_NAME_GPIO_7
#define DHT11IO_FUNC    HI_IO_FUNC_GPIO_7_GPIO

static osTimerId_t timerID;         //存放定时器 ID
static sem_t            semLock;    //信号量变量
int n = 0;                          //记录当前第几次中断
int times[41],                      //记录中断时系统的微秒时间
    vals[5];                        //存放温度、湿度数据与校验和

void timerFunc(void *arg)           //定时读取数据
{
    for (int i = 0; i < 5; i++)     //i 表示第几字节
    {
        int v = 0;                  //暂存第 i 字节数据，初始为 0
        for (int j = 0; j < 8; j++) //第 i 字节的第 j 位数据
        {  //中断间隔时间大于 100μs 则是二进制数值 1，否则是二进制数值 0
            if ((times[i*8+j+1] - times[i*8+j]) > 100)
                v |= 1<<(7-j);  //从高位顺序存入
        }
       vals[i] = v;                 //存放数组
    }
    sem_post(&semLock);             //信号量解锁
}

void dht11Isr(void *arg)            //中断时调用
{
    times[n] = hi_get_us();
    n++;
    //启动定时器，1s 后执行 timerFunc 函数。注意，若未超时并再次启动，定时器会重新计时
    osTimerStart(timerID, 100);//正常情况下，设备会产生 41 中断，但电路有可能遭到干扰，从而导
                               致发生丢失中断信号的情况，所以这里设置定时器，不管这次数据
                               是否完整，都会定时重新开始，计算中断的次数
}

void dht11Start()
{
    IoTGpioInit(DHT11IO);
    hi_io_set_func(DHT11IO, DHT11IO_FUNC);
```

```
        hi_io_set_pull(DHT11IO, HI_IO_PULL_UP);        //配置上拉，IO 口默认高电平状态
        IoTGpioSetDir(DHT11IO, IOT_GPIO_DIR_OUT); //配置输出功能，可控制输出电平

        IoTGpioSetOutputVal(DHT11IO, 0);            //输出低电平
        usleep(20000);//20ms
        IoTGpioSetOutputVal(DHT11IO, 1);            //输出高电平
        hi_udelay(50);//50us
        IoTGpioSetDir(DHT11IO, IOT_GPIO_DIR_IN);//配置为输入，获取 DHT11 模块 DOUT 引脚输出的电平
    }

    static void *threadFunc(void *arg)
    {
        int v;
        while (1)
        {
            dht11Start();
            //检查 DHT11 的响应信号，获取模块 DOUT 引脚输出的低电平
            IoTGpioGetInputVal(DHT11IO, &v);
            if (v)
            { //如没有接收到 DHT11 输出的低电平响应信号，则退出循环
                printf("no dht11 found\n");
                break;
            }
            n = 0;
            //注册 GPIO 中断
            IoTGpioRegisterIsrFunc(DHT11IO, IOT_INT_TYPE_EDGE, IOT_GPIO_EDGE_FALL_LEVEL_
LOW, dht11Isr, NULL);

            //信号量上锁，当前线程进入休眠状态，直到其他线程解锁信号量，由系统唤醒为止
            sem_wait(&semLock);
            //线程被唤醒后，输出获取的湿度整数、湿度小数、温度整数、温度小数、校验和
            for (int i = 0; i < 5; i++)
                printf("%d ", vals[i]);
            printf("\n");

            sleep(3);
            IoTGpioUnregisterIsrFunc(DHT11IO);//释放 GPIO 的中断功能
        }
        return NULL;
    }


    void mydht11Init()
    {
        pthread_t tid;
        //初始化信号量
        sem_init(&semLock, 1, 0);
        //创建线程
        pthread_create(&tid, NULL, threadFunc, NULL);
        //创建 osTimer 定时器
        timerID = osTimerNew(timerFunc, osTimerOnce, NULL, NULL);
```

```
}
APP_FEATURE_INIT(mydht11Init);
```

程序编译、烧录后，复位开发板，就会在终端上定时输出获取的湿度与温度数据。

3.3 鸿蒙系统 I2C 应用开发

I2C（Inter-Integrated Circuit），也写为 I2C 或 IIC。它既是一种通信协议，也是一种硬件接口，可用于连接 I2C 设备，实现硬件模块间的通信。在物联网系统中，I2C 是常见的通信接口，常用于获取触摸屏的触点坐标、控制声卡的音量、配置摄像头的图像分辨率等操作。

3.3.1 I2C 的通信原理与用途

如图 3-6 所示，I2C 接口是由两根导线组成的：一根是数据线 SDA，另一根是时钟线 SCL。时钟线提供周期信号，每个周期信号表示数据线上传输一位数据；数据线在每个时钟周期中，用高电平表示二进制数值 1，低电平表示二进制数值 0。此外，I2C 传输以 8 位为一个单位，每传输 8 位后，应由接收方拉低 SDA 引脚电平作为应答 ACK 信号。

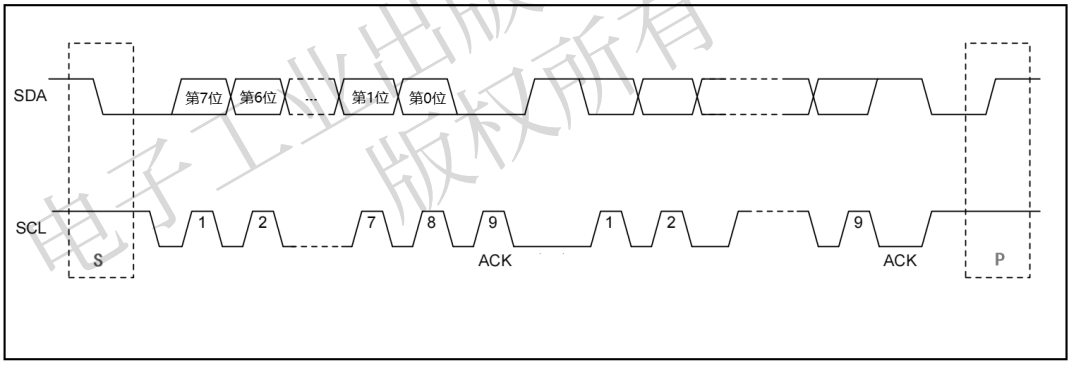


图 3-6 I2C 通信原理

在 I2C 设备通信过程中，通常按功能分成主机（Master）和从机（Slave）。主机表示发起通信操作方，而从机表示被操作方。I2C 设备通常情况下都是扮演从机的角色。SCL 和 SDA 这两根导线默认处于高电平状态，通信的开始和结束由以下两个信号表示。

（1）开始信号（图中的 S 标注）：SCL 高电平状态，SDA 处于从高电平到低电平的下降沿状态变化信号，表示 I2C 传输的开始。

（2）停止信号（图中的 P 标注）：SCL 高电平状态，SDA 处于从低电平到高电平的上升沿状态变化信号，表示 I2C 传输的结束。

虽然 I2C 的接口只有两根导线，但一个接口可并联接入多个 I2C 设备，如图 3-7 所示。在同一接口上的每个 I2C 设备都需要有一个不同的设备地址（通常是 7 位地址）予以区分，每次 I2C 通信时都需要指定所操作的设备地址，而且通常情况下，一次 I2C 通信过程只能访问一个设备。

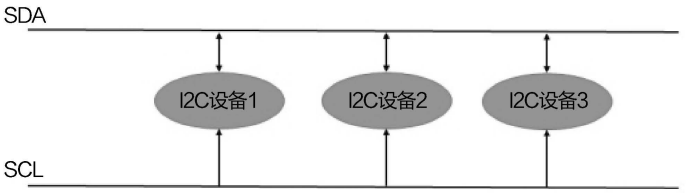


图 3-7　I2C 接口

I2C 通信协议如图 3-8 所示。

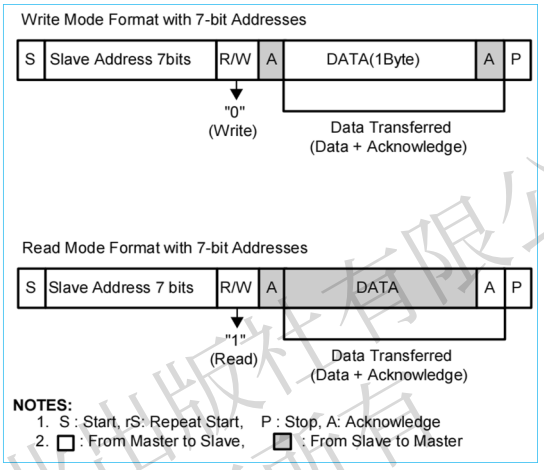


图 3-8　I2C 通信协议

Write Mode（向从机设备发送数据）：主机先发送开始信号，接着发送 8 位数据（由 7 位设备地址和 1 位读写位组成），读写位的值为 0。同一 I2C 接口的所有设备都会在接收到此设备地址后，与自身的地址进行比较，如果地址一样，则由匹配的设备回复应答信号。如果没有设备匹配设备地址，则没有应答信号。主机收到应答信号后，发送 8 位数据，从机收到数据回复应答，表示已收到数据。主机如需再发数据，则在接收到应答信号后，再发送 8 位数据，如此循环，最后发送停止信号。

Read Mode（读取从机设备数据）：主机先发送开始信号，接着发送 8 位数据（由 7 位设备地址和 1 位读写位组成），读写位的值为 1。从机地址匹配后，回复应答信号，接着从机输出 8 位数据，而主机接收数据后，如需再接收数据，则向从机回复应答信号。从机收到应答信号后，再发送 8 位数据，如此循环，直到主机停止接收，发送停止信号。

3.3.2　鸿蒙系统 I2C 操作函数

通常情况下，在低端的单片机计算机系统中，I2C 通信往往需要通过控制 GPIO 的电平，按协议来实现，这样的通信方式较为麻烦，开发难度较大。Hi3861 芯片内置了 I2C 控制器，该控制器能够依据 I2C 通信协议自动按时调节 SDA 引脚和 SCL 引脚的电平状态，从而实现 I2C 数据的传输功能。当发送数据时，可以先通过调用鸿蒙系统的 I2C 操作函数将传输的数据提交到 I2C 控制器，再由控制器负责控制 SDA 引脚和 SCL 引脚的电平来实现数据的传输；当接收数据时，可以先由控制器负责根据 SCL 的时钟信号来判断 SDA 引脚的电平，接

收并存储数据，然后通过调用鸿蒙系统 I2C 操作函数从控制器里取回数据即可。

鸿蒙系统在 iot_i2c.h 头文件中提供了 I2C 控制器操作函数，关键的函数如下：

```
//初始化 I2C 控制器
unsigned int IoTI2cInit(unsigned int id, unsigned int baudrate);
//参数 id 指定操作第几个控制器，参数 baudrate 指定 I2C 传输速率(标准 100kHz, 高速 400kHz)
//执行成功返回 IOT_SUCCESS，执行失败则返回 IOT_FAILURE

//通过 I2C 控制器发送数据
unsigned int IoTI2cWrite(unsigned int id, unsigned short devAddr, const unsigned char *data, unsigned int dataLen);
//参数 id 指定操作第几个控制器，参数 devAddr 指定操作的设备地址(包含读写位)
//参数 data 为存放要发送数据的缓冲区地址，参数 dataLen 指定要发送的数据长度
//执行成功返回 IOT_SUCCESS，执行失败则返回 IOT_FAILURE

//通过 I2C 控制器接收数据
unsigned int IoTI2cRead(unsigned int id, unsigned short devAddr, unsigned char *data, unsigned int dataLen);
//参数 id 指定操作第几个控制器，参数 devAddr 指定操作的设备地址(包含读写位)
//参数 data 为存放接收数据的缓冲区地址，参数 dataLen 指定要接收的数据长度
//执行成功返回 IOT_SUCCESS，执行失败则返回 IOT_FAILURE
```

Hi3861 里有多个 IO 可作为 I2C 控制器的 SDA 或 SCL 引脚，这里选 IO13、IO14 作为 I2C0 控制器的引脚功能。初始化 I2C0 控制器的代码如下：

```
#define OLED_SDA        HI_IO_NAME_GPIO_13
#define OLED_SDAFUNC HI_IO_FUNC_GPIO_13_I2C0_SDA
#define OLED_SCL        HI_IO_NAME_GPIO_14
#define OLED_SCLFUNC HI_IO_FUNC_GPIO_14_I2C0_SCL
//OLED 屏所接的 I2C 接口初始化
static void oledI2cInit()
{
    IoTGpioInit(OLED_SDA);
    IoTGpioInit(OLED_SCL);
    //配置 IO 口由 I2C 控制器使用
    hi_io_set_func(OLED_SDA, OLED_SDAFUNC);
    hi_io_set_func(OLED_SCL, OLED_SCLFUNC);
    //配置上拉
    hi_io_set_pull(OLED_SDA, HI_IO_PULL_UP);
    hi_io_set_pull(OLED_SCL, HI_IO_PULL_UP);
    //初始化第 0 个 I2C 控制器
    IoTI2cInit(0,   400000);//I2C 传输率为 400kHz
}
```

3.3.3 OLED 屏工作原理

OLED，全称是 Organic Light-Emitting Diode，即有机发光二极管。OLED 屏也就是使用一个发光二极管的亮与灭分别表示图像每个像素点的白与黑的屏。由于存在能发出不同颜色光的二极管，OLED 屏因此既有白、蓝、黄等单色屏，也有由红、绿、蓝三个不同的发光二极管共同构成每个像素颜色的 OLED 彩屏。本书采用单蓝色的 OLED 屏模块，如图 3-9 所示。

此 OLED 屏在物理上一行有 128 个像素点，共有 64 行，因此有些场合也称这种屏为

12864 屏。屏幕上的每个像素点，通过控制其对应的发光二极管来显示黑色或蓝色，而图像与字符的呈现，则依赖由多个像素点构成的点阵来实现。如图 3-10 所示，K 字符 16×8 点阵的像素数据分成两行，每行 8 字节数据，每字节表示从低位到高位上下 8 行一列的像素点数据。

精准地控制 OLED 屏每个像素点的发光二极管是一份不容出错的工作，因此为了便于开发应用，每个 OLED 屏都会集成一个驱动 IC，因此通常将集成驱动 IC 的屏称为屏模组。OLED 屏驱动 IC 将通信接口传输的像素点阵 0 或 1 二进制数据存入内部的显存中，并负责根据显存中的像素点数据控制对应的发光二极管进行工作。本次 OLED 屏模组采用了 SSD1306 驱动 IC，该驱动 IC 内部集成了 128 像素×64 像素大小的像素点阵的显存及众多配置寄存器，并支持高速的 I2C 通信接口。OLED 屏的驱动流程如图 3-11 所示。

图 3-9　OLED 屏

```
0x08,0xF8,0x88,0xC0,0x28,0x18,0x08,0x00,
0x20,0x3F,0x20,0x01,0x26,0x38,0x20,0x00
    0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0
    1 1 1 0 1 1 1 0
    0 1 0 0 0 1 0 0
    0 1 0 0 1 0 0 0
    0 1 0 1 0 0 0 0
    0 1 1 1 0 0 0 0
    0 1 0 1 0 0 0 0
    0 1 0 0 1 0 0 0
    0 1 0 0 1 0 0 0
    0 1 0 0 0 1 0 0
    0 1 0 0 0 1 0 0
    1 1 1 0 1 1 1 0
    0 0 0 0 0 0 0 0
    0 0 0 0 0 0 0 0
```

图 3-10　字符点阵

图 3-11　OLED 屏的驱动流程

如图 3-11 所示，SSD1306 本质上作为一个 I2C 设备，可以先通过 Hi3861 芯片上的 I2C 控制器向它传输配置参数与像素点数据，然后由驱动 IC 负责控制 OLED 屏的发光二极管，实现图像或字符的显示。

3.3.4　应用案例：OLED 屏驱动

为了提高硬件模块对项目的适配性，在购买屏模块时，厂家会提供一系列的应用案例程序源码与芯片手册，开发人员可以基于此进行 OLED 屏在 Hi3861 芯片上的驱动开发。根据芯片手册，SSD1306 芯片在作为 I2C 设备时，其设备地址为 0x3C。虽然在物理上 OLED 屏的像素点阵共分成 64 行像素点，每行 128 个像素点，但在厂家提供的案例程序中，OLED 屏的像素点阵仅仅分成 8 个逻辑行，每个逻辑行由物理上的 8 行组成，每行由 128 个像素点组成，如图 3-12 所示。

图 3-12　像素排列

1. OLED 屏接入

I2C 接口的 OLED 屏模块电路接入较为简单，模块引脚如图 3-9 所示，只需接入电源与 I2C 共 4 个接口即可。OLED 屏模块引脚与开发板的连接如下：

```
OLED 屏模块                          开发板
    VCC          -------              3.3V
    GND          -------              GND
    SDA          -------              IO13 //I2C0_SDA
    SCL          -------              IO14 //I2C0_SCL
```

2. OLED 屏初始化

创建 IoTOled.h 头文件与 IoTOled.c 源文件，在 IoTOled.h 头文件中声明 OLED 屏的功能函数与自定义类型，文件的具体内容如下：

```
#ifndef __IOTOLED_H__
#define __IOTOLED_H__

typedef unsigned char u8;                    //自定义类型

#define SSD1306_I2CADDR       0x3C //SSD1306 驱动 IC 的设备地址

//在 OLED 屏上显示图像，x、y 表示屏显的始坐标，bmpW 表示图像宽的像素数，bmpH 表示图像高
  的像素数，bmpData 表示图像数据缓冲区的首地址
void oledShowBmp(u8 x, u8 y, u8 bmpW, u8 bmpH, u8 *bmpData);

//在 OLED 屏上显示中文字符，x、y 表示屏显的始坐标，index 表示第几个中文字符
void oledShowChinese(u8 x, u8 y, u8 index);

//在 OLED 屏上显示浮点数，x、y 表示屏显的始坐标，n 表示要显示的浮点数
void oledShowFloat(u8 x, u8 y, float n);

//清屏，让整个屏幕成为黑色
void oledClear(void);

#endif // __IOTOLED_H__
```

接着在 IoTOled.c 源文件中根据屏的 51 单片机例程源码实现 OLED 屏在 Hi3861 芯片上的驱动。先初始化 Hi3861 芯片的 I2C 控制器，然后通过 I2C 控制器传输配置参数初始化 SSD1306 驱动 IC，让 OLED 屏启动点亮屏，IoTOled.c 源文件中的具体内容如下：

```
#include <math.h>
```

```
#include <stdlib.h>
#include <string.h>
#include <stdio.h>
#include <unistd.h>
#include <ohos_init.h>
#include <hi_io.h>
#include "iot_i2c.h"
#include "IoTOled.h"

#define OLED_SDA            HI_IO_NAME_GPIO_13
#define OLED_SDAFUNC  HI_IO_FUNC_GPIO_13_I2C0_SDA
#define OLED_SCL            HI_IO_NAME_GPIO_14
#define OLED_SCLFUNC   HI_IO_FUNC_GPIO_14_I2C0_SCL

//向 SSD1306 发送命令
static void oledSendCmd(u8 val)
{
    u8 data[] = {0x00, val}; //SSD1306 的命令寄存器地址为 0x00
    if (IoTI2cWrite(0, SSD1306_I2CADDR<<1, data, sizeof(data)) < 0)
        printf("oled write command failed\n");
}

//向 SSD1306 发送数据
static void oledSendData(u8 val)
{
    u8 data[] = {0x40, val};
    if (IoTI2cWrite(0, SSD1306_I2CADDR<<1, data, sizeof(data)) < 0)
        printf("oled write data failed\n");
}

static void oledInit()
{
    oledSendCmd(0xAE);//--turn off oled panel
    oledSendCmd(0x00);//---set low column address
    oledSendCmd(0x10);//---set high column address
    oledSendCmd(0x40);//--set start line address    Set Mapping RAM Display Start Line (0x00~0x3F)
    oledSendCmd(0x81);//--set contrast control register
    oledSendCmd(0xCF); // Set SEG Output Current Brightness
    oledSendCmd(0xA1);//--Set SEG/Column Mapping        0xa0 左右反置 0xa1 正常
    oledSendCmd(0xC8);//Set COM/Row Scan Direction      0xc0 上下反置 0xc8 正常
    oledSendCmd(0xA6);//--set normal display
    oledSendCmd(0xA8);//--set multiplex ratio(1 to 64)
    oledSendCmd(0x3f);//--1/64 duty
    oledSendCmd(0xD3);//-set display offset        Shift Mapping RAM Counter (0x00~0x3F)
    oledSendCmd(0x00);//-not offset
    oledSendCmd(0xd5);//--set display clock divide ratio/oscillator frequency
    oledSendCmd(0x80);//--set divide ratio, Set Clock as 100 Frames/Sec
    oledSendCmd(0xD9);//--set pre-charge period
    oledSendCmd(0xF1);//Set Pre-Charge as 15 Clocks & Discharge as 1 Clock
    oledSendCmd(0xDA);//--set com pins hardware configuration
    oledSendCmd(0x12);
```

```
        oledSendCmd(0xDB);//--set vcomh
        oledSendCmd(0x40);//Set VCOM Deselect Level
        oledSendCmd(0x20);//-Set Page Addressing Mode (0x00/0x01/0x02)
        oledSendCmd(0x02);//
        oledSendCmd(0x8D);//--set Charge Pump enable/disable
        oledSendCmd(0x14);//--set(0x10) disable
        oledSendCmd(0xA4);// Disable Entire Display On (0xa4/0xa5)
        oledSendCmd(0xA6);// Disable Inverse Display On (0xa6/a7)
        oledSendCmd(0xAF); /*display ON*/
}


//OLED 屏所接的 I2C 控制器初始化
static void oledI2cInit()
{
    IoTGpioInit(OLED_SDA);
    IoTGpioInit(OLED_SCL);
    //配置 IO 口由 I2C 控制器使用
    hi_io_set_func(OLED_SDA, OLED_SDAFUNC);
    hi_io_set_func(OLED_SCL, OLED_SCLFUNC);
    //配置上拉
    hi_io_set_pull(OLED_SDA, HI_IO_PULL_UP);
    hi_io_set_pull(OLED_SCL, HI_IO_PULL_UP);
    //初始化第 0 个 I2C 控制器
    IoTI2cInit(0, 400000);          //I2C 传输率为 400kHz
}
void IoTOledInit()
{
    printf("IoTOled Init\n");
    oledI2cInit();                  //初始化 I2C 控制器
    oledInit();                     //OLED 屏初始化
}

SYS_RUN(IoTOledInit);
```

修改 BUILD.gn 编译文件，加入 IoTOled.c 源文件的编译。烧录并复位操作后，因程序仅仅初始化 SSD1306，尚未传输像素数据，所以 OLED 屏会出现花屏的情况，如图 3-13 所示。

图 3-13　OLED 屏初始化

出现花屏的情况也表示 OLED 屏初始化正常，随后，在源代码中添加了一个用于清屏的功能函数：

```
//清屏，让整个屏幕成为黑色
void oledClear(void)
{
    u8 i,n;
    for(i=0; i<8; i++)
    {
        oledSendCmd(0xb0+i);      //设置页地址（0~7）
        oledSendCmd(0x00);        //设置显示位置：列低地址
        oledSendCmd(0x10);        //设置显示位置：列高地址
        for(n=0; n<128; n++)
            oledSendData(0);      //0 表示黑色，  0xff 表示相反色
    } //更新显示
}
```

在程序中调用此函数后，整个屏幕变成黑色。若要使屏幕显示全蓝色，则需将传输的像素数据从 0 改为 0xff。

3. OLED 屏图片显示

由于图片中的每个像素通常由红、绿、蓝三种颜色的数据组成，而 OLED 屏的每个像素点则仅支持 0 或 1 的二进制数据，因此要借助特定工具将图像转换为二进制像素数据，以便在 OLED 屏上正确显示。首先，在系统上利用“画图”工具，将图片的大小调整至 128 像素×64 像素以内，并保存为 bmp 格式的图片文件。接着，在 PCtoLCD 工具中打开已保存的 bmp 图片文件，并根据需要配置选项，具体设置如图 3-14 所示。

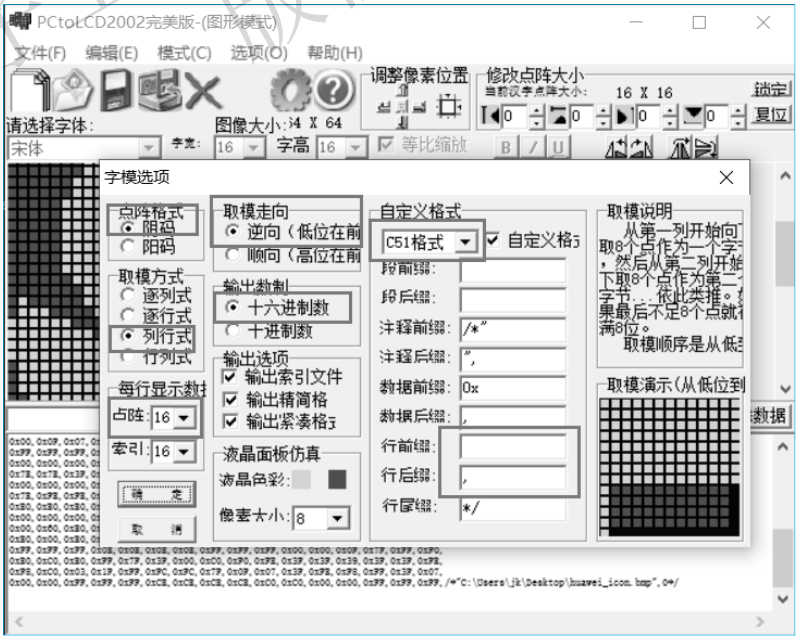


图 3-14　配置选项

生成字模数据后，图像的二进制数据由十六进制数据表示。首先将图像数据复制到源文

件中，并由一个数组存放起来，如图 3-15 所示。

```
static u8 hwLogo[] = { // 64 x 64
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x80,0xE0,0xF8,0xFC,0xFE,0xFE,0xFF,0xFF,0xFF,0xFF,0xFF,0xFE,0xF0,0x00,0x00,
    0x00,0x00,0xF0,0xFE,0xFF,0xFF,0xFF,0xFF,0xFF,0xFE,0xFE,0xFC,0xF8,0xE0,0x80,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x80,0xE0,0xF8,0xFC,0xFE,0xFF,0xFE,0xF8,0xF0,0xE0,0x80,
    0x00,0x0F,0x7F,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x00,
    0x00,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x7F,0x0F,0x00,
    0x80,0xE0,0xF0,0xF8,0xFE,0xFF,0xFE,0xFC,0xF8,0xE0,0x80,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x3F,0x7F,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,
    0xFF,0xFC,0xF8,0xE3,0xCF,0x3F,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x00,
    0x20,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x3F,0xCF,0xE3,0xF8,0xFC,0xFF,
    0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0xFF,0x7F,0x3F,0x00,0x00,0x00,0x00,0x00,
    0x3C,0xFF,0xFE,0xFE,0xFC,0xFC,0xF8,0xF9,0xF3,0xF3,0xE7,0xCF,0xDF,0x9F,0xBF,0x7F,
    0x7F,0xFF,0xFF,0xFF,0xFF,0xFF,0xFC,0xF3,0xE7,0x9F,0x7F,0xFF,0xFF,0xFF,0xFF,0x00,
    0x00,0xFF,0xFF,0xFF,0xFF,0x7F,0x9F,0xE7,0xF3,0xFC,0xFF,0xFF,0xFF,0xFF,0xFF,0x7F,
    0x7F,0xBF,0x9F,0xDF,0xCF,0xE7,0xF3,0xF3,0xF9,0xF8,0xFC,0xFC,0xFE,0xFE,0xFF,0x3C,
    0x00,0x01,0x07,0x0F,0x1F,0x3F,0x3F,0x7F,0x7F,0x7F,0x7F,0xFF,0xFF,0xFF,0xFF,0xFF,
    0xFF,0xFE,0xFC,0xFD,0xFB,0xFB,0xF7,0xE7,0xEF,0xCF,0x1E,0x18,0x03,0x07,0x0F,0x00,
    0x00,0x0F,0x07,0x03,0x18,0x9E,0xCF,0xEF,0xE7,0xF7,0xFB,0xFB,0xFD,0xFC,0xFE,0xFF,
    0xFF,0xFF,0xFF,0xFF,0xFF,0x7F,0x7F,0x7F,0x7F,0x3F,0x3F,0x1F,0x0F,0x07,0x01,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x02,0x0E,0x1E,0x3E,0x3E,0x7E,0x7E,0xFE,0xFE,0x7E,
    0x7E,0x7E,0x3F,0x3F,0x1F,0x0F,0x0F,0x07,0x03,0x03,0x00,0x00,0x00,0x00,0x00,0x00,
    0x00,0x00,0x00,0x00,0x00,0x00,0x03,0x03,0x07,0x0F,0x0F,0x1F,0x3F,0x3F,0x7E,0x7E,
    0x7E,0xFE,0xFE,0x7E,0x7E,0x3E,0x3E,0x1E,0x0E,0x02,0x00,0x00,0x00,0x00,0x00,0x00,
    0xE0,0xE0,0xE0,0x00,0x00,0x00,0x00,0xE0,0xE0,0xE0,0x00,0x00,0xE0,0xE0,0xE0,0x00,
    0x00,0x00,0x00,0xE0,0xE0,0xE0,0x00,0x00,0x00,0x00,0x80,0xE0,0xE0,0xE0,0x80,0x00,
    0x00,0x60,0xE0,0xE0,0x80,0x00,0x00,0x80,0xE0,0xE0,0xE0,0x00,0x00,0x00,0xE0,0xE0,
    0xE0,0x00,0xE0,0xE0,0xE0,0xE0,0xE0,0xE0,0xE0,0xE0,0xE0,0x00,0x00,0xE0,0xE0,0xE0,
    0xFF,0xFF,0xFF,0x0E,0x0E,0x0E,0x0E,0xFF,0xFF,0xFF,0x00,0x00,0x0F,0x7F,0xFF,0xF0,
    0xE0,0xC0,0xE0,0xFF,0x7F,0x3F,0x00,0xC0,0xF0,0xFE,0x3F,0x3F,0x39,0x3F,0x3F,0xFE,
    0xF8,0xC0,0x03,0x1F,0xFF,0xFC,0xFC,0x7F,0x0F,0x07,0x3F,0xFE,0xF8,0xFF,0x3F,0x07,
    0x00,0x00,0xFF,0xFF,0xFF,0xCE,0xCE,0xCE,0xCE,0xC0,0xC0,0x00,0x00,0xFF,0xFF,0xFF
};
```

图 3-15　图像数据

接着开发实现设置 OLED 屏刷出像素点的开始坐标，函数代码如下：

```
void oledSetPosition(u8 x, u8 y)
{ //x 表示一行的第几个像素点，y 表示 OLED 屏的第几行(注意此行为物理上的 8 行像素)
    oledSendCmd(0xb0+y);
    oledSendCmd(((x & 0xf0)>>4)|0x10);
    oledSendCmd(x & 0x0f);
}
```

然后开发将图像数据传输至 SSD1306 驱动 IC，让其在 OLED 屏上显示图像的功能函数代码如下：

```
void oledShowBmp(u8 x, u8 y, u8 bmpW, u8 bmpH, u8 *bmpData)
{ //OLED 每行 8 列，也就是 OLED 屏的一行相当于 8 行像素
    int n = 0;
    for (int r = 0; r < bmpH/8; r++)
    {
        oledSetPosition(x, r+y);                    //设置屏显的始坐标
        for (int c = 0; c < bmpW; c++)
        {
            oledSendData(bmpData[n++]); //每次发一列的 8 个像素数据
        }
    }
}
```

完成后在 IoTOledInit 函数中增加调用代码如下：

```
//在 OLED 屏上的 32:0 坐标开始显示华为 Logo，图像宽与高均是 64 像素
oledShowBmp(32, 0, 64, 64, hwLogo);
```

程序编译、烧录与开发板复位后，OLED 屏显示的图像如图 3-16 所示。

图 3-16　OLED 屏显示的图像

3.4　综合项目实践：带屏显的温湿度计

温湿度计是一种用于测量和监测环境中温度和湿度的仪器，它在多个领域中都发挥着重要作用。在家庭环境中，温湿度计可帮助用户了解当前的温湿度状态，从而调节空调或暖气等设备，以便于营造更加舒适、环保的生活环境；冷库中储存的食物对温湿度的要求更加敏感，需要使用温湿度计进行监控，以防止食物因温湿度变化而变质损坏。

本项目基于本章所学的内容来实现在 OLED 屏上显示 DHT11 传感器温湿度数据的功能。

3.4.1　OLED 屏显示数字

在 OLED 屏上显示一个字符如同显示一张小图片一样，字符的像素点阵数据记录每个像素点是否要显示出来，如“W”字符的 8 像素×6 像素点阵图 3-17 所示。

点阵数据第 0 列的数据为 0x00，第 1 列的数据为 0x3F，依次对应。同理，操作系统不同的字体在字库中也是仅存储每个字符的像素点阵二进制数据，而像素点的颜色信息则不包含在字库中。

因温湿度值是两个数字类型的数据，所以需要数字字符的像素点阵数据。打开 PCtoLCD 工具后，在菜单栏中单击“模式”，然后单击“字符模式”，接着单击菜单栏中的“选项”，打开“字模选项”对话框，如图 3-18 所示。

点阵数据: 0x00, 0x3F, 0x40, 0x38, 0x40, 0x3F

0	1	0	0	0	1	
0	1	0	0	0	1	
0	1	0	0	0	1	
0	1	0	1	0	1	
0	1	0	1	0	1	
0	1	0	1	0	1	
0	0	1	0	1	0	
0	0	0	0	0	0	

图 3-17　“W”字符的 8 像素×6 像素点阵图

图 3-18　“字模选项”对话框

选项配置完成后，在输入框中输入“0123456789.”，生成字模 16 像素×8 像素大小的像素点阵数据，如图 3-19 所示。

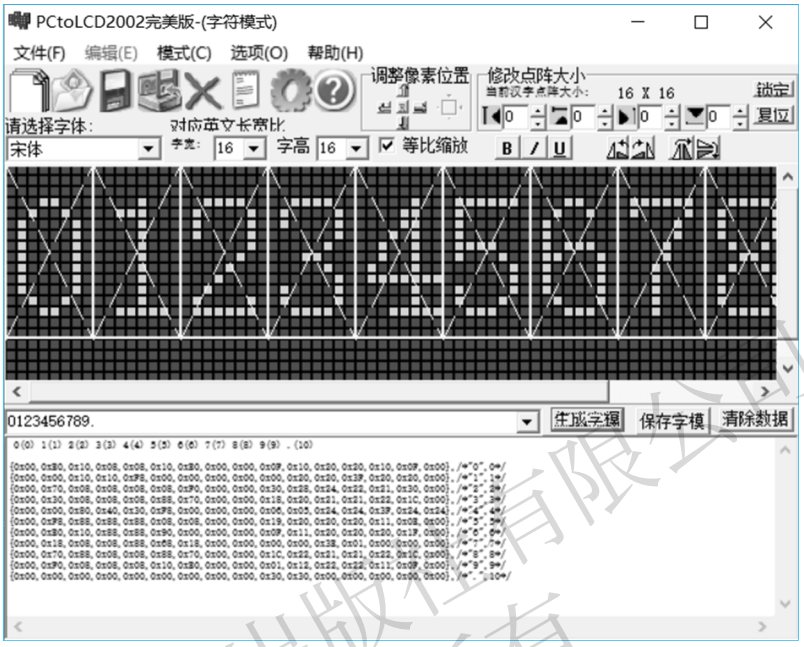


图 3-19　像素点阵数据

然后将字符的像素点阵数据复制到 IoTOled.c 源文件中，并由一个二维数组存放起来，如图 3-20 所示。

```
static u8 numberData[][16] = { //每个数字与英文字母像素: 16 x 8 共128个像素点, 16个字节
    {0x00,0xE0,0x10,0x08,0x08,0x10,0xE0,0x00,0x00,0x0F,0x10,0x20,0x20,0x10,0x0F,0x00},/*"0",0*/
    {0x00,0x00,0x10,0x10,0xF8,0x00,0x00,0x00,0x00,0x00,0x20,0x20,0x3F,0x20,0x20,0x00},/*"1",1*/
    {0x00,0x70,0x08,0x08,0x08,0x08,0xF0,0x00,0x00,0x30,0x28,0x24,0x22,0x21,0x30,0x00},/*"2",2*/
    {0x00,0x30,0x08,0x08,0x08,0x88,0x70,0x00,0x00,0x18,0x20,0x21,0x21,0x22,0x1C,0x00},/*"3",3*/
    {0x00,0x00,0x80,0x40,0x30,0xF8,0x00,0x00,0x00,0x06,0x05,0x24,0x24,0x3F,0x24,0x24},/*"4",4*/
    {0x00,0xF8,0x88,0x88,0x88,0x08,0x08,0x00,0x00,0x19,0x20,0x20,0x20,0x11,0x0E,0x00},/*"5",5*/
    {0x00,0xE0,0x10,0x88,0x88,0x90,0x00,0x00,0x00,0x0F,0x11,0x20,0x20,0x20,0x1F,0x00},/*"6",6*/
    {0x00,0x18,0x08,0x08,0x88,0x68,0x18,0x00,0x00,0x00,0x00,0x3E,0x01,0x00,0x00,0x00},/*"7",7*/
    {0x00,0x70,0x88,0x08,0x08,0x88,0x70,0x00,0x00,0x1C,0x22,0x21,0x21,0x22,0x1C,0x00},/*"8",8*/
    {0x00,0xF0,0x08,0x08,0x08,0x10,0xE0,0x00,0x00,0x01,0x12,0x22,0x22,0x11,0x0F,0x00},/*"9",9*/
    {0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x30,0x30,0x00,0x00,0x00,0x00,0x00},/*".",10*/
};
```

图 3-20　字符数组

在源文件中实现让 OLED 屏显示数字字符的功能函数，代码如下：

```
//在 OLED 屏上显示浮点数， x、y 表示屏显的始坐标， n 表示要显示的浮点数
void oledShowFloat(u8 x, u8 y, float n)
{
    u8 data[20];
    sprintf(data, "%.1f", n);//转换成一位小数的浮点数字符串

    for (int k = 0; k < strlen(data); k++)
```

```
        {
            int n = 0;
            for (int r = 0; r < 2; r++)  //每个字符使用 16 像素×8 像素大小的像素点阵数据，因此只算 2 行
            {
                oledSetPosition(x+k*8, r+y);
                for (int c = 0; c < 8; c++)        //一行 8 个像素点
                {
                    if ('.' == data[k])
                        oledSendData(numberData[10][n++]);
                    else
                        oledSendData(numberData[data[k]-'0'][n++]);
                }
            }
        }
    }
```

函数功能完成后，可以用 IoTOledInit 函数增加调用代码进行测试，正常情况下可以在指定的屏幕行列位置上显示数字。

3.4.2　OLED 屏显示中文字符

为了更加方便用户区分屏上显示的湿度与温度数据，将在屏上加入中文字符的说明。首先在 PCtoLCD 工具上生成中文字符的 16 像素×16 像素大小的像素点阵数据，如图 3-21 所示。

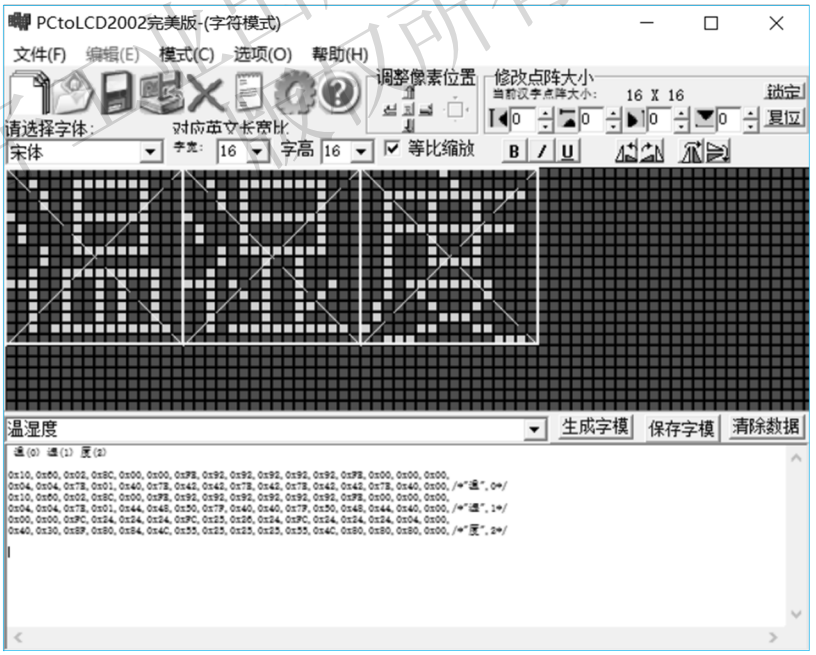


图 3-21　中文字符数据

将中文字符的点阵数据复制到 IoTOled.c 源文件中，修改后由一个二维数组存放起来，如图 3-22 所示。

```
static u8 charData[][32] = { //每个中文像素：16x16 共256个像素点，32字节
    {0x10,0x60,0x02,0x8C,0x00,0x00,0xFE,0x92,0x92,0x92,0x92,0x92,0xFE,0x00,0x00,0x00,
     0x04,0x04,0x7E,0x01,0x40,0x7E,0x42,0x42,0x7E,0x42,0x7E,0x42,0x42,0x7E,0x40,0x00},/*"温",0*/
    {0x10,0x60,0x02,0x8C,0x00,0xFE,0x92,0x92,0x92,0x92,0x92,0x92,0xFE,0x00,0x00,0x00,
     0x04,0x04,0x7E,0x01,0x44,0x48,0x50,0x7F,0x40,0x40,0x7F,0x50,0x48,0x44,0x40,0x00},/*"湿",1*/
    {0x00,0x00,0xFC,0x24,0x24,0x24,0xFC,0x25,0x26,0x24,0xFC,0x24,0x24,0x24,0x04,0x00,
     0x40,0x30,0x8F,0x80,0x84,0x4C,0x55,0x25,0x25,0x25,0x55,0x4C,0x80,0x80,0x80,0x00},/*"度",2*/
};
```

图 3-22　中文字符点阵数组

然后在源文件中开发实现 OLED 屏显示中文字符的功能函数，代码如下：

```
//在 OLED 屏上显示中文字符，x、y 表示屏显的始坐标，index 表示第几个中文字符
void oledShowChinese(u8 x, u8 y, u8 index)
{
    int n = 0;
    for (int r = 0; r < 2; r++)                    //16 列，OLED 屏 2 行
    {
        oledSetPosition(x, r+y);                   //设置屏显的始坐标
        for (int c = 0; c < 16; c++)               //每行有 16 列
        {
            oledSendData(charData[index][n++]);    //每次发一列的 8 个像素数据
        }
    }
}
```

函数完成后，可调用测试，如在 OLED 屏的 20：4 坐标上开始显示“温度”字符的代码如下：

```
oledShowChinese(10, 4, 0);
oledShowChinese(26, 4, 2);
```

3.4.3　主程序编程

本节旨在 OLED 屏上显示 DHT11 传感器所获取的温度与湿度数据。由于 OLED 屏已接入 3.3V 电源接口，因此 DHT11 的电源引脚需通过公对母的杜邦线连接到开发板，确保杜邦线的公头与开发板的 3.3V 接口底座插孔正确对接。传感器接入开发板后，将 3.2.3 节的 DHT11 温湿度传感器驱动源文件 mydht11Thread.c 复制到与 IoTOled.c 源文件相同的目录中，并修改 mydht11Thread.c，使其包含 IoTOled.h 的头文件，并在接收到传感器数据后在屏上画出温湿度值，mydht11Thread.c 源文件修改的内容如下：

```
  #include "IoTOled.h"
static void *threadFunc(void *arg)
{
    int v;

    sleep(3);                    //在 OLED 屏上显示 3s logo
    oledClear();                 //清屏

    oledShowChinese(10, 1, 0);   //显示"温"
    oledShowChinese(27, 1, 2);   //显示"度"
    oledShowFloat(70, 1, 0.0f);  //显示温度值
```

```
        oledShowChinese(10, 5, 1);  //显示"湿"
        oledShowChinese(27, 5, 2);  //显示"度"
        oledShowFloat(70, 5, 0.0f)   //显示湿度值


        while (1)
        {
            sleep(3);
            dht11Start();
            //检查 DHT11 的响应信号，获取模块 DOUT 输出的低电平
            IoTGpioGetInputVal(DHT11IO, &v);
            if (v)
            { //如没有接收到 DHT11 输出的低电平响应信号，则退出循环
                printf("no dht11 found\n");
                continue;
            }
            n = 0;
            //注册 GPIO 中断
            IoTGpioRegisterIsrFunc(DHT11IO, IOT_INT_TYPE_EDGE, IOT_GPIO_EDGE_FALL_LEVEL_
LOW, dht11Isr, NULL);

            //信号量上锁，使当前线程进入休眠状态，直到信号量被解锁为止
            sem_wait(&semLock);
            //线程被唤醒后，输出获取的湿度整数、湿度小数、温度整数、温度小数、校验和
            for (int i = 0; i < 5; i++)
                printf("%d ", vals[i]);
            printf("\n");
            IoTGpioUnregisterIsrFunc(DHT11IO);   //释放 GPIO 的中断功能

            oledShowFloat(70, 1, vals[2]);              //显示温度值
            oledShowFloat(70, 5, vals[0]);              //显示湿度值
        }
        return NULL;
    }
```

接着修改或创建目录下的 BUILD.gn 配置文件，编译两个源文件的合并编译，BUILD.gn 编译文件中的具体内容如下：

```
    static_library("MyTest") {
        sources = [
            "mydht11Thread.c",
            "IoTOled.c",
        ]

        include_dirs = [
            "./",
            "//utils/native/lite/include",
            "//kernel/liteos_m/kal/cmsis",
            "//base/iot_hardware/peripheral/interfaces/kits",
            "//device/hisilicon/hispark_pegasus/sdk_liteos/platform/os/Huawei_LiteOS/components/lib/libc/
musl/include",
        ]
```

```
}
```

程序执行后，OLED 屏显示温湿度数据的效果如图 3-23 所示。

图 3-23　OLED 屏显示温湿度数据的效果

3.4.4　项目总结与拓展

本次项目通过综合运用多线程、定时器、I2C 与中断等开发技术，实现了在 OLED 屏上显示产品图像的功能，并以字符的形式显示当前环境的温度与湿度数据。通过本次的项目实践，学生可以体会到通过操作像素点的方式实现页面效果是一份非常麻烦的工作，因此在资源充足的嵌入式系统开发中要学会移植图形处理库，从而提高界面设计的开发效率。

知识点总结

（1）线程是操作系统能够进行任务调度的最小单位。

（2）osTimer 定时器是基于一个 tick 硬件定时器封装出来的软件定时器。

（3）I2C 接口由两根导线组成：一根为数据线 SDA，另一根为时钟线 SCL。时钟线提供周期信号，每个周期信号表示数据线上传输的一位数据；数据线在每个时钟周期信号中用高电平表示二进制数值 1，低电平表示二进制数值 0。

（4）OLED 屏以一个发光二极管实现一个像素点的显示。

思考与练习

一、判断题

（1）同属一个程序的多个线程可直接访问程序的全局变量。（　　）

（2）每个线程都有自己的栈空间。（　　）

（3）I2C 是串行传输的接口。（　　）

（4）在 OLED 屏驱动中，驱动 IC 不是必须的。（　　）

（5）当信号量没有资源时，则上锁的线程或程序会进入休眠状态。（　　）

二、单选题

（1）I2C 接口有（　　）个引脚。

A. 1　　B. 2　　C. 3　　D. 4

（2）I2C 接口中的 SCL 引脚的作用是（　　）。

A. 传输数据　　B. 传输时钟信号　　C. 接收数据　　D. 使能设备

（3）本章的 OLED 屏模块使用了（　　）接口。

A. PWM　　B. 多线程　　C. I2C　　D. SPI

（4）属于 I2C 接口的引脚是（　　）。

A. CS　　B. DATA　　C. SDA　　D. SPICLK

三、简述题

请简述 OLED 屏的驱动步骤过程。