第 3 章 OpenHarmony 轻量系统系统开发

本章通过 6 个案例详细讲解 OpenHarmony 轻量系统中任务、互斥锁、软件定时器、中断处理、内存的开发技术。

3.1 任务

任务是竞争系统资源的最小运行单元。任务可以使用或等待 CPU、使用内存空间等系统资源，并独立于其他任务运行。任务模块可以向用户提供多个任务，实现任务之间的切换和通信，帮助用户管理业务程序流程。

(1) 支持多任务，一个任务表示一个线程。

(2) 任务是抢占式调度机制，同时支持时间片轮转调度方式。

(3) 高优先级的任务可打断低优先级任务，低优先级任务必须在高优先级任务阻塞或结束后才可以得到调度。

(4) 由于系统自身任务需要及时调度，建议用户使用的任务优先级范围是[10,30]。应用级任务建议使用低于系统级任务的优先级。

3.1.1 案例 26：计时器

本案例为计时器，实现以秒为单位的计时功能。案例通过调用任务 sleep 函数实现秒数的累加，最后格式化输出数据。

开发步骤如下：

(1) 创建本章源码存放目录 chapter_03。

(2) 在 chapter_03 中创建工程目录 case26-timer。

(3) 在 case26-timer 中创建源码文件 timer_demo.c。

(4) 实现计时器功能，代码如下：

```
//applications/sample/wifi-iot/app/ohos_50/chapter_03/case26-timer/timer_demo.c
```

```
#include <stdio.h>
#include <unistd.h>
#include <ohos_init.h>
int i = 0;          //保存时间
//任务回调执行函数
static void entry(void )
{
   while (i<3600)
   {
      //函数声明在头文件 unistd.h 中,实现任务秒级休眠功能
      sleep(1);
      i++;
      printf("\r%02d:%02d", i/60,i%60);
   }
}
//初始化模块入口函数
APP_FEATURE_INIT(entry);
```

(5) 创建并编写模块,构建脚本 BUILD.gn,代码如下:

```
//applications/sample/wifi-iot/app/ohos_50/chapter_03/case26-timer/BUILD.gn
static_library("ch_03_timer") {
   sources = [
      "timer_demo.c",
   ]
}
```

(6) 将应用模块 ch_03_timer 配置到应用子系统,代码如下:

```
//applications/sample/wifi-iot/app/BUILD.gn
import("//build/lite/config/component/lite_component.gni")
lite_component("app") {
   features = [
      "ohos_50/chapter_03/case26-timer:ch_03_timer",
   ]
}
```

(7) 测试:编译应用模块,将固件烧写到开发板,运行 IPOP 终端工具,以便与开发板相连,复位开发板。观察 IPOP 终端工具的打印信息,如图 3-1 所示。

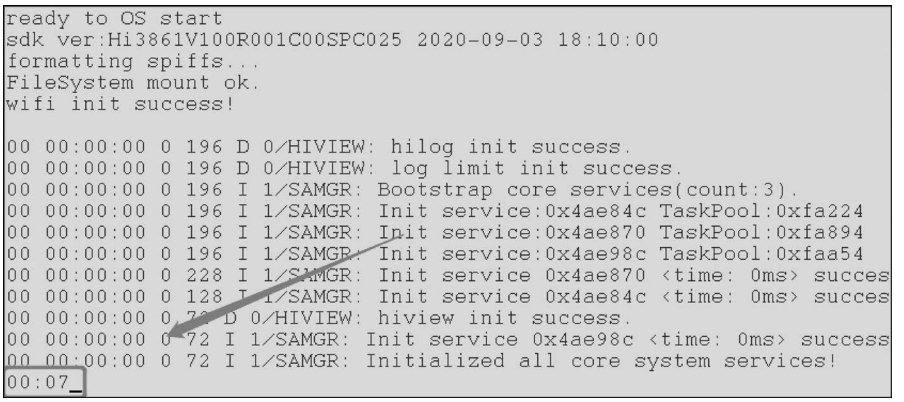

图 3-1　计时器的运行效果

3.1.2　案例 27：自动售票系统 V1.0

本案例为自动售票系统的 1.0 版，实现多个任务操作共享数据的功能，即多个终端同时具有售票功能。

案例通过函数 osThreadNew 创建了 3 个任务，模拟 3 个售票终端同时售卖同一种票。

开发步骤如下：

(1) 在 chapter_03 中创建工程目录 case27-atm_v1.0。

(2) 在 case27-atm_v1.0 中创建源码文件 atm_v1.0_demo.c。

(3) 实现自动售票机交替售票功能，代码如下：

```
//applications/sample/wifi-iot/app/ohos_50/chapter_03/case27-atm_v1.0/atm_v1.0_demo.c
#include <stdio.h>
#include <unistd.h>
#include <ohos_init.h>
#include "cmsis_os2.h"
int i = 20;                    //剩余票数
//任务回调执行函数
static void task_function(void *task_name)
{
    while (i > 0)
    {
        //函数声明在头文件 unistd.h 中,实现任务微秒级休眠功能
        usleep(20);
        //售票成功,剩余票数减 1
        i--;
        printf("%s Issue 1 ticket ,remaining votes : %d!\r\n", (char *)task_name, i);
    }
}
```

```
//创建并运行自动售票机
void task_create(char * task_name, osThreadFunc_t func)
{
   //创建任务结构体 attr,配置任务属性
   osThreadAttr_t attr;
   //配置任务名称
   attr.name = task_name;
   attr.attr_bits = 0U;
   attr.cb_mem = NULL;
   attr.cb_size = 0U;
   attr.stack_mem = NULL;
   attr.stack_size = 1024; //将栈配置为 1KB
   attr.priority = osPriorityNormal;
   //函数 osThreadNew 声明在头文件 cmsis_os2.h 中,实现任务的创建和运行功能
   if (osThreadNew(func, (void * )task_name, &attr) == NULL)
   {
      printf(" % s create failed!\n", task_name);
   }
}
void entry(void)
{
   //函数声明在头文件 unistd.h 中,实现任务秒级休眠功能
   sleep(1);
   //创建并运行 3 个自动售票机(ATM)
   task_create("ATM_1", task_function);
   task_create("ATM_2", task_function);
   task_create("ATM_3", task_function);
}
//初始化模块入口函数
APP_FEATURE_INIT(entry);
```

(4) 创建并编写模块,构建脚本 BUILD.gn,代码如下:

```
//applications/sample/wifi - iot/app/ohos_50/chapter_02/case27 - atm_v1.0/BUILD.gn
static_library("ch_03_atm_v1.0") {
   sources = [
      "atm_v1.0_demo.c",
   ]
   # 配置头文件 cmsis_os2.h 的路径
   include_dirs = [
      "//kernel/liteos_m/components/cmsis/2.0",
   ]
}
```

(5) 将应用模块配置到应用子系统,代码如下:

```
//applications/sample/wifi-iot/app/BUILD.gn
import("//build/lite/config/component/lite_component.gni")

lite_component("app") {
   features = [
      "ohos_50/chapter_03/case27-atm_v1.0:ch_03_atm_v1.0",
   ]
}
```

（6）测试：编译应用模块，将固件烧写到开发板，运行 IPOP 终端工具，以便与开发板相连，复位开发板。观察 IPOP 终端工具的打印信息，如图 3-2 所示。

```
00 00:00:00 0 8 I 1/SAMGR: Init service 0x4ae808 <time: 0ms> success!
00 00:00:00 0 8 I 1/SAMGR: Initialized all core system services!
ATM_1 Issue 1 ticket ,remaining votes :19!
ATM_2 Issue 1 ticket ,remaining votes :18!
ATM_3 Issue 1 ticket ,remaining votes :17!
ATM_1 Issue 1 ticket ,remaining votes :16!
ATM_2 Issue 1 ticket ,remaining votes :15!
ATM_3 Issue 1 ticket ,remaining votes :14!
ATM_1 Issue 1 ticket ,remaining votes :13!
ATM_2 Issue 1 ticket ,remaining votes :12!
ATM_3 Issue 1 ticket ,remaining votes :11!
ATM_1 Issue 1 ticket ,remaining votes :10!
ATM_2 Issue 1 ticket ,remaining votes :9!
ATM_3 Issue 1 ticket ,remaining votes :8!
ATM_1 Issue 1 ticket ,remaining votes :7!
ATM_2 Issue 1 ticket ,remaining votes :6!
ATM_3 Issue 1 ticket ,remaining votes :5!
ATM_1 Issue 1 ticket ,remaining votes :4!
ATM_2 Issue 1 ticket ,remaining votes :3!
ATM_3 Issue 1 ticket ,remaining votes :2!
ATM_1 Issue 1 ticket ,remaining votes :1!
ATM_2 Issue 1 ticket ,remaining votes :0!
ATM_3 Issue 1 ticket ,remaining votes :-1!
ATM_1 Issue 1 ticket ,remaining votes :-2!
```

图 3-2　自动售票机 V1.0 的运行效果

3.2　案例 28：自动售票系统 V2.0

本案例是案例 27 的改进版，在案例 27 的基础上使用了互斥锁，以便保证数据被正确操作。涉及的函数为 osMutexAcquire 和 osMutexRelease，在保存数据时使用函数 osMutexAcquire 进行上锁以保证只有当前任务可以操作该数据，操作完成后使用函数 osMutexRelease 释放锁，以便其他任务可以对该数据进行操作。

互斥锁用来保证共享数据操作的完整性。它可以保证共享数据在任一时刻，只能有一个线程执行操作，其他线程处于等待状态。

开发步骤如下：

（1）在本章源码存放目录 chapter_03 中创建应用模块工程目录 case28-atm_v2.0。

（2）在应用模块工程目录 case28-atm_v2.0 中创建应用模块源码文件 atm_v2.0_

demo.c。

（3）实现自动售票机交替售票功能，代码如下：

```
//applications/sample/wifi-iot/app/ohos_50/chapter_03/case28-atm_v2.0/atm_v2.0_demo.c
#include <stdio.h>
#include <unistd.h>
#include <ohos_init.h>
#include "cmsis_os2.h"
osMutexId_t mutex_id;                    //互斥锁 ID
int i = 20;                              //剩余票数
//任务回调执行函数
static void example_mutex_entry(void *task_name)
{
    while (1)
    {
        //函数 osMutexAcquire 声明在头文件 cmsis_os2.h 中,实现获取互斥锁,锁住其他任务功能
        osMutexAcquire(mutex_id, osWaitForever);
        usleep(20);
        if (i <= 0)
        {
            //函数 osMutexRelease 声明在头文件 cmsis_os2.h 中,实现释放互斥锁功能
            osMutexRelease(mutex_id);
            break;
        }

        //售票成功,剩余票数减 1
        i--;
            printf("%s Issue 1 ticket ,remaining votes :%d!\r\n", (char *)task_name, i);
        //函数 osMutexRelease 声明在头文件 cmsis_os2.h 中,实现释放互斥锁功能
        osMutexRelease(mutex_id);
    }
}
//创建并运行自动售票机
void example_ATM_create(char *task_nane, osThreadFunc_t func)
{
    //创建任务结构体 attr,配置任务属性
    osThreadAttr_t attr;
    //配置任务名称
    attr.name = task_nane;
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 1024;              //将栈配置为 1KB
    attr.priority = osPriorityNormal;
```

```
    //创建并运行任务
    if (osThreadNew(func, (void * )task_nane, &attr) == NULL)
    {
        printf(" % s create failed!\n", task_nane);
    }
}
void entry(void)
{
    sleep(1);
    //创建并运行 3 个自动售票机(ATM)
    example_ATM_create("ATM_1", example_mutex_entry);
    example_ATM_create("ATM_2", example_mutex_entry);
    example_ATM_create("ATM_3", example_mutex_entry);
}
//初始化模块入口函数
APP_FEATURE_INIT(entry);
```

(4) 创建并编写模块,构建脚本 BUILD.gn,代码如下:

```
\\//applications/sample/wifi - iot/app/ohos_50/chapter_03/case28 - atm_v2.0/BUILD.gn
static_library("ch_03_atm_v2.0") {
    sources = [
        "atm_v2.0_demo.c",
    ]
    include_dirs = [
        "//kernel/liteos_m/components/cmsis/2.0",
    ]
}
```

(5) 将应用模块配置到应用子系统,代码如下:

```
\\//applications/sample/wifi - iot/app/BUILD.gn
import("//build/lite/config/component/lite_component.gni")
lite_component("app") {
    features = [
        "ohos_50/chapter_03/case28 - atm_v2.0:ch_03_atm_v2.0",
    ]
}
```

(6) 测试:编译应用模块,将固件烧写到开发板,运行 IPOP 终端工具,以便与开发板相连,复位开发板。观察 IPOP 终端工具的打印信息,如图 3-3 所示。

```
00 00:00:00 0 8 D 0/HIVIEW: hiview init success.
00 00:00:00 0 8 I 1/SAMGR: Init service 0x4ae828 <time: 0ms> success!
00 00:00:00 0 8 I 1/SAMGR: Initialized all core system services!
ATM_1 Issue 1 ticket ,remaining votes :19!
ATM_2 Issue 1 ticket ,remaining votes :18!
ATM_3 Issue 1 ticket ,remaining votes :17!
ATM_1 Issue 1 ticket ,remaining votes :16!
ATM_2 Issue 1 ticket ,remaining votes :15!
ATM_3 Issue 1 ticket ,remaining votes :14!
ATM_1 Issue 1 ticket ,remaining votes :13!
ATM_2 Issue 1 ticket ,remaining votes :12!
ATM_3 Issue 1 ticket ,remaining votes :11!
ATM_1 Issue 1 ticket ,remaining votes :10!
ATM_2 Issue 1 ticket ,remaining votes :9!
ATM_3 Issue 1 ticket ,remaining votes :8!
ATM_1 Issue 1 ticket ,remaining votes :7!
ATM_2 Issue 1 ticket ,remaining votes :6!
ATM_3 Issue 1 ticket ,remaining votes :5!
ATM_1 Issue 1 ticket ,remaining votes :4!
ATM_2 Issue 1 ticket ,remaining votes :3!
ATM_3 Issue 1 ticket ,remaining votes :2!
ATM_1 Issue 1 ticket ,remaining votes :1!
ATM_2 Issue 1 ticket ,remaining votes :0!
```

图 3-3　自动售票机 V2.0 的运行效果

3.3　案例 29：软件定时器

本案例为软件定时器，通过函数 osTimerNew 和 osTimerStart 实现软件定时执行指定功能函数的功能。

软件定时器是基于系统 Tick 时钟中断且由软件来模拟的定时器，当经过设定的 Tick 时钟计数值后会触发用户定义的回调函数。定时精度与系统 Tick 时钟的周期有关。

（1）硬件定时器受硬件的限制，数量上不足以满足用户的实际需求，因此为了满足用户需求，需要提供更多的定时器，系统提供软件定时器功能。

（2）软件定时器扩展了定时器的数量，允许创建更多的定时业务。

软件定时器支持的功能如下：

（1）软件定时器创建。

（2）软件定时器启动。

（3）软件定时器停止。

（4）软件定时器删除。

（5）运作机制：

① 软件定时器使用了系统的一个队列和一个任务资源，先进先出。定时时间短的定时器总是比定时时间长的靠近队列头，满足优先被触发的准则。

② 当 Tick 中断到来时，在 Tick 中断处理函数中扫描软件定时器的计时任务，查看是否有定时器超时，如果有，则将超时的定时器记录下来。

③ Tick 中断处理函数结束后，软件定时器任务（优先级为高）被唤醒，在该任务中调用

之前记录下来的定时器的超时回调函数。

（6）软件定时器提供了两类定时器机制。

① 单次触发定时器：在启动后只会触发一次定时器事件，然后定时器自动删除。

② 周期触发定时器：会周期性地触发定时器事件，直到用户手动地停止定时器，否则将永远持续执行。

开发步骤如下：

（1）在本章目录 chapter_03 中创建应用模块工程目录 case29-software_timer。

（2）在应用模块工程目录 case29-software_timer 中创建应用模块源码文件 software_timer_demo.c。

（3）实现软件定时器功能，代码如下：

```
//applications/sample/wifi - iot/app/ohos_50/chapter_03/case29 - software_timer/software_
timer_demo.c
#include "stdio.h"
#include "ohos_init.h"
#include "cmsis_os2.h"
#include <unistd.h>
int g_timercount1 = 0;                    //记录定时器 1 回调函数被执行的次数
int g_timercount2 = 0;                    //记录定时器 2 回调函数被执行的次数
//定时器 1 任务执行回调函数
void timer1_callback(void * arg)
{
    //函数参数必须使用,否则编译器会报变量未使用错误
    arg = arg;
    g_timercount1++;
    printf("g_timercount1 = %d\n", g_timercount1);
}
//定时器 2 任务执行回调函数
void timer2_callback(void * arg)
{
    arg = arg;
    g_timercount2++;
    printf("g_timercount2 = %d\n", g_timercount2);
}
void entry(void)
{
    sleep(1);
    //创建定时器 id 数组
    osTimerId_t timer_id[2];
    //创建定时器回调函数数组
    osTimerFunc_t timer_callback[2] = {timer1_callback,timer2_callback};
    //创建定时器属性结构体数组
    osTimerAttr_t timer_att[2] = {{"timer1",0,NULL,0},{"timer2",0,NULL,0}};
```

```
    //创建定时器回调间隔的 ticks 数组,1tick 的时长由芯片主频决定,如 Hi3861 芯片中 1tick 约
    //等于 10ms
    unsigned int ticks[2] = {300,500};
    for(int i = 0;i < 2;i++){
        //函数 osTimerNew 声明在头文件 cmsis_os2.h 中,实现创建定时器功能
        timer_id[i] = osTimerNew(timer_callback[i],osTimerPeriodic,NULL,timer_att + i);

        //函数 osTimerStart 声明在头文件 cmsis_os2.h 中,实现启用定时器功能
        osTimerStart(timer_id[i],ticks[i]);
    }

    sleep(60);
    //判断定时器是否运行,如果运行,则停止并删除
    for(int i = 0;i < 2;i++){
        //函数 osTimerIsRunning 声明在头文件 cmsis_os2.h 中,用于检查定时器是否运行
        if(osTimerIsRunning(timer_id[i])){
            //函数 osTimerStop 声明在头文件 cmsis_os2.h 中,实现停止定时器功能
                osTimerStop(timer_id[i]);
                //函数 osTimerDelete 声明在头文件 cmsis_os2.h 中,实现删除定时器功能
            osTimerDelete(timer_id[i]);
        }
    }
}
//创建并运行定时器任务
void example_timer_create(void )
{
    //创建 osThreadAttr_t 结构体 attr,用于存放任务参数
    osThreadAttr_t attr;
    //设置任务名称
    attr.name = "timer";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 1024; //将栈配置为 1KB
    attr.priority = osPriorityNormal;
    //创建并运行任务
    if (osThreadNew((osThreadFunc_t)entry, NULL, &attr) == NULL)
    {
        printf("timer task create failed!\n");
    }
}
//初始化模块入口函数
APP_FEATURE_INIT(example_timer_create);
```

（4）创建并编写模块，构建脚本 BUILD.gn，代码如下：

```
//applications/sample/wifi-iot/app/ohos_50/chapter_03/case29-software_timer/BUILD.gn
static_library("ch_03_software_timer") {
    sources = [
        "software_timer_demo.c",
    ]
    include_dirs = [
        "//kernel/liteos_m/components/cmsis/2.0",
    ]
}
```

（5）将应用模块配置到应用子系统，代码如下：

```
//applications/sample/wifi-iot/app/BUILD.gn
import("//build/lite/config/component/lite_component.gni")
lite_component("app") {
    features = [
        "ohos_50/chapter_03/case29-software_timer:ch_03_software_timer",
    ]
}
```

（6）测试：编译应用模块，将固件烧写到开发板，运行 IPOP 终端工具，以便与开发板相连，复位开发板。观察 IPOP 终端工具的打印信息，如图 3-4 所示。

```
00 00:00:00 0 8 I 1/SAMGR: Initialized all core system services!
00 00:00:00 0 64 I 1/SAMGR: Bootstrap system and application services
00 00:00:00 0 64 I 1/SAMGR: Initialized all system and application se
00 00:00:00 0 64 I 1/SAMGR: Bootstrap dynamic registered services(cou
g_timercount1=1
g_timercount2=1
g_timercount1=2
g_timercount1=3
g_timercount2=2
g_timercount1=4
g_timercount2=3
g_timercount1=5
g_timercount1=6
g_timercount2=4
g_timercount1=7
g_timercount1=8
g_timercount2=5
g_timercount1=9
g_timercount2=6
g_timercount1=10
g_timercount1=11
g_timercount2=7
g_timercount1=12
g_timercount1=13
g_timercount2=8
```

图 3-4　软件定时器的运行效果

3.4 案例30：按键中断处理

本案例为按键中断处理，通过函数 GpioRegisterIsrFunc 和 GpioUnregisterIsrFunc 实现按键 S2 中断注册与注销功能。

中断是指 CPU 暂停执行当前程序，转而执行新程序的过程。与中断相关的硬件可以划分为 3 类。

（1）设备：发起中断的源，当设备需要请求 CPU 时，产生一个中断信号，该信号连接至中断控制器。

（2）中断控制器：接收中断输入并上报给 CPU。可以设置中断源的优先级、触发方式、打开和关闭等操作。

（3）CPU：判断和执行中断任务。

与中断相关的名词解释如下。

（1）中断号：每个中断请求信号都会有特定的标志，使计算机能够判断是哪个设备提出的中断请求，这个标志就是中断号。

（2）中断请求："紧急事件"需向 CPU 提出申请（发一个电脉冲信号），要求中断，以及要求 CPU 暂停当前执行的任务，转而处理该"紧急事件"，这一申请过程称为中断申请。

（3）中断优先级：为使系统能够及时响应并处理所有中断，系统根据中断事件的重要性和紧迫程度，将中断源分为若干个级别，称作中断优先级。系统中所有的中断源优先级相同，不支持中断嵌套或抢占。

（4）中断处理程序：当外设产生中断请求后，CPU 暂停当前的任务，转而响应中断申请，即执行中断处理程序。

（5）中断触发：中断源发出并送给 CPU 控制信号，将接口卡上的中断触发器置"1"，表明该中断源产生了中断，要求 CPU 去响应该中断，CPU 暂停当前任务，执行相应的中断处理程序。

（6）中断触发类型：外部中断申请通过一个物理信号发送到 CPU，可以是电平触发或边沿触发。

（7）中断向量：中断服务程序的入口地址。

（8）中断向量表：存储中断向量的存储区，中断向量与中断号对应，中断向量在中断向量表中按照中断号顺序存储。

S2 位于主控板 Type-C 接口的左侧，如图 3-5 所示。

S2 按键的电路原理图如图 3-6 所示。

S2 按键的处理流程如图 3-7 所示。

S2 按键的电平变化如图 3-8 所示。

开发步骤如下：

（1）在本章目录 chapter_03 中创建应用模块工程目录 case30-isr。

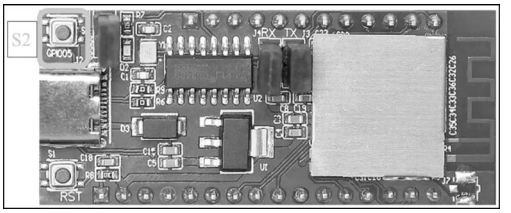

图 3-5　S2 按键

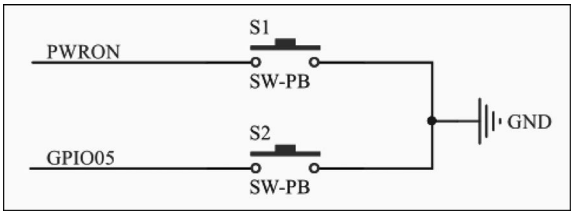

图 3-6　S2 按键的电路原理图

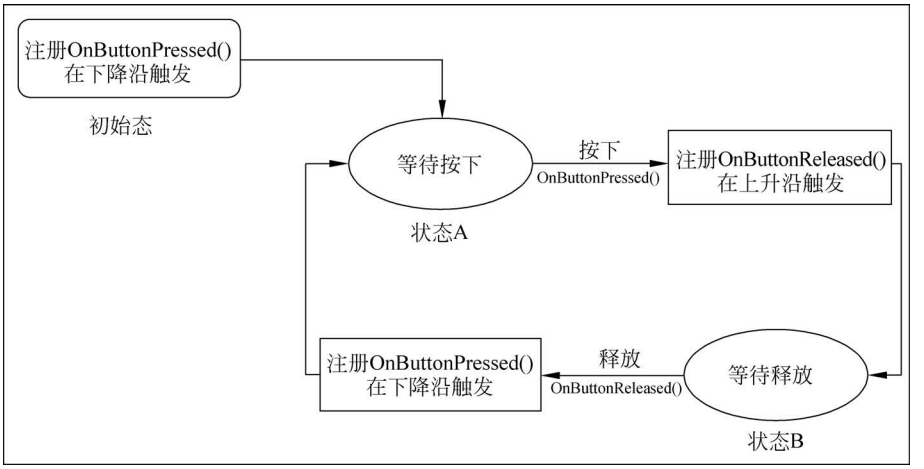

图 3-7　S2 按键的处理流程

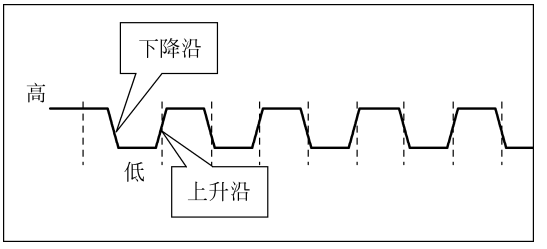

图 3-8　S2 按键的电平变化

（2）在应用模块工程目录 case30-isr 中创建应用模块源码文件 isr.c。

（3）引用必要的头文件，代码如下：

```
//applications/sample/wifi-iot/app/ohos_50/chapter_02/case30-isr/isr.c
#include <stdio.h>
#include <unistd.h>
#include <ohos_init.h>
#include "cmsis_os2.h"
#include "wifiiot_gpio.h"
#include "wifiiot_gpio_ex.h"
```

（4）在 isr.c 文件中创建按键中断处理回调函数 on_button_pressed，实现中断处理功能，代码如下：

```
//applications/sample/wifi-iot/app/ohos_50/chapter_02/case30-isr/isr.c
//按键中断处理回调函数
static void on_button_pressed(char *arg)
{
    (void)arg;
    //函数声明在头文件 wifiiot_gpio.h 中，实现给 GPIO5 引脚注销中断功能
    GpioUnregisterIsrFunc(WIFI_IOT_IO_NAME_GPIO_5);
    printf("key is pressed\r\n");
    sleep(1);
    //函数声明在头文件 wifiiot_gpio.h 中，实现给 GPIO5 引脚注册下降沿触发中断
GpioRegisterIsrFunc(WIFI_IOT_IO_NAME_GPIO_5, WIFI_IOT_INT_TYPE_EDGE, WIFI_IOT_GPIO_EDGE_
FALL_LEVEL_LOW,on_button_pressed, NULL);
}
```

（5）创建主任务函数 example_isr_entry，实现 GPIO 引脚的初始化、功能设置、IO 方向设置、注册中断功能，代码如下：

```
//applications/sample/wifi-iot/app/ohos_50/chapter_02/case30-isr/isr.c
static void example_isr_entry(void)
{
    //函数声明在头文件 wifiiot_gpio.h 中，实现初始化 GPIO 引脚功能
    GpioInit();
    //函数声明在头文件 wifiiot_gpio.h 中，实现给 GPIO5 引脚设置 GPIO 功能
    IoSetFunc(WIFI_IOT_IO_NAME_GPIO_5, WIFI_IOT_IO_FUNC_GPIO_5_GPIO);
    //函数声明在头文件 wifiiot_gpio.h 中，实现将 GPIO5 引脚 IO 方向设置为输入
    GpioSetDir(WIFI_IOT_IO_NAME_GPIO_5, WIFI_IOT_GPIO_DIR_IN);
    //函数声明在头文件 wifiiot_gpio_ex.h 中，实现给 GPIO5 引脚默认为 3.3V 高电平
    IoSetPull(WIFI_IOT_IO_NAME_GPIO_5, WIFI_IOT_IO_PULL_UP);
    //函数声明在头文件 wifiiot_gpio.h 中，实现给 GPIO5 引脚注册下降沿触发中断
    GpioRegisterIsrFunc(WIFI_IOT_IO_NAME_GPIO_5, WIFI_IOT_INT_TYPE_EDGE, WIFI_IOT_GPIO_EDGE_
FALL_LEVEL_LOW,on_button_pressed, NULL);
```

```
        //主任务处于等待状态
    while (1)
    {
        usleep(1000);
    }
}
```

（6）创建函数 example_isr_create，在函数中创建任务，将任务执行函数设置为 example_isr_entry，将模块入口函数初始化为 example_isr_create，代码如下：

```
//applications/sample/wifi - iot/app/ohos_50/chapter_02/case30 - isr/isr.c
static void example_isr_create(void)
{
    osThreadAttr_t attr;
    attr.name = "isr_task";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 1024;
    attr.priority = osPriorityNormal;
    if (osThreadNew((osThreadFunc_t)example_isr_entry, NULL, &attr) == NULL)
    {
        printf("[example_isr_create] Failed to create isr_task!\n");
    }
}

APP_FEATURE_INIT(example_isr_create);
```

（7）创建并编写模块，构建脚本 BUILD.gn，代码如下：

```
//applications/sample/wifi - iot/app/ohos_50/chapter_02/case30 - isr/BUILD.gn
static_library("ch_03_isr") {
    sources = [
        "isr.c",
    ]
    include_dirs = [
        "//kernel/liteos_m/components/cmsis/2.0",
        "//base/iot_hardware/interfaces/kits/wifiiot_lite",
    ]
}
```

（8）将应用模块配置到应用子系统，代码如下：

```
//applications/sample/wifi - iot/app/BUILD.gn
```

```
import("//build/lite/config/component/lite_component.gni")
lite_component("app") {
   features = [
        "ohos_50/chapter_02/case30 - isr:ch_03_isr",
   ]
}
```

(9) 测试：编译应用模块，将固件烧写到开发板，运行 IPOP 终端工具，以便与开发板相连，复位开发板，然后多次按 S2 按键。观察 IPOP 终端工具的打印信息，如图 3-9 所示。

```
00 00:00:00 0 132 D 0/HIVIEW: log limit init success.
00 00:00:00 0 132 I 1/SAMGR: Bootstrap core services(count:3).
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4ae8bc TaskPool:0xfa1e4
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4ae8e0 TaskPool:0xfa854
00 00:00:00 0 132 I 1/SAMGR: Init service:0x4aea44 TaskPool:0xfaa14
00 00:00:00 0 164 I 1/SAMGR: Init service 0x4ae8e0 <time: 0ms> succes
00 00:00:00 0 64 I 1/SAMGR: Init service 0x4ae8bc <time: 0ms> success
00 00:00:00 0 8 D 0/HIVIEW: hiview init success.
00 00:00:00 0 8 I 1/SAMGR: Init service 0x4aea44 <time: 0ms> success!
00 00:00:00 0 8 I 1/SAMGR: Initialized all core system services!
00 00:00:00 0 64 I 1/SAMGR: Bootstrap system and application services
00 00:00:00 0 64 I 1/SAMGR: Initialized all system and application se
00 00:00:00 0 64 I 1/SAMGR: Bootstrap dynamic registered services(cou
key is pressed
key is pressed
key is pressed
key is pressed
key is pressed
key is pressed
key is pressed
key is pressed
key is pressed
key is pressed
key is pressed
```

图 3-9　按键中断的处理效果

3.5　案例 31：内存申请与释放

本案例为内存申请与释放，实现动态内存申请、操作和释放功能。由于 OpenHarmony 轻量系统支持标准 C 库，所以本案例讲解使用标准 C 库中的函数 malloc 和 free 实现内存的申请与释放功能。

内存管理模块管理系统的内存资源，通过对内存的申请/释放操作来管理用户和 OS 对内存的使用，使内存的利用率和效率最优，最大限度地解决系统的内存碎片问题，其中，OS 的内存管理为动态内存管理，提供内存初始化、分配、释放等功能。动态内存是指在动态内存池中分配用户指定大小的内存块。

(1) 优点：按需分配。

(2) 缺点：内存池中可能出现碎片。

开发步骤如下：

（1）在本章目录 chapter_03 中创建应用模块工程目录 case31-memory。

（2）在应用模块工程目录 case31-memory 中创建应用模块源码文件 memory.c。

（3）引用必要的头文件，代码如下：

```
//applications/sample/wifi-iot/app/ohos_50/chapter_03/case31-memory/memory.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <unistd.h>
#include <ohos_init.h>
#include "cmsis_os2.h"
```

（4）在 memory.c 文件中创建函数 example_mem_entry，调用函数 malloc 和 free 实现内存的申请与释放功能，代码如下：

```
//applications/sample/wifi-iot/app/ohos_50/chapter_03/case31-memory/memory.c
static void example_mem_entry(void)
{
    //申请 100 字节的内存空间
    void * ptr = malloc(100);
    if(ptr == NULL ){
        printf("Malloc failed!\n\r");
        return;
    }
    //以 0 进行内存初始化
    memset(ptr,0,100);
    //将字符串复制到内存
    strcpy(ptr,"hello world!\r\n");
    //打印内存内容
    printf("ptr: %s\r\n",ptr);
    //释放内存，防止内存泄漏
    free(ptr);
    ptr = NULL;
}
```

（5）创建函数 example_mem_create，在函数中创建任务，将任务执行函数设置为 example_mem_entry，将模块入口函数初始化为 example_mem_create，代码如下：

```
//applications/sample/wifi-iot/app/ohos_50/chapter_03/case31-memory/memory.c
static void example_mem_create(void)
```

```
{
    osThreadAttr_t attr;
    sleep(1);
    attr.name = "memory_task";
    attr.attr_bits = 0U;
    attr.cb_mem = NULL;
    attr.cb_size = 0U;
    attr.stack_mem = NULL;
    attr.stack_size = 1024;
    attr.priority = osPriorityNormal;
    if (osThreadNew((osThreadFunc_t)example_mem_entry, NULL, &attr) == NULL)
    {
        printf("[example_mem_create] Failed to create memory task!\n");
    }
}
APP_FEATURE_INIT(example_mem_create);
```

(6) 创建并编写模块,构建脚本 BUILD.gn,代码如下:

```
//applications/sample/wifi-iot/app/ohos_50/chapter_03/case31-memory/BUILD.gn
static_library("ch_03_mem") {
    sources = [
        "memory.c",
    ]
    include_dirs = [
        "//kernel/liteos_m/components/cmsis/2.0",
    ]
}
```

(7) 将应用模块配置到应用子系统,代码如下:

```
//applications/sample/wifi-iot/app/BUILD.gn
import("//build/lite/config/component/lite_component.gni")
lite_component("app") {
    features = [
         "ohos_50/chapter_03/case31-memory:ch_03_mem",
    ]
}
```

(8) 测试:编译应用模块,将固件烧写到开发板,运行 IPOP 终端工具,以便与开发板相

连，复位开发板。观察 IPOP 终端工具的打印信息，如图 3-10 所示。

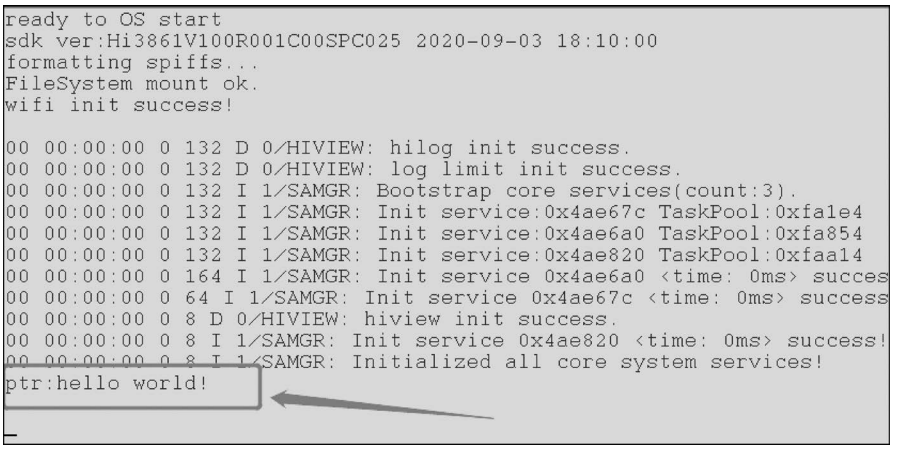

图 3-10　内存申请与释放