第5章 基于Contiki操作系统的基础项目开发

本章介绍Contiki操作系统的基本知识，先分析Contiki操作系统的特点和源代码结构，介绍操作系统的主要数据结构，有进程、事件和etimer机制，并分析三者之间的关系，然后将操作系统通过移植到STM32，并通过任务式开完成了对GPIO控制、多线程、进程间通信、定时器的驱动，最后实现了对LCD的驱动。

5.1 任务35 认识Contiki操作系统

5.1.1 学习目标

- 初步了解Contiki操作系统基本特点；
- 理解Contiki事件驱动和protothread机制。

5.1.2 原理学习

5.1.2.1 Contiki操作系统

Contiki是一个开源的、高度可移植的多任务操作系统，适用于联网嵌入式系统和无线传感器网络，由瑞典计算机科学学院（Swedish Institute of Computer Science）的Adam Dunkels和他的团队开发。Contiki完全采用C语言开发，可移植性非常好，对硬件的要求极低，能够运行在各种类型的微处理器及计算机上。

Contiki适用于存储器资源十分受限的嵌入式单片机系统，是一种开源的操作系统，适用于BSD协议，即可以任意修改和发布，已经应用在许多项目中。Contiki操作系统是基于事件驱动（Event-driven）内核的操作系统，在此内核上，应用程序可以在运行时动态加载，非常灵活。在事件驱动内核基础上，Contiki实现了一种轻量级的名为protothread的线程模型，实现线性的、类似于线程的编程风格。该模型类似于Linux和Windows中线程的概念，多个线程共享同一个任务栈，从而减少RAM占用。

Contiki系统内部集成了两种类型的无线传感器网络协议栈：uIP和Rime。uIP是一个小型的符合RFC规范的TCP/IP协议栈，可以直接与Internet通信，uIP包含了IPv4和IPv6两种协议栈，支持TCP、UDP、ICMP等协议；Rime是一个轻量级、为低功耗无线传感器网络设计的协议栈，提供了大量的通信原语，能够实现从简单的一跳广播通信，到复杂的可靠多跳数据传输等通信功能。

5.1.2.2 Contiki的特点

Contiki操作系统也是一种嵌入式的多任务操作系统，是专为内存资源严重受限的嵌入

式设备开发设计的。通常一个典型的 Contiki 系统只需占用 2KB RAM 和 40KB ROM。Contiki 有如下突出的特点：

(1) Contiki 系统开发使用纯 C 语言，采用 C 编译器，如 GCC、IAR 等，开发调试简单。

(2) Contiki 兼容性好，可在 8 位、16 位、32 位几乎所有类型的处理器上移植。

(3) Contiki 采用模块化松耦合的结构方式，支持选择性的可抢占式多任务。

(4) 低功率无线电通信。Contiki 同时提供完整的 IP 网络和低功率无线电通信机制。对于无线传感器网络内部通信，Contiki 使用低功率无线电网络栈。

(5) 网络交互。可以通过多种方式完成与使用 Contiki 的传感器网络的交互，如 Web 浏览器、基于文本的命令行接口，或者存储和显示传感器数据的专用软件等，为传感器网络的交互与感知提供了一些特殊的命令。

(6) 能量效率。为了延长传感器网络的生命周期，控制和减少传感器节点的功耗很重要。Contiki 提供了一种基于软件的能量分析机制，记录每个传感器节点的能量消耗，这种机制不需要额外的硬件就能完成网络级别的能量分析。

5.1.2.3 Contiki 事件驱动和 protothread 机制

Contiki 有两个主要机制：事件驱动和 protothread 机制，前者是为了降低功耗，后者是为了节省内存。

1. 事件驱动

嵌入式系统常常被设计成响应周围环境的变化，而这些变化可以看成一个个事件。事件来了，操作系统处理之；没有事件到来，就去休眠(降低功耗)。这就是所谓的事件驱动，类似于中断。

在 Contiki 系统中，事件被分为以下 3 种类型：

1) 定时器事件(timer event)

进程可以设置一个定时器，在给定的时间完成之后生成一个事件，进程一直阻塞直到定时器终止，才继续执行。定时器事件对于周期性的操作很有帮助，如一些网络协议等。

2) 外部事件(external event)

外围设备连接到具有中断功能的微处理器 I/O 引脚，触发中断时可能生成事件。最常见的如按键中断，可以生成此类事件。这类事件发生后，相应的进程就会响应。

3) 内部事件(internal event)

任何进程都可以为自身或其他进程指定事件。这类事件对进程间的通信很有作用，例如通知某一个进程，数据已经准备好可以进行计算。

对事件的操作被称为投递(posted)。当一个进程执行时，中断服务程序将投递一个事件给进程。事件具有如下信息。

process：进程被事件寻址，它可以使特定的进程或者所有注册的进程。

event type：事件类型。开发者可以为进程定义一些事件类型用来区分它们，例如一个类型为收到数据包，另一个为发送数据包。

data：数据可以和事件一起提供给进程。

Contiki 操作系统的主要原理：事件投递给进程，进程触发后开始执行直到阻塞，然后等待下一个事件。

2. protothread 机制

传统的操作系统使用栈保存进程上下文，每个进程需要一个栈，这对于内存极度受限的传感器设备将难以忍受。protothread 机制解决了这个问题，通过保存进程被阻塞处的行数，从而实现进程切换，当该进程下一次被调度时，通过 switch(__LINE__)跳转到刚才保存的点，恢复执行。整个 Contiki 只用一个栈，当进程切换时清空，大大节省内存。

在 Contiki 中，protothread 的切换，实质是函数调用，通过 call_process()函数调用 protothread 函数体的函数指针切换 protothread，即“ret = p-> thread(&p-> pt, ev, data);”。这里的 p-> thread 指向的就是定义 protothread 的函数。而由于此函数中代码基本都是在 PT_BEGIN 和 PT_END 之间(宏展开后是一个完整的 switch 语句)，因此处于保存状态的就是在本函数中运行的位置，通过__LINE__保存上一次运行到哪里，然后当再次调用这个 protothread 时，就可以通过 switch 跳到上一次执行的地方继续执行。

5.1.2.4 Contiki 操作系统源代码结构分析

Contiki 是一个高度可移植的操作系统，其设计目的是获得良好的可移植性，因此源代码的组织很有特点。打开 Contiki 源文件目录，可以看到主要有 apps、core、cpu、doc、examples、platform、tools 等目录模块，如图 5.1 所示。下面对部分模块进行介绍。

apps	2016/12/23 14:41	文件夹	
core	2016/12/23 14:41	文件夹	
cpu	2016/12/23 14:41	文件夹	
doc	2016/12/23 14:41	文件夹	
examples	2015/3/16 20:59	文件夹	
platform	2016/12/23 14:41	文件夹	
tools	2016/12/23 14:41	文件夹	
zonesion	2016/12/23 14:41	文件夹	
.gitignore	2013/7/10 23:24	GITIGNORE 文件	1 KB
Makefile.include	2013/6/19 22:32	INCLUDE 文件	8 KB
pax_global_header	2013/6/19 22:32	文件	1 KB
README	2013/6/19 22:32	文件	1 KB
README.md	2013/6/19 22:32	MD 文件	1 KB
README-BUILDING	2013/6/19 22:32	文件	5 KB
README-EXAMPLES	2013/6/19 22:32	文件	7 KB

图 5.1 Contiki 源代码目录模块

1. apps 模块

apps 目录下是一些应用程序，例如 telnet、shell、webbrowser 等，在项目程序开发过程中可以直接使用。

2. core 模块

core 目录下是 Contiki 的核心源代码，包括网络、文件系统、外部设备、链接库等，并且包含了时钟、I/O、ELF 装载器、网络驱动等的抽象。

3. cpu 模块

cpu 目录下是 Contiki 目前支持的微处理器，例如 arm、stm32w108、cc253x 等。

4. platform 模块

platform 目录下是 Contiki 支持的硬件平台，例如 stm32f10x、cc2530dk 和 Win32 等。Contiki 的平台移植主要在这个目录下完成，该部分的代码与相应的硬件平台相关。

5. tools 模块

tools 目录下是开发过程中常用的一些工具，例如 CFS 相关的 makefsdata、网络相关的 tunslip、模拟器 cooja 和 mspsim 等。

为了获得良好的可移植性，除了 cpu 和 platform 中的源代码与硬件平台相关以外，其他目录中的源代码都尽可能与硬件无关。

5.2 任务 36 认识 Contiki 操作系统的数据结构

5.2.1 学习目标

熟悉 Contiki 操作系统常用的数据结构。

5.2.2 原理学习

5.2.2.1 进程的数据结构分析

进程结构体源代码如下：

```
struct process
{
    struct process * next;                  //指向下一个进程
    /* 进程名称 */
    #if PROCESS_CONF_NO_PROCESS_NAMES
      #define PROCESS_NAME_STRING(process) ""
    #else
      const char * name;
      #define PROCESS_NAME_STRING(process) (process)->name
    #endif
    /* 进程主体 */
    PT_THREAD((*thread)(struct pt *, process_event_t, process_data_t));
    struct pt pt;                           //保存进程中断行数的结构体
    unsigned char state;                    //进程执行状态
    unsigned char needspoll;                //进程优先级
};
```

下面针对进程结构体的源代码进行解析。

1. 进程名称

运用 C 语言预编译指令，可以配置进程名称，宏 PROCESS_NAME_STRING(process) 用于返回进程 process 名称，若系统无配置进程名称，则返回空字符串。

2. PT_THREAD 宏

PT_THREAD 宏定义如下：

```
#define PT_THREAD(name_args) char name_args
```

故该语句展开如下：

```
char ( * thread)(struct pt * , process_event_t, process_data_t);
```


该语句声明一个函数指针 thread,指向的是一个含有 3 个参数,返回值为 char 类型的函数。这是进程的主体,当进程执行时,主要是执行这个函数的内容。

3. pt 结构体

pt 结构体展开如下:

```
struct pt{
  lc_t lc;
};
typedef unsigned short lc_t;
```

lc(local continuations)用于保存程序被中断的行数,当该进程再次被调度时,程序会调到保存的行数继续执行。

4. 进程状态

进程共 3 个状态,宏定义如下:

```
#define PROCESS_STATE_NONE 0
/* 类似于 Linux 系统的僵尸状态,进程已退出,只是还没从进程链表删除 */
#define PROCESS_STATE_RUNNING 1  /* 进程正在执行 */
#define PROCESS_STATE_CALLED 2   /* 实际上是返回,并保存 lc 值 */
```

5. needspoll

needspoll 为进程的优先级,needspoll 为 1 的进程有更高的优先级。具体表现为:当系统调用 process_run()函数时,把所有 needspoll 标志为 1 的进程投入运行,然后才从事件队列取出下一个事件传递给相应的监听进程。

与 needspoll 相关的另一个变量 poll_requested,用于标识系统是否存在高优先级进程,即标记系统是否有进程的 needspoll 为 1。

```
static volatile unsigned char poll_requested;
```

将代码展开或简化,得到如表 5.1 所示的进程链表信息。

表 5.1　Contiki 进程链表信息

const char * name
char (* thread)(lc,ev,data)
struct pt pt
unsigned char state
unsigned char needspoll
struct process * next

5.2.2.2 事件的数据结构分析

1. 事件结构体源代码

```
struct event_data
{
  process_event_t ev;
  process_data_t data;
  struct process * p;
};
typedef unsigned char process_event_t;
typedef void * process_data_t;
```

各成员变量含义如下：ev，标识所产生事件；data，保存事件产生时获得的相关信息，即事件产生后可以给进程传递的数据；p，指向监听该事件的进程。

2. 事件分类

事件可以被分为 3 类：定时器事件(timer event)、外部事件和内部事件。Contiki 核心数据结构就只有进程和事件。

3. 系统定义的事件

1）系统事件

系统定义了 10 个事件，源代码和注释如下：

```
/ * 配置系统最大事件数 * /
# ifndef PROCESS_CONF_NUMEVENTS
# define PROCESS_CONF_NUMEVENTS 32
# endif
# define PROCESS_EVENT_NONE     0x80   //函数 dhcpc_request 调用
handle_dhcp(PROCESS_EVENT_NONE,NULL)
# define PROCESS_EVENT_INIT     0x81   //启动一个进程 process_start,通过传递该事件
# define PROCESS_EVENT_POLL     0x82   //在 PROCESS_THREAD(etimer_process, ev, data)使用到
# define PROCESS_EVENT_EXIT     0x83   //进程退出,传递该事件给进程主体函数 thread
# define PROCESS_EVENT_SERVICE_REMOVED     0x84
# define PROCESS_EVENT_CONTINUE 0x85   //PROCESS_PAUSE 宏用到这个事件
# define PROCESS_EVENT_MSG      0x86
# define PROCESS_EVENT_EXITED   0x87   //进程退出,传递该事件给其他进程
# define PROCESS_EVENT_TIMER    0x88   //etimer 到期时,传递该事件
# define PROCESS_EVENT_COM      0x89
/ * 进程初始化时,让 lastevent = PROCESS_EVENT_MAX,即新产生的事件从 0x8b 开始 * /
/ * 函数 process_alloc_event 用于分配一个新的事件 * /
# define PROCESS_EVENT_MAX      0x8a
```

PROCESS_EVENT_EXIT 与 PROCESS_EVENT_EXITED 的区别：事件 PROCESS_EVENT_EXIT 用于传递给进程的主体函数 thread，如在 exit_process 函数中的 p-> thread (&p-> pt, PROCESS_EVENT_EXIT, NULL)。而 PROCESS_EVENT_EXITED 用于传递给进程，如 call_process(q, PROCESS_EVENT_EXITED, (process_data_t)p)(注：EXITED 是完成式，发给进程，让整个进程结束。而一般式 EXIT，发给进程主体 thread，只

是使其退出 thread)。

2) 一个特殊事件

如果事件结构体 event_data 的成员变量 p 指向 PROCESS_BROADCAST,则该事件是一个广播事件;在 do_event 函数中,若事件的 p 指向的是 PROCESS_BROADCAST,则让进程链表 process_list 所有进程投入运行。部分源代码如下:

```
#define PROCESS_BROADCAST NULL        //广播进程
/*保存待处理事件的成员变量*/
ev = events[fevent].ev;
data = events[fevent].data;
receiver = events[fevent].p;
if (receiver == PROCESS_BROADCAST){
  for (p = process_list; p != NULL; p = p->next) {
    if (poll_requested) {
      do_poll();
    }
    call_process(p, ev, data);
  }
}
```

5.2.2.3 etimer 的数据结构分析

由于 etimer 是定时器中的一种,讲解 etimer 定时器之前先介绍 Contiki 系统中的几个定时器。

1. Contiki 系统的定时器

Contiki 包含一个时钟模型和 5 个定时器模型(timer, stimer, ctimer, etimer,rtimer)。5 种定时器简述如下。

timer,stimer:提供了最简单的时钟操作,即检查时钟周期是否已经结束。应用程序需要从 timer 中读出状态,判断时钟是否过期。两种时钟最大的不同在于,tmier 使用的是系统时钟的 tick,而 stimer 使用的是秒,也就是 stimer 的一个时钟周期要长一些。和其他的时钟不同,这两个时钟能够在中断中安全使用,可以用到低层的驱动代码上。

ctimer:回调定时器,驱动某一个回调函数。

etimer:事件定时器,驱动某一个事件。

rtimer:实时时钟。

2. etimer 结构体

etimer 提供一种 timer 机制产生定时器事件,可以理解成 etimer 是 Contiki 特殊的一种事件。当 etimer 到期时,会给相应的进程传递事件 PROCESS_EVENT_TIMER,从而使该进程启动。

timer 仅包含起始时刻和间隔时间,所以 timer 只记录到期时间,通过比较到到期时间和新的当前时钟,从而判断该定时器是不是到期。

3. 定时器的产生及处理

通过 add_timer 函数将 etimer 加入 timerlist,etimer 处理由系统进程 etimer_process 负责。

5.3 任务 37 Contiki 操作系统移植

5.3.1 学习目标

- 理解 Contiki 的基本工作原理；
- 学会在 STM32 平台上进行 Contiki 操作系统移植。

5.3.2 开发环境

- 硬件：ZXBee 无线节点板，调试转接板，USB MINI 线，J-Link 仿真器，PC；
- 软件：Windows XP/7/8/10，IAR 集成开发环境，串口调试工具。

5.3.3 原理学习

Contiki 采用事件驱动机制，怎样才能够产生事件呢？阅读 5.1 节的内容就很容易知道怎样才能够产生事件，如：通过时钟定时，定时事件到就产生一个事件；通过某种中断，某个中断发生，就产生某个事件，例如外部中断。那么移植 Contiki 到底要做哪些工作呢？首先时钟一定是必要的，所以移植 Contiki 系统的重点在于系统时钟。

5.3.4 开发内容

理解 Contiki 系统的移植原理之后，再进行系统的移植将会变得很容易，下面详细介绍 Contiki 系统移植的整个过程。

1）下载源代码

开发者可以从官网获取源代码，Contiki 官网下载系统源代码地址：http://sourceforge.net/projects/contiki/files/Contiki/Contiki%202.6/。

2）创建工程

将 contiki-2.6 整个文件夹复制到 PC 任意目录下，进入 Contiki 系统源代码的 contiki-2.6\zonesion\example\iar 目录下，创建 5.3-testSample 目录，然后在 5.3-testSample 目录下创建一个 IAR 工程，工程名称以及工作空间的名称为 testSample，过程如下：

（1）创建一个一个空的 arm 工程，命名为 testSample。

选择 Project→Create New Project，创建工程，如图 5.2 所示。

在 Tool chain(工具链)下拉列表框中选择 ARM，并选择 Empty project，如图 5.3 所示。

单击 OK 按钮后，就会提示工程的保存路径，并填写工程名，此处填写为 testSample，如图 5.4 所示。

（2）保存工作空间并命名为 testSample，选择 File→Save Workspace，保存工程，如图 5.5 所示。

（3）创建完工程后，在工程目录下即可看到新增了几个文件，如图 5.6 所示。

3）给工程添加组目录

方法：在工程名上右击，在弹出的快捷菜单中选择 Add→Add Group，然后填写组名称即可，如图 5.7 所示。

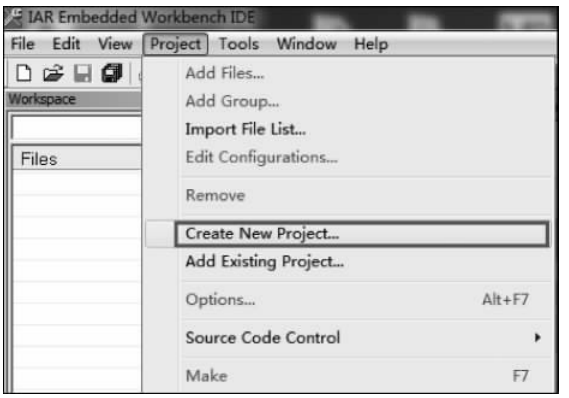

图 5.2　创建工程 1

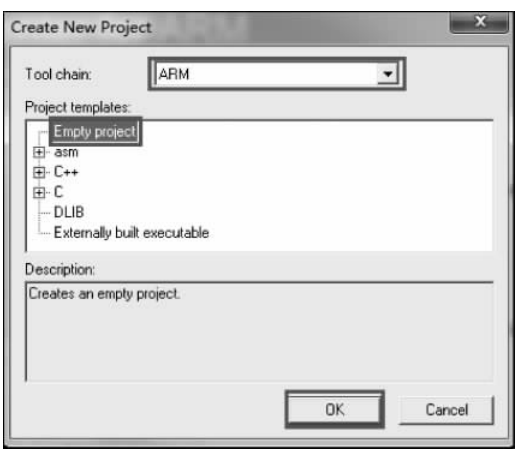

图 5.3　创建工程 2

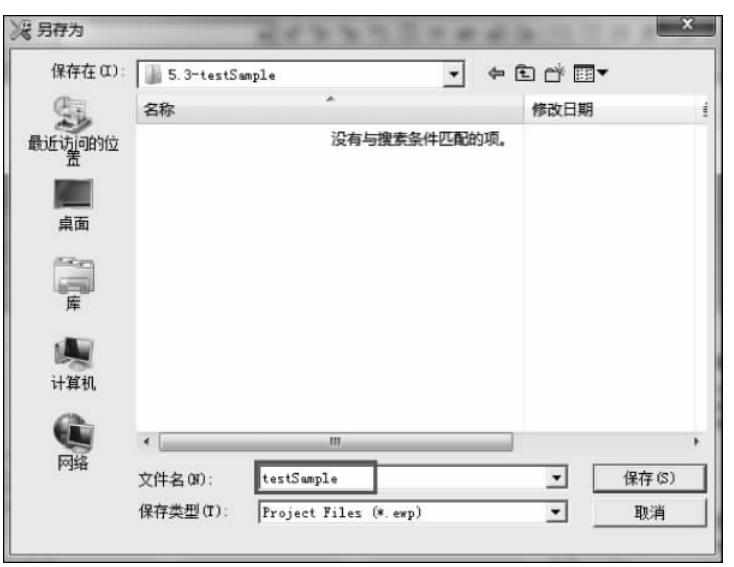

图 5.4　保存工程并命名

图 5.5　保存工作空间

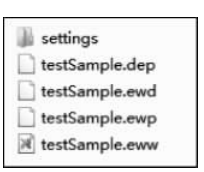

图 5.6　工程文件

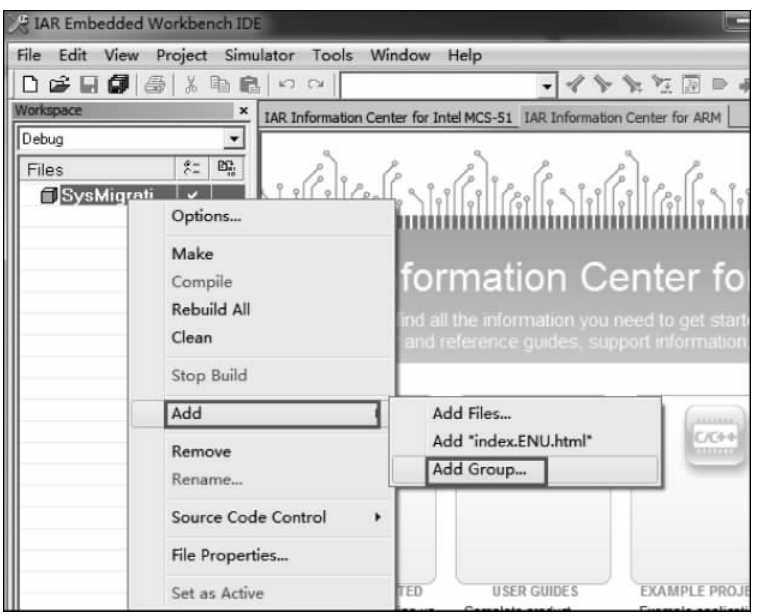

图 5.7　添加组

按照添加组的方法，依次添加如图 5.8 所示的组目录结构。

以 core 目录下的子目录 sys 目录为例，只要在 core 文件夹上右击，在弹出的快捷菜单中选择 Add→Group 即可添加子目录。

4）添加完组目录之后，往组里面添加.c 等文件

（1）添加 Contiki 系统文件。在工程 sys 目录名上右击，在弹出的快捷菜单中选择Add→Add Files，如图 5.9 所示。

在工程的 sys 组目录下添加 Contiki 的系统文件：autostart.c、ctimer.c、etimer.c、process.c、timer.c，这几个文件在 Contiki 系统源代码的 contiki-2.6\core\sys 目录下，如图 5.10 所示。

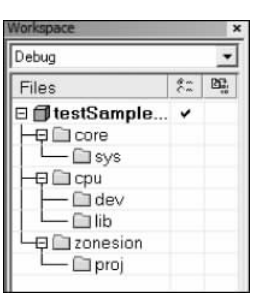

图 5.8　工程组目录

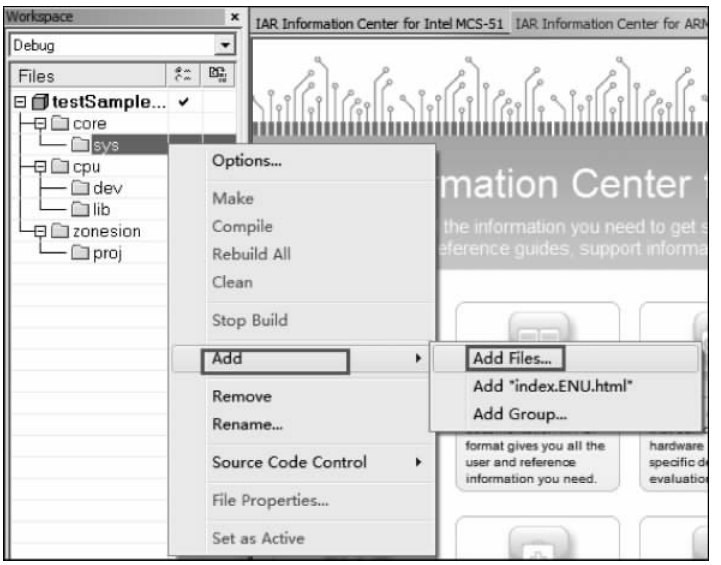

图 5.9　添加 Contiki 系统文件 1

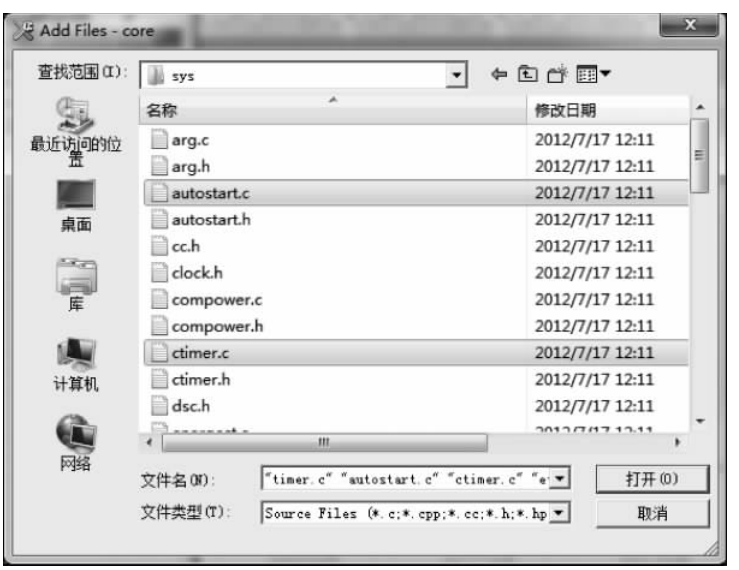

图 5.10　添加 Contiki 系统文件 2

(2) 添加 STM32 官方库文件。将 STM32F10x_StdPeriph_Lib_V3.5.0(ST 公司提供的 STM32 标准库文件,3.5 版本)放在 Contiki 系统源代码的 contiki-2.6\cpu\arm\stm32f10x 目录下。在工程的 cpu\lib 组目录下添加 STM32 的库文件到工程中,添加的文件为:system_ stm32f10x.c、startup_stm32f10x_md.s、misc.c、stm32f10x_exti.c、stm32f10x_gpio.c、stm32f10x_rcc.c、stm32f10x_usart.c。

文件所在目录说明:

- system_ stm32f10x.c 文件所在目录为 STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x。

- startup_stm32f10x_md.s 文件所在目录为 STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\CMSIS\CM3\DeviceSupport\ST\STM32F10x\startup\iar。
- 其余 5 个文件所在目录均为 STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\STM32F10x_StdPeriph_Driver\src。

(3) 在工程的 cpu 组目录下添加 Contiki 系统时钟 clock.c 文件,该文件所在目录为 contiki-2.6\cpu\arm\stm32f10x。

5) 创建 contiki-main.c 文件

通过上述步骤,移植 Contiki 系统所需要的 Contiki 系统文件、STM32 官方库所需文件都添加完毕,若想让程序执行就必须有 main 函数,上面添加的都是相应的支持文件,下面在工程的 zonesion\proj 组目录下新建一个 contiki-main.c 文件。创建方法如下:

选择 File→New→File 或者直接单击 File 下面的 New Document 按钮创建一个空白文件,如图 5.11 所示。

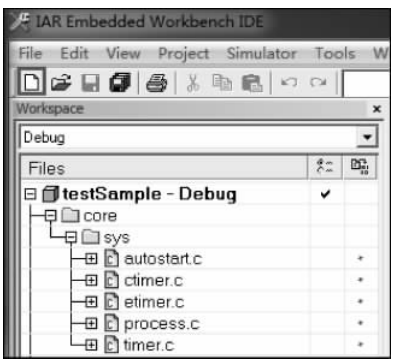

图 5.11 创建 contiki-main.c 文件 1

按下 Ctrl+S 键保存文件,保存路径选择当前工程 testSample 的根目录下,保存文件名为 contiki-main.c,如图 5.12 所示。

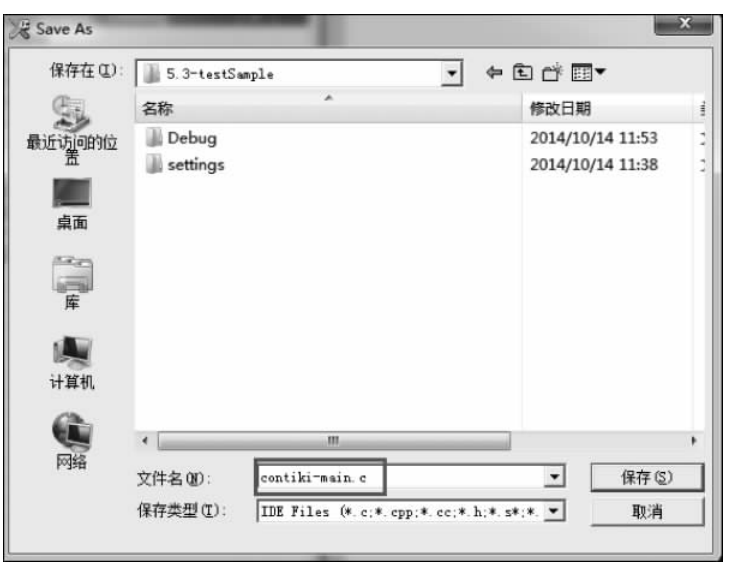

图 5.12 创建 contiki-main.c 文件 2

然后,再将已创建的 contiki-main.c 文件添加到工程的 zonesion\proj 组目录下。

添加完成后整个工程目录结构如图 5.13 所示。

6) 配置工程

添加完.c 文件后,就需要配置工程,如硬件芯片的型号选择、头文件的路径等。配置方法及步骤如下:

(1) 在工程名上右击,在弹出的快捷菜单中选择 Options,如图 5.14 所示。

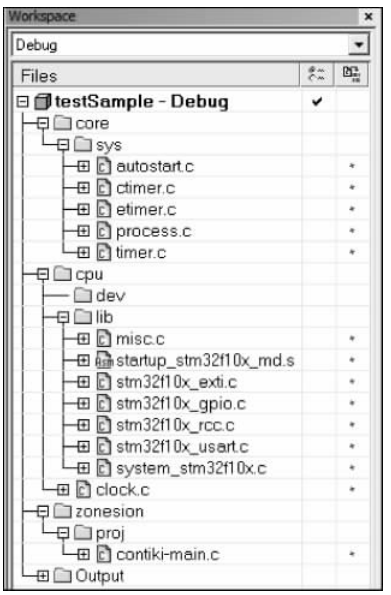

图 5.13 工程目录

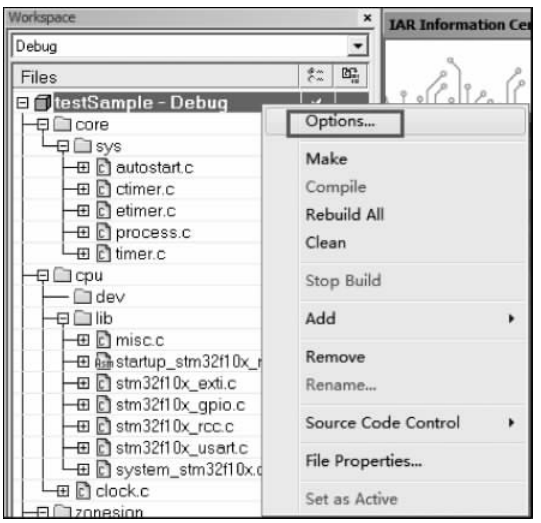

图 5.14 工程配置

(2) 在 General Options 配置页面,选中 Device 单选按钮,然后选择 STM32F10xxB,如图 5.15 所示。

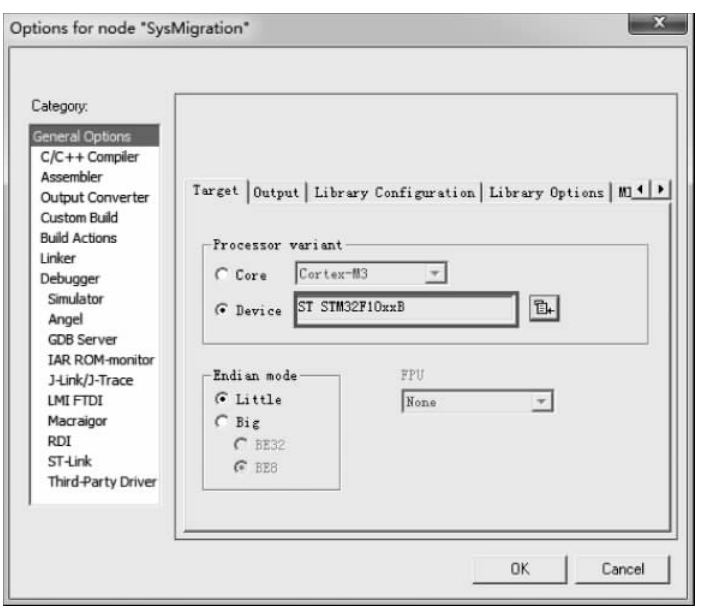

图 5.15 芯片型号选择

(3) 在 Debugger 配置页面,将 Driver 选项设置成 J-Link/J-Trace,如图 5.16 所示。

(4) 添加头文件的路径和宏定义。

配置完工程选项后,需要在工程配置选项里面添加头文件的路径,否则在编译.c 文件

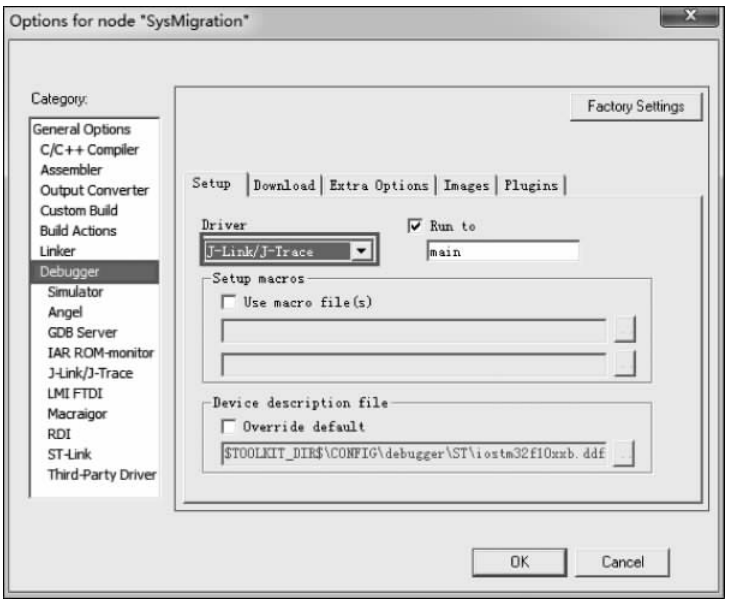

图 5.16　Debugger 选项配置

时会因找不到相应的头文件而出现编译错误。

```
$PROJ_DIR$\..\..\..\..\core
$PROJ_DIR$\..\..\..\..\core\lib
$PROJ_DIR$\..\..\..\..\core\sys
$PROJ_DIR$\..\..\..\..\zonesion\example
$PROJ_DIR$\..\..\..\..\cpu\arm\stm32f10x
$PROJ_DIR$\..\..\..\..\cpu\arm\stm32f10x\STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\
STM32F10x_StdPeriph_Driver\inc
$PROJ_DIR$\..\..\..\..\cpu\arm\stm32f10x\STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\
CMSIS\CM3\DeviceSupport\ST\STM32F10x
$PROJ_DIR$\..\..\..\..\cpu\arm\stm32f10x\STM32F10x_StdPeriph_Lib_V3.5.0\Libraries\
CMSIS\CM3\CoreSupport\
```

添加宏定义：

```
STM32F10X_MD
USE_STDPERIPH_DRIVER
_DLIB_FILE_DESCRIPTOR
MCK=72000000
```

宏定义说明：STM32F10X_MD，中容量型号的 cpu；USE_STDPERIPH_DRIVER，使用官方提供的标准库；MCK = 72000000，MCU 的主频为 72MHz；_DLIB_FILE_DESCRIPTOR，文件描述符。

头文件及宏定义的添加方法：将头文件路径复制到 C/C++Compiler 配置选项中的 Additionnal include derectories 列表框中；将宏定义添加到 Defined symbols 列表框中。添

加完成后如图 5.17 所示。

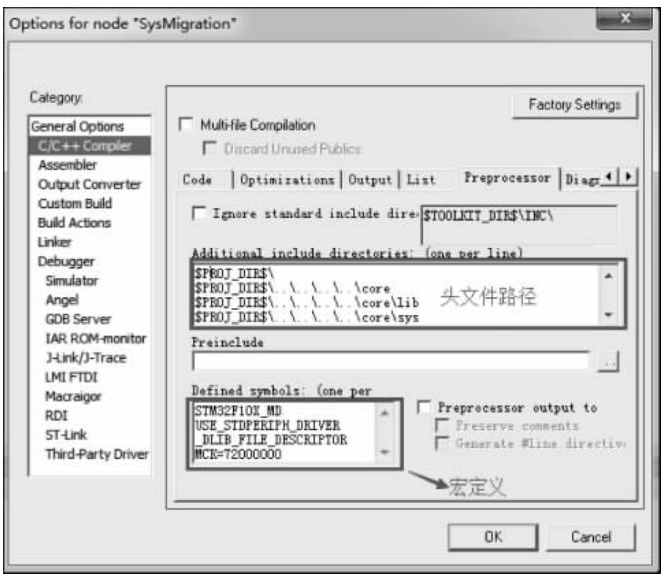

图 5.17　添加头文件路径和宏定义

7）系统时钟

接下来介绍整个系统移植的重点，也就是系统时钟。没有系统时钟，系统就跑不起来。在本次移植过程中，并不需要修改 clock.c 文件，因为该 clock.c 文件已经修改好了，直接拿过来使用即可。下面对 clock.c 里面的源代码进行相应的解析。

（1）系统时钟初始化。

系统时钟初始化是整个 STM32 工作的核心，根据 STM32 的主频，以及 CLOCK_SENCOND 的参数，可以设置系统时钟中断的时间。

```
void clock_init()
{
  NVIC_SET_SYSTICK_PRI(8);
  SysTick->LOAD = MCK/8/CLOCK_SECOND;
  SysTick->CTRL = SysTick_CTRL_ENABLE | SysTick_CTRL_TICKINT;
}
```

（2）系统时钟中断处理函数。

要让 Contiki 操作系统运行起来，关键部分是启动系统时钟，对应到 Contiki 系统的进程就是启动 etimer 进程。etimer_process 由 Contiki 系统提供，这里只需对系统时钟进行初始化并定时更新系统时钟（开发者自定义 current_clock），并判断 etimer 的下一个定时时刻是否已到（通过比较 current_clock 与 etimer 的定时时刻判定）如果时钟等待序列中有等待时钟的进程，那么就调用 etimer 进程执行，通过其唤醒相关进程。

```
void Systick_handler(void)
{
```

```
    (void)SysTick->CTRL;
    SCB->ICSR = SCB_ICSR_PENDSTCLR;
    current_clock++;

    if(etimer_pending() && etimer_next_expiration_time() <= current_clock) {
      etimer_request_poll();
      /* printf("etimer: %d, %d\n", clock_time(),etimer_next_expiration_time()); */
    }
    if ( --second_countdown == 0) {
      current_seconds++;
      second_countdown = CLOCK_SECOND;
    }
  }
```

Contiki 系统移植需要修改的源代码就是更改系统时钟初始化以及中断服务函数。上述步骤完成后，系统的移植工作基本完成了，接下来验证系统是否移植成功。

要验证系统是否移植成功，需写一段代码测试。在 contiki-main.c 里面添加如下内容：

```
#include <stm32f10x_map.h>
#include <stm32f10x_dma.h>
#include <gpio.h>
//省略头文件

unsigned int idle_count = 0;
int main()
{
  clock_init();
  process_init();
  process_start(&etimer_process, NULL);
  printf("HelloWorld!\r\n");
  while(1) {
    do {
    } while(process_run() > 0);
    idle_count++;
  }
  return 0;
}
/* 参数校验函数,此处为空,官方库函数需要调用 */
void assert_param(int b)
{
}
```

添加完成后，将程序烧写到 ZXBee 无线节点板中，Contiki 系统可以运行起来了，但是要想看到直观的效果，可以尝试在 main 函数中添加串口打印的方法：在 Contiki 系统源代码目录 contiki-2.6\cpu\arm\stm32f10x\dev 目录下已经有了串口驱动的程序（uart1.c、uart2.c），在这里直接拿过来用即可，在工程的 cpu\dev 组目录下打开 uart1.c 文件，然后再在工程配置选项添加 uart1.h 头文件的路径：

```
$PROJ_DIR$\..\..\cpu\arm\stm32f10x\dev\
```


最后在 main 中添加串口初始化，以及串口打印消息的方法即可。

最终的 contiki-main.c 文件源代码如下：

```
#include <stm32f10x_map.h>
//头文件省略
unsigned int idle_count = 0;

int main()
{
  clock_init();
  uart1_init(115200);
  process_init();
  process_start(&etimer_process, NULL);
  printf("HelloWorld!\r\n");
  while(1) {
    do {
    } while(process_run() > 0);
    idle_count++;
  }
  return 0;
}

/* 参数校验函数,此处为空,官方库函数需要调用 */
void assert_param(int b)
{
}
```

5.3.5 开发步骤

(1) 通过调试转接板将 J-Link 仿真器连接到 PC 和 ZXBee 无线节点板，在 PC 上打开串口助手或者超级终端，设置接收的波特率为 115 200b/s；

(2) 参考 5.3.4 节的系统移植过程，并打开移植成功的工程文件(双击 testSample.eww 文件)；

(3) 给连接好的硬件平台起电，将程序下载到 ZXBee 无线节点板中；

(4) 下载完毕后选择 Debug→Go，运行程序；

(5) 程序成功运行后，在 PC 上打开串口助手或者超级终端，设置接收的波特率为 115 200b/s，数据位为 8，奇偶校验为无，停止位为 1，数据流控制为无。

5.3.6 总结与扩展

程序成功运行后，在串口显示区显示：

```
HelloWorld!
```

上面的显示即可表明 Contiki 系统已经成功移植到 ZXBee 无线 STM32 处理器。

5.4 任务 38 Contiki 操作系统的进程开发

5.4.1 学习目标

- 理解 Contiki 进程的工作流程；
- 学会编写简单的进程开发程序。

5.4.2 开发环境

- 硬件：ZXBee 无线节点板，调试转接板，USB MINI 线，J-Link 仿真器，PC；
- 软件：Windows XP/7/8/10，IAR 集成开发环境，串口调试工具。

5.4.3 原理学习

本任务介绍 Contiki 系统中进程的使用方法以及相关原理，定义了一个 blink_process 进程，驱动 ZXBee 无线节点板上的 D4、D5 灯进行闪烁，blink_process 进程定义在工程根目录下的 blink.c 文件中，源代码如下：

```
#include "contiki.h"
#include "dev/leds.h"
#include <stdio.h>
static struct etimer et_blink;
static uint8_t blinks;
//定义 blink_process 进程
PROCESS(blink_process, "blink led process");
//将 blink_process 进程定义成自启动
AUTOSTART_PROCESSES(&blink_process);
PROCESS_THREAD(blink_process, ev, data)
{

  PROCESS_BEGIN();                          //进程开始
  blinks = 0;
  while(1) {
    etimer_set(&et_blink, CLOCK_SECOND);    //设置定时器 1s
    PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
    leds_off(LEDS_ALL);                     //关灯
    leds_on(blinks & LEDS_ALL);             //开灯
    blinks++;
    printf("Blink… (state %0.2X)\n\r", leds_get());
  }
  PROCESS_END();                            //进程结束
}
```

在上述代码当中，leds_on(blinks&LEDS_ALL)函数实现了两个 LED 点亮的不同方式，其中函数的参数功能是选择点亮哪一个 LED。在本任务中，LED 的显示一共有 3 种状

态：①D4 点亮；②D5 点亮；③D4 和 D5 同时点亮。下面根据 blink_process 进程进行详细的解析。

1. PROCESS 宏

PROCESS 宏通过声明一个函数定义进程，该函数是进程的执行体，源代码展开如下：

```
PROCESS(blink_process, "blink led process ");
#define PROCESS(name, strname) PROCESS_THREAD(name, ev, data);
struct process name = { NULL, strname, process_thread_## name }
```

将宏展开为：

```
#define PROCESS((blink_process, " blink led process ")
PROCESS_THREAD(blink_process, ev, data);
struct processblink_process = { NULL, "blink led process ", process_thread_blink_process };
```

1）PROCESS_THREAD 宏

PROCESS_THREAD 宏用于定义进程的执行主体，宏展开如下：

```
#define PROCESS_THREAD(name, ev, data) \
static PT_THREAD(process_thread_## name(struct pt * process_pt, process_event_t ev,
process_data_t data))
```

进一步展开为：

```
//PROCESS_THREAD(blink_process, ev, data);
static PT_THREAD(process_thread_blink_process(struct pt * process_pt, process_event_t ev,
process_data_t data));
```

2）PT_THREAD 宏

PT_THREAD 宏用于声明一个 protothread，即进程的执行主体，宏展开如下：

```
#define PT_THREAD(name_args) char name_args
```

进一步展开为：

```
//static PT_THREAD(process_thread_blink_process(struct pt * process_pt, process_event_t
ev, process_data_t data));
static char process_thread_blink_process(struct pt * process_pt, process_event_t ev,
process_data_t data);
```

上面的宏定义其实就是声明一个静态的函数 process_thread_blink_process，返回值是 char 类型。

3）定义一个进程

PROCESS 宏展开的第二句，定义一个进程 blink_process，源代码如下：

```
struct processblink_process = { NULL, " Blink led process ", process_thread_blink_process };
```

结构体 process 定义如下：

```
struct process
{
    struct process * next;
    const char * name;      /* 此处略做简化，源代码包含了预编译 # if，可以通过配置，使得进
                            程名称可有可无 */
    PT_THREAD(( * thread)(struct pt * , process_event_t, process_data_t));
    struct pt pt;
    unsigned char state, needspoll;
};
```

进程 blink_process 的 lc、state、needspoll 都默认置为 0。关于 process 结构体请参见 5.2 节内容。

2. AUTOSTART_PROCESSES 宏

AUTOSTART_PROCESSES 宏实际上是定义一个指针数组，存放 Contiki 系统运行时需自动启动的进程，宏展开如下：

```
//AUTOSTART_PROCESSES(&blink_process);
#define AUTOSTART_PROCESSES(…) \ struct process * const autostart_processes[] = {__VA_
ARGS__, NULL}
```

这里用到 C99 支持可变参数宏的特性，如 # define debug(…) printf(__VA_ARGS__)，缺省号代表一个可以变化的参数表，宏展开时，实际的参数就传递给 printf()了。例如“debug("Y＝%d\n", y);”被替换成“printf("Y＝%d\n", y); ”，那么，“AUTOSTART_PROCESSES(&blink_process);”实际上被替换成：

```
struct process * const autostart_processes[] = {&blink_process, NULL};
```

这样就知道如何让多个进程自启动了，直接在宏 AUTOSTART_PROCESSES()加入需自启动的进程地址，如让 hello_process 和 world_process 这两个进程自启动，源代码如下：

```
AUTOSTART_PROCESSES(&hello_process,&world_process);
```

最后一个进程指针设成 NULL，则是一种编程技巧，设置一个“哨兵”（提高算法效率的一个手段），以提高遍历整个数组的效率。

3. PROCESS_THREAD 宏

“PROCESS(blink_process, "Blink led process");”展开成两句，其中有一句是“PROCESS_THREAD(blink_process, ev, data);”。这里要注意到分号，它是一个函数声明。而 PROCESS_THREAD(blink_process, ev, data)没有分号，而是紧跟着“{}”，是上述声明函数的实现。关于 PROCESS_THREAD 宏的分析，最后展开如下：

```
static char process_thread_blink_process(struct pt * process_pt, process_event_t ev,
process_data_t data);
```

注意：在阅读 Contiki 源代码、手动展开宏时，要特别注意分号。

4. PROCESS_BEGIN 宏和 PROCESS_END 宏

原则上，所有的执行代码都要放在 PROCESS_BEGIN 宏和 PROCESS_END 宏之间（如果程序全部使用静态局部变量，这样做总是对的。倘若使用局部变量，情况比较复杂。当然，不建议这样做）。

1）PROCESS_BEGIN 宏

PROCESS_BEGIN 宏一步步展开如下：

```
#define PROCESS_BEGIN() PT_BEGIN(process_pt)
```

process_pt 是 struct pt * 类型，在函数头传递过来的参数（参见 3 的宏定义展开），直接理解成 lc，用于保存当前被中断的地方，以便下次恢复执行。继续展开如下：

```
#define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1; LC_RESUME((pt)->lc)
#define LC_RESUME(s) switch(s) { case 0:
```

替换参数，结果如下：

```
{
    char PT_YIELD_FLAG = 1;      /* 将 PT_YIELD_FLAG 置 1，类似于关中断 */
    switch(process_pt->lc)       /* 程序根据 lc 的值进行跳转，lc 用于保存程序断点 */
    {
        case 0:                  /* 第一次执行从这里开始，可以放一些初始化的内容 */
        ;
```

PROCESS_BEGIN 宏展开并不是完整的语句，可以通过下面的 PROCESS_END 宏定义查看 Contiki 是如何设计的。

2）PROCESS_END 宏

PROCESS_END 宏一步步展开如下：

```
#define PROCESS_END() PT_END(process_pt)
#define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0; \ PT_INIT(pt); return PT_ENDED; }
#define LC_END(s) }
```

```
#define PT_INIT(pt) LC_INIT((pt)->lc)
#define LC_INIT(s) s = 0;
#define PT_ENDED 3
```

得到 PROCESS_END 宏源代码如下：

```
    }
    PT_YIELD_FLAG = 0;
    (process_pt)->pt = 0;
    return 3;
}
```

综合来看容易理解 PROCESS_BEGIN 宏和 PROCESS_END 宏的作用。

5. 宏展开和总结

1）宏全部展开

根据上述分析，该实例全部展开的代码如下：

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

static char process_thread_blink_process(struct pt * process_pt, process_event_t ev,
process_data_t data);

struct process blink_process = { ((void *)0), "Blink led process", process_thread_blink_
process};
struct process * const autostart_processes[] = {&blink_process, ((void *)0)};

char process_thread_blink_process(struct pt * process_pt, process_event_t ev, process_data_
t data)
{
    {
        char PT_YIELD_FLAG = 1;
        switch((process_pt)->lc)
        {
            case 0:
            blinks = 0;
            while(1) {
            etimer_set(&et_blink, CLOCK_SECOND);          //设置定时器 1s
            PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
            leds_off(LEDS_ALL);                           //关灯
            leds_on(blinks & LEDS_ALL);                   //开灯
            blinks++;
            printf("Blink… (state %0.2X)\n\r", leds_get());
        }

    };
}
```

```
    PT_YIELD_FLAG = 0;
    (process_pt) -> lc = 0;
    return 3;
}
```

2）宏总结

本实例用到的宏总结如下，以后就直接把宏当 API 用。声明进程 name 的主体函数 process_thread_＃＃name(进程的 thread 函数指针所指的函数)，并定义一个进程 name。

```
PROCESS(name, strname)
```

定义一个进程指针数组 autostart_processes：

```
AUTOSTART_PROCESSES(…)
```

进程 name 的定义或声明，取决于宏后面是“;”还是“{}”：

```
PROCESS_THREAD(name, ev, data)
```

进程的主体函数开始标志：

```
PROCESS_BEGIN()
```

进程的主体函数结束标志：

```
PROCESS_END()
```

进程的主体函数从这里结束。

3）编程模型

本实例给出了定义一个进程的模型(以 Hello world 为例)，实际编程过程中，只需要将“printf("Hello world! \n\r");”换成自己需要实现的代码即可。

```
//假设进程名称为 Hello world
＃include "contiki.h"
＃include < stdio.h >

PROCESS(blink_process, "Hello world");          //PROCESS(name, strname)
AUTOSTART_PROCESSES(&blink_process);            //AUTOSTART_PROCESS(…)

PROCESS_THREAD(blink_process, ev, data)         //PROCESS_THREAD(name, ev, data)
{
    PROCESS_BEGIN();
    /***这里填入执行代码***/
    PROCESS_END();
}
```

注意：声明变量最好不要放在 PROCESS_BEGIN 之前，因为进程再次被调度，总是从头开始执行，直到 PROCESS_BEGIN 宏中的 switch 判断才跳转到断点 case __LINE__。也就是说，进程被调度总是会执行 PROCESS_BEGIN 之前的代码。

上述内容主要是结合 LED 示例进程源代码分析了进程在定义、执行时的原理，并没有讲到 LED 的驱动程序与 Contiki 系统之间的关联，那么 Contiki 系统是如何实现对 LED 灯的控制呢？

通过 LED 的显示进程，可以看到 LED 的控制调用了 leds_off()、leds_on()这两个方法，这两个方法的区别就是对 LED 灯的 I/O 引脚的电平分别置为高电平、低电平。以 leds_on()方法为例，通过分析源代码，Contiki 系统对 LED 灯的控制过程调用的方法流程如图 5.18 所示。

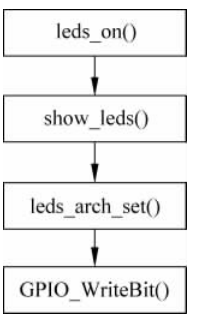

图 5.18　LED 灯控制调用的方法过程

shao_leds()方法是确定要点亮的 LED 灯序号，GPIO_WriteBit()方法就是给 LED 灯的 I/O 引脚置为高或低电平，从而达到控制 LED 的功能。

5.4.4　开发步骤

(1) 通过调试转接板将 J-Link 仿真器连接到 PC 和 ZXBee 无线节点板，在 PC 上打开串口助手或者超级终端，设置接收的波特率为 115 200b/s；

(2) 打开例程，将本任务开发例程整个文件夹复制到源代码目录的\zonesion\example\iar 文件夹下，打开任务工程；

(3) 给连接好的硬件平台起电，将程序下载到 ZXBee 无线节点板中；

(4) 下载完毕后选择 Debug→Go，运行程序；

(5) 程序成功运行后，在 PC 上打开串口助手或者超级终端，设置接收的波特率为 115 200b/s，数据位为 8，奇偶校验为无，停止位为 1，数据流控制为无。

5.4.5　总结与扩展

程序成功运行后，此时在 ZXBee 无线节点板可观察到 D4 和 D5 进行有规则的点亮，同时在串口显示区有如下显示：

```
Starting Contiki 2.6 on STM32F10x
autostart_start: starting process 'blink led process'
Blink… (state 00)
Blink… (state 01)
Blink… (state 02)
Blink… (state 03)
```

串口显示区的“Starting Contiki 2.6 on STM32F10x”表明 Contiki 操作系统成功移植到 ZXBee 无线节点板中，Blink…(state 00)表示 D4 亮，Blink…(state 01)表示 D5 亮，Blink…(state 03)表示 D4 和 D5 同时亮。

5.5 任务 39 Contiki 多进程开发

5.5.1 学习目标

- 理解 Contiki 多线程工作流程；
- 学会 Contiki 多线程程序开发。

5.5.2 开发环境

- 硬件：ZXBee 无线节点板、调试转接板，USB MINI 线，J-Link 仿真器，PC；
- 软件：Windows XP/7/8/10，IAR 集成开发环境，串口调试工具。

5.5.3 原理学习

在本任务当中定义了两个进程，一个是显示 Hello world 的进程，另一个是 LED 的闪烁进程。定义过程如下：

1）定义 Hello world 进程

```
PROCESS(hello_world_process, "Hello world process");
```

2）定义 LED 闪烁进程

```
PROCESS(blink_process, "LED blink process");
```

3）两个进程定义结束之后在自动启动进程的参数列表里面加上两个进程名

```
AUTOSTART_PROCESSES(&hello_world_process, &blink_process);
```

4）编写 hello_world_process 进程和 blink_process 进程的执行体

```
//hello_world 打印进程
PROCESS_THREAD(hello_world_process, ev, data)
{
  PROCESS_BEGIN();
  etimer_set(&et_hello, CLOCK_SECOND * 4);                //设置定时器 4s
  while(1) {
    PROCESS_WAIT_EVENT();
    if(ev == PROCESS_EVENT_TIMER) {
      printf("Hello world!\n\r");
      etimer_reset(&et_hello);
    }
  }
  PROCESS_END();
}
```

```
//LED灯闪烁进程
PROCESS_THREAD(blink_process, ev, data)
{
  PROCESS_BEGIN();
  blinks = 0;
  while(1) {
    etimer_set(&et_blink, CLOCK_SECOND);          //设置定时器1s
    PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
    leds_off(LEDS_ALL);
    leds_on(blinks & LEDS_ALL);
    blinks++;
    printf("Blink… (state %0.2X)\n\r", leds_get());
  }
  PROCESS_END();
}
```

两个进程的执行体分别添加打印 Hello world 和控制 LED 闪烁的代码，然后在 Contiki 的 main 函数中调用这两个进程即可实现多线程的运行。

在本任务中，Hello world 进程设置成 4s 执行一次，blink 进程设置成一秒执行一次，blink 进程中 LED 的闪烁分成 3 个状态：状态 01 表示 D4 亮；状态 02 表示 D5 亮；状态 03 表示 D4 和 D5 同时亮。

5.5.4 开发步骤

(1) 通过调试转接板将 J-Link 仿真器连接到 PC 和 ZXBee 无线节点板，在 PC 上打开串口助手或者超级终端，设置接收的波特率为 115 200b/s；

(2) 打开例程，将本任务开发例程整个文件夹复制到源代码目录的“\zonesion\example\iar”文件夹下，打开任务工程；

(3) 给连接好的硬件平台起电，将程序下载到 ZXBee 无线节点板中；

(4) 下载完毕后选择 Debug→Go，运行程序；

(5) 在 PC 上打开串口助手或者超级终端，设置接收的波特率为 115 200b/s，数据位为 8，奇偶校验为无，停止位为 1，数据流控制为无。

5.5.5 总结与扩展

程序成功运行后，在串口显示区显示：

```
Starting Contiki 2.6 on STM32F10x
autostart_start: starting process 'Blink led process'
autostart_start: starting process 'LED blink process'
Blink… (state 01)
Blink… (state 02)
Blink… (state 03)
Hello world!
Blink… (state 00)
```

```
Blink… (state 01)
Blink… (state 02)
Blink… (state 03)
Hello world!
…
```

上面显示的信息表示两个进程开始运行，串口显示区显示 Blink…(state 01)时，可以看到 ZXBee 无线节点板上的 D4 亮；显示 Blink…(state 02)时，可以看到 D5 亮；显示 Blink…(state 03)时，可看到 D4 和 D5 同时亮。显示 Hello world 时表明 Hello world 进程开始执行，并在串口显示区打印“Hello world!”。

5.6 任务 40 Contiki 进程通信基础开发

5.6.1 学习目标

- 理解 Contiki 进程共享内存的通信原理；
- 学会在 STM32 微处理器上编写开发程序，实现多进程通信。

5.6.2 开发环境

- 硬件：ZXBee 无线节点板，调试转接板，USB MINI 线，J-Link 仿真器，PC；
- 软件：Windows XP/7/8/10，IAR 集成开发环境，串口调试工具。

5.6.3 原理学习

进程通信的方式有多种，本任务使用的是共享内存的方式，共享内存允许两个或多个进程共享一给定的存储区，因为数据不需要来回复制，所以它是最快的一种进程间通信机制。

在本任务中，定义了两个进程，一个是 count_process 进程，另一个是 print_process 进程。在 count_process 进程中，添加 LED 翻转的效果，以便达到观看 count_process 进程是否执行，同时设置一个静态变量 count，只要 count 的值发生变化，print_process 进程可以即时看到 count 数值的变化，并将其显示出来。

下面是两个进程的实现源代码：

```
static process_event_t event_data_ready;
/* 定义 count 和 print 进程 */
PROCESS(count_process, "count process");
PROCESS(print_process, "print process");
/* 将两个进程设置成自启动 */
AUTOSTART_PROCESSES(&count_process, &print_process);

PROCESS_THREAD(count_process, ev, data)              //count 进程执行体
{
    static struct etimer count_timer;
```

```
    static int count = 0;
    PROCESS_BEGIN();

    event_data_ready = process_alloc_event();
    etimer_set(&count_timer, CLOCK_SECOND / 2);     //设置定时器 2s
    leds_init();                                    //led 初始化
    leds_on(1);                                     //点亮 led1
    while(1)
    {
        PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);  //2s 结束后
        leds_toggle(LEDS_ALL);                      //led 反转
        count ++;
        //将 event_data_ready 事件、count 数据传递给 print 进程
        process_post(&print_process, event_data_ready, &count);
        etimer_reset(&count_timer);                 //复位 count_timer,相当于继续延时 2s
    }

    PROCESS_END();
}

PROCESS_THREAD(print_process, ev, data)             //打印进程执行体
{
    PROCESS_BEGIN();
    while(1)
    {
        PROCESS_WAIT_EVENT_UNTIL(ev == event_data_ready);
        printf("counter is %d\n\r", (*(int *)data));
    }
    PROCESS_END();
}
```

count_process 与 print_process 是如何交互的呢？进程 count_process 一直执行到 PROCESS_WAIT_EVENT_UNTIL(ev==PROCESS_EVENT_TIMER)，此时 etimer 还没到期，进程被挂起。转去执行 print_process，待执行到 PROCESS_WAIT_EVENT_UNTIL(ev==event_data_ready)被挂起(因为 count_process 还没 post 事件)。而后再转去执行系统进程 etimer_process，若检测到 etimer 到期，则继续执行 count_process、count++，并传递事件 event_data_ready 给 print_process，初始化 timer，待执行到 PROCESS_WAIT_EVENT_UNTIL(while 死循环)，再次被挂起。转去执行 print_process，打印 count 的数值，待执行到 PROCESS_WAIT_EVENT_UNTIL(ev==event_data_ready)又被挂起。再次执行系统进程 etimer_process，如此反复执行。

5.6.4 开发步骤

(1) 通过调试转接板将 J-Link 仿真器连接到 PC 和 ZXBee 无线节点板；

(2) 打开例程，将本任务开发例程整个文件夹复制到源代码目录的“\zonesion\example\iar”文件夹下，打开任务工程；

(3) 给连接好的硬件平台起电,将程序下载到 ZXBee 无线节点板中;

(4) 下载完毕后选择 Debug→Go,运行程序;

(5) 在 PC 上打开串口助手或者超级终端,设置接收的波特率为 115 200b/s,数据位为 8,奇偶校验为无,停止位为 1,数据流控制为无。

5.6.5 总结与扩展

程序成功运行后,在串口显示区有如下显示:

```
Starting Contiki 2.6 on STM32F10x
autostart_start: starting process 'count process'
autostart_start: starting process 'print process'
counter is 1
counter is 2
counter is 3
counter is 4
counter is 5
counter is 6
counter is 7
...
```

上面程序的第一行表示 Contiki 操作系统成功移植到 ZXBee 无线节点板中,并成功运行。从第二行开始每看到 ZXBee 无线节点板上的 LED 亮一次,就会看到串口显示区的 counter 数值在增加,表明两个进程都在运行,且实现了进程之间的通信。

5.7 任务 41 Contiki 进程通信高级开发

5.7.1 学习目标

用进程通信实现对硬件驱动与控制。

5.7.2 开发环境

- 硬件:ZXBee 无线节点板,调试转接板,USB MINI 线,J-Link 仿真器,PC;
- 软件:Windows XP/7/8/10,IAR 集成开发环境,串口调试工具。

5.7.3 开发内容

在本任务当中,通过两个进程的共享,实现对开发硬件的驱动。

定义了 buttons_test_process 进程,只需要在 buttons_test_process 的执行体中实现按键位的操作即可。如下为 buttons_test_process 进程的定义:

```
PROCESS(buttons_test_process, "Button Test Process");
```

将 button_test_process 进程设置成自启动:

```
AUTOSTART_PROCESSES(&buttons_test_process);
```

在 PROCESS_BEGIN()与 PROCESS_END()之间编写实现按键位操作的源代码：

```
PROCESS_BEGIN();
while(1) {
    PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event);
                                          //在 sensors_event 到来之前将此进程挂起
    sensor = (struct sensors_sensor * )data;
    if(sensor == &button_1_sensor) {      //如果检测到按钮 1
      PRINTF("Button 1 Press\n\r");
      leds_toggle(LEDS_1) ;               //翻转 D4
    }
    if(sensor == &button_2_sensor) {      //如果检测到按钮 2
      PRINTF("Button 2 Press\n\r");
      leds_toggle(LEDS_2);                //翻转 D5
    }
}
PROCESS_END();
```

buttons_test_process 进程的执行关键就是等待 sensors_event 事件的发生，这个事件一旦发生就将按钮的相关参数传递给 buttons_test_process 进程，该进程在执行时就会判断具体是哪一个按键，从而执行相应的操作。这样分析下来，就要剖析 sensors_event 事件，理解该事件是如何产生的，以及怎样将该事件传递给 buttons_test_process 进程，这个过程的时间是由 sensor.c 文件中的 sensor_process 进程实现的，那么该进程又是如何启动的呢？

从 2.2 节的开发中可知，按下 K1 或者 K2 就会触发一个按键中断服务程序，其实也是一样的，因为按键中断服务程序是最底层的程序，在 Contiki 系统中要实现按键驱动，也脱离不开最底层的按键中断服务程序。在 Contiki 系统中，没有中断的说法，而是各种各样的事件，当某一事件来临，就会执行相应的进程代码。那么底层的按键中断是如何与 sensor_event 事件关联在一起呢？通过分析按键中断服务程序的源代码就可以理解。

```
//按键 1 服务中断程序
void EXTI0_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line0) != RESET)
  {
     if (Bit_RESET == GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_0)) {
      timer_set(&debouncetimer, CLOCK_SECOND / 20);
    } else if(timer_expired(&debouncetimer)) {
        //bv = 1;
        sensors_changed(&button_1_sensor);  //将按键 1 的信息传递给 sensors_changed 函数
    }
    / * Clear the EXTI line 0 pending bit * /
```

```
    EXTI_ClearITPendingBit(EXTI_Line0);
  }
}
//按键 2 服务中断程序
void EXTI1_IRQHandler(void)
{
    if(EXTI_GetITStatus(EXTI_Line1) != RESET) {
    if (Bit_RESET == GPIO_ReadInputDataBit(GPIOA, GPIO_Pin_1)) {
      timer_set(&debouncetimer, CLOCK_SECOND / 20);
    } else if(timer_expired(&debouncetimer)) {
        //bv = 2;
        sensors_changed(&button_2_sensor); //将按键 2 的信息传递给 sensors_changed 函数
    }
    / * Clear the EXTI line 0 pending bit * /
    EXTI_ClearITPendingBit(EXTI_Line1);
  }
}
```

通过上述代码可知，中断服务程序与 2.2 节的中断服务程序大同小异，只不过是在其中调用了 sensors_changed 函数，那么关键点就是这个函数，继续剖析，将 sensors_changed 函数源代码展开：

```
void
sensors_changed(const struct sensors_sensor * s)
{
  sensors_flags[get_sensor_index(s)] | = FLAG_CHANGED;     //标记改变的状态
  process_poll(&sensors_process);                    //更改 sensors_process 进程的优先级为 1
}
```

在上述代码中将 sensors_process 进程的优先级进行了修改，其实就是相当于启动了该进程：

```
PROCESS_THREAD(sensors_process, ev, data)
{
  static int i;
  static int events;
  PROCESS_BEGIN();
  sensors_event = process_alloc_event();
  for(i = 0; sensors[i] != NULL; ++i) {       //初始化状态
    sensors_flags[i] = 0;
    sensors[i] -> configure(SENSORS_HW_INIT, 0);
  }
  num_sensors = i;
  while(1) {
    PROCESS_WAIT_EVENT();
```

```
    do {
      events = 0;
      for(i = 0; i < num_sensors; ++i) {
    if(sensors_flags[i] & FLAG_CHANGED) {//如果状态进行了改变
    //将 sensor_event 事件广播给所有进程,并传递 sensor[i]的数据给所有进程
      if(process_post(PROCESS_BROADCAST, sensors_event, (void * )sensors[i]) == PROCESS_
ERR_OK) {
        PROCESS_WAIT_EVENT_UNTIL(ev == sensors_event);
      }
      sensors_flags[i] &= ~FLAG_CHANGED;
      events++;
    }
      }
    } while(events);
  }

  PROCESS_END();
}
```

按键位进程工作的部分流程如图 5.19 所示。

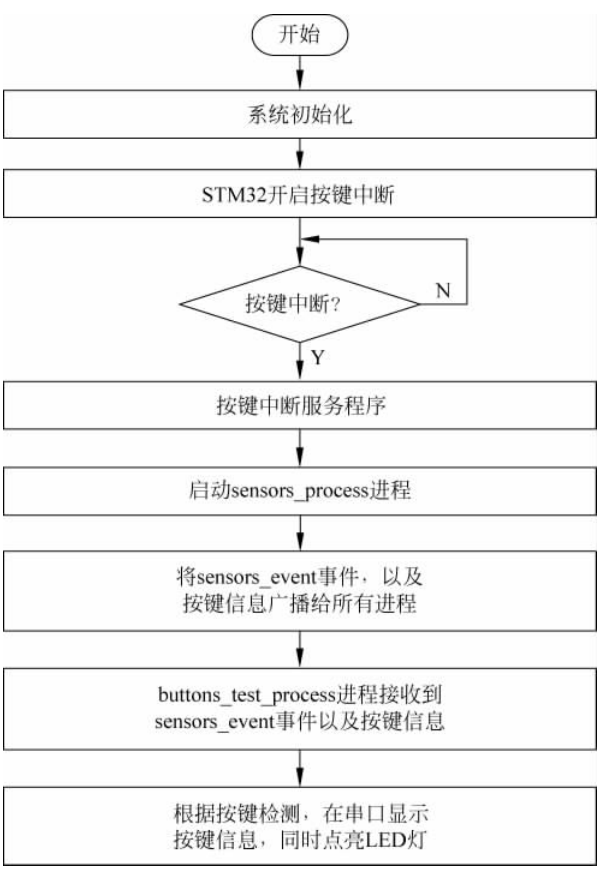

图 5.19　按键位进程工作的部分流程图

5.7.4 开发步骤

(1) 通过调试转接板将 J-Link 仿真器连接到 PC 和 ZXBee 无线节点板;

(2) 打开例程,将本任务开发例程整个文件夹复制到源代码目录的"\zonesion\example\iar"文件夹下,打开任务工程;

(3) 给连接好的硬件平台起电,将程序下载到 ZXBee 无线节点板中;

(4) 下载完毕后选择 Debug→Go,运行程序;

(5) 在 PC 上打开串口助手或者超级终端,设置接收的波特率为 115 200b/s,数据位为 8,奇偶校验为无,停止位为 1,数据流控制为无。

5.7.5 总结与扩展

程序成功运行后,此时在 ZXBee 无线节点板上按下 K1 和 K2,可观察到 D4 和 D5 灯点亮,同时在串口显示区有如下显示:

```
Starting Contiki 2.6 on STM32F10x
autostart_start: starting process 'Button Test Process'
Button 1 Press
Button 2 Press
```

串口显示区的 Starting Contiki 2.6 on STM32F10x 表明 Contiki 操作系统成功移植到 ZXBee 无线节点板中,Button 1 Press 表明 button_test_process 进程成功被调用,并执行相应按键检测操作。

5.8 任务 42 定时器驱动开发

5.8.1 学习目标

- 理解 Contiki 操作系统的定时器基本工作原理;
- 掌握将进程和定时器结合开发项目。

5.8.2 开发环境

- 硬件: ZXBee 无线节点板,调试转接板,USB MINI 线,J-Link 仿真器,PC;
- 软件: Windows XP/7/8/10,IAR 集成开发环境,串口调试工具。

5.8.3 原理学习

Contiki 系统提供一组 timer 库,除了用于 Contiki 系统本身,也可以用于应用程序。timer 库包含一些实用功能,例如检查一个时间周期是否过去了、在预定时间将系统从低功耗模式唤醒,以及实时任务的调度。定时器也可在应用程序中使用,以使系统与其他任务协调工作,或使系统在恢复运行前的一段时间内进入低功耗模式。

Contiki 有一个时钟模块和一组定时器模块: timer、stimer、ctimer、etimer、rtimer。本

任务中应用到的是 etimer 定时器模块，etimer 库主要用于调度事件按预定时间周期来触发 Contiki 系统的进程，可使得进程等待一段时间，以便于系统的其他功能运行，或这段时间让系统进入低功耗模式。

etimer 提供时间事件，当 etimer 时间到期，会给相应的进程传递 PROCEE_EVENT_TIMER 事件，从而使该进程运行。

```
struct etimer{
struct timer timer;     //timer 包含起始时刻和间隔时间，故此 timer 只记录到期时间
                        //通过比较到到期时间和新的当前时钟，从而判断是否到期
struct etimer * next;
struct process * p;
};
```

Contiki 系统有一个全局静态变量 timerlist，保存 etimer 链表，从第一个 etimer 到最后 NULL。

在本任务中，定义了 clock_test_process，只需要在 clock_test_process 的执行体中实现定时器的操作即可。clock_test_process 进程的定义如下：

```
PROCESS(clock_test_process, "Clock test process");
```

将 clock_test_process 进程设置成自启动：

```
AUTOSTART_PROCESSES(&clock_test_process);
```

在 PROCESS_BEGIN()与 PROCESS_END()之间编写实现定时器操作的源代码：

```
PROCESS_BEGIN();
etimer_set(&et, 2 * CLOCK_SECOND);
PROCESS_YIELD();
printf("Clock tick and etimer test, 1 sec ( % u clock ticks):\n\r", CLOCK_SECOND);
i = 0;
while(i < 10) {
    etimer_set(&et, CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    etimer_reset(&et);
    count = clock_time();
    printf(" % u ticks\n\r", count);
    leds_toggle(LEDS_RED);
    i++;
}
printf("Clock seconds test (5s):\n\r");
i = 0;
while(i < 10) {
    etimer_set(&et, 5 * CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
```

```
        etimer_reset(&et);
        sec = clock_seconds();
        printf(" % lu seconds\n\r", sec);
        leds_toggle(LEDS_GREEN);
        i++;
    }
    printf("Done!\n\r");
    PROCESS_END();
```

5.8.4 开发步骤

(1) 通过调试转接板将 J-Link 仿真器连接到 PC 和 ZXBee 无线节点板;

(2) 打开例程,将本任务开发例程整个文件夹复制到源代码目录的"\zonesion\example\iar"文件夹下,打开任务工程;

(3) 给连接好的硬件平台起电,将程序下载到 ZXBee 无线节点板中;

(4) 下载完毕后选择 Debug→Go,运行程序;

(5) 在 PC 上打开串口助手或者超级终端,设置接收的波特率为 115 200b/s,数据位为 8,奇偶校验为无,停止位为 1,数据流控制为无。

5.8.5 总结与扩展

程序成功运行后,此时在串口显示区有如下显示:

```
Starting Contiki 2.6 on STM32F10x
autostart_start: starting process 'Clock test process'
Clock tick and etimer test, 1 sec (100 clock ticks):
300 ticks
400 ticks
500 ticks
600 ticks
700 ticks
800 ticks
900 ticks
1000 ticks
1100 ticks
1200 ticks
Clock seconds test (5s):
17 seconds
22 seconds
27 seconds
32 seconds
37 seconds
42 seconds
47 seconds
52 seconds
```

```
57 seconds
62 seconds
Done!
```

5.9 任务 43 基于 Contiki 的 LCD 驱动开发

5.9.1 学习目标

- 理解 LCD 的工作原理；
- 学会基于 Contiki 操作系统的 LCD 的驱动开发。

5.9.2 开发环境

- 硬件：ZXBee 无线节点板，调试转接板，USB MINI 线，J-Link 仿真器，PC；
- 软件：Windows XP/7/8/10，IAR 集成开发环境，串口调试工具。

5.9.3 原理学习

本任务实现基于 Contiki 系统的 LCD 显示。要在 Contiki 系统上实现 LCD 的显示，主要分成两个内容：LCD 驱动的实现；LCD 显示进程的实现。下面结合本次开发例程的源代码分别解析 LCD 驱动的实现以及 LCD 显示进程的实现过程。

1. LCD 驱动的实现

ZXBee 无线节点板利用 SPI 总线与 LCD 进行通信，驱动 LCD 需要 STM32 通过 SPI 总线向其发送相关初始化指令，因此 LCD 驱动的实现分为两个步骤：SPI 总线初始化，LCD 模块初始化。

1）SPI 总线初始化

SPI 总线由 SCK、MISO 和 MOSI 等引脚组成，要完成 SPI 总线初始化首先需要实现其相关引脚的 I/O 口初始化，然后再配置 SPI 总线。本任务中的源代码采用 STM32 的官方库实现，下面是 SPI 总线初始化的源代码：

```
void SPI_LCD_Init(void)
{
  SPI_InitTypeDef SPI_InitStructure;
  GPIO_InitTypeDef GPIO_InitStructure;
#if USE_SPI

  RCC_APB1PeriphClockCmd(RCC_APB1Periph_SPI2, ENABLE);  //开启 SPI 时钟
#endif
//开启 SQI 总线 I/O 引脚的时钟
  RCC_APB2PeriphClockCmd( RCC_APB2Periph_GPIO_RS |RCC_APB2Periph_GPIO_REST |
                          RCC_APB2Periph_GPIO_CS, ENABLE);
#if USE_SPI
```

```
  /* LCD SPI 总线的 SCK, MISO 及 MOSI 引脚配置 */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15 | GPIO_Pin_13;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
  GPIO_Init(GPIOB, &GPIO_InitStructure);
#else

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_15 | GPIO_Pin_13;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_10MHz;
  GPIO_Init(GPIOB, &GPIO_InitStructure);
  SPI_LCD_CLK(1);

#endif

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_REST;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIO_REST, &GPIO_InitStructure);

  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_RS;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIO_RS, &GPIO_InitStructure);

  /* Configure I/O for Flash Chip select */
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_CS;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIO_CS, &GPIO_InitStructure);

  SPI_LCD_CS_HIGH();
#if USE_SPI
  /* LCD 的 SPI 总线配置 */
  SPI_InitStructure.SPI_Direction = SPI_Direction_1Line_Tx;
  SPI_InitStructure.SPI_Mode = SPI_Mode_Master;
  SPI_InitStructure.SPI_DataSize = SPI_DataSize_8b;
  SPI_InitStructure.SPI_CPOL = SPI_CPOL_High;
  SPI_InitStructure.SPI_CPHA = SPI_CPHA_2Edge;
  SPI_InitStructure.SPI_NSS = SPI_NSS_Soft;
  SPI_InitStructure.SPI_BaudRatePrescaler = SPI_BaudRatePrescaler_4;
  SPI_InitStructure.SPI_FirstBit = SPI_FirstBit_MSB;
  SPI_InitStructure.SPI_CRCPolynomial = 7;
  SPI_Init(LCD_SPIx, &SPI_InitStructure);
  /* 使能 SPI 总线 */
  SPI_Cmd(LCD_SPIx, ENABLE);
#endif
}
```

2）LCD 模块初始化

SPI 总线驱动实现完成后，就需要对 LCD 模块的初始化。要完成对 LCD 模块的初始化，只需要向其发送相关的初始化指令即可。下面便是 LCD 模块初始化的源代码：

```
void lcd_initial()
{
    int i;
    SPI_LCD_REST_HIGH();
    delay_ms(10);
    SPI_LCD_REST_LOW();
    delay_ms(10);
    SPI_LCD_REST_HIGH();
    delay_ms(10);

    write_command(0x11);
    delay_ms(10);
    //ST7735R Frame Rate
    write_command(0xB1);
    write_data(0x01); write_data(0x2C); write_data(0x2D);
    write_command(0xB2);
    write_data(0x01); write_data(0x2C); write_data(0x2D);
    write_command(0xB3);
    write_data(0x01); write_data(0x2C); write_data(0x2D);
    write_data(0x01); write_data(0x2C); write_data(0x2D);
    write_command(0xB4);
    write_data(0x07);

    //ST7735R Power Sequence
    write_command(0xC0);
    write_data(0xA2); write_data(0x02); write_data(0x84);
    write_command(0xC1); write_data(0xC5);
    write_command(0xC2);
    write_data(0x0A); write_data(0x00);
    write_command(0xC3);
    write_data(0x8A); write_data(0x2A);
    write_command(0xC4);
    write_data(0x8A); write_data(0xEE);
    write_command(0xC5);
    write_data(0x0E);
    write_command(0x36);
    #if ROTATION == 90
        write_data(0xa0 | SBIT_RGB);
    #elif ROTATION == 180
        write_data(0xc0 | SBIT_RGB);
    #elif ROTATION == 270
        write_data(0x60 | SBIT_RGB);
    #else
        write_data(0x00 | SBIT_RGB);
    #endif
    #if ROTATION == 90 || ROTATION == 270
      write_command(0x2a);
      write_data(0x00);write_data(0x00);
      write_data(0x00);write_data(0x9f);
      write_command(0x2b);
```

```
        write_data(0x00);write_data(0x00);
        write_data(0x00);write_data(0x7f);
      #else
        write_command(0x2a);
        write_data(0x00);write_data(0x00);
        write_data(0x00);write_data(0x7f);
        write_command(0x2b);
        write_data(0x00);write_data(0x00);
        write_data(0x00);write_data(0x9f);
      #endif

        write_command(0xe0);
        write_data(0x0f); write_data(0x1a);
        write_data(0x0f); write_data(0x18);
        write_data(0x2f); write_data(0x28);
        write_data(0x20); write_data(0x22);
        write_data(0x1f); write_data(0x1b);
        write_data(0x23); write_data(0x37);
        write_data(0x00); write_data(0x07);
        write_data(0x02); write_data(0x10);
        write_command(0xe1);
        write_data(0x0f); write_data(0x1b);
        write_data(0x0f); write_data(0x17);
        write_data(0x33); write_data(0x2c);
        write_data(0x29); write_data(0x2e);
        write_data(0x30); write_data(0x30);
        write_data(0x39); write_data(0x3f);
        write_data(0x00); write_data(0x07);
        write_data(0x03); write_data(0x10);
        write_command(0xF0);
        write_data(0x01);
        write_command(0xF6);
        write_data(0x00);
        write_command(0x3A);
        write_data(0x05);
        write_command(0x29);
    }
```

通过上述源代码可知，LCD 模块在初始化时通过调用 write_command()和 write_data()两个方法实现 STM32 向 LCD 模块发送数据，同时这两个方法是实现 LCD 内容显示的核心方法，LCD 显示内容之前，需要先通过 write_command()方法向 LCD 模块发送相关的指令，然后再调用 write_data()方法向 LCD 模块发送 LCD 显示的数据内容。下面是这两个方法的源代码：

```
/* 指令写函数 */
void write_command(unsigned char c)
{
  SPI_LCD_RS_LOW();
  SPI_LCD_CS_LOW();
  /* 发送写使能指令 */
#if USE_SPI

  SPI_I2S_SendData(LCD_SPIx, c);

  while (SPI_I2S_GetFlagStatus(LCD_SPIx, SPI_I2S_FLAG_TXE) == RESET);
#else
{
    int i;
    for (i = 0; i < 8; i++) {
        SPI_LCD_DAT(0x01&(c >>(7 - i)));
        SPI_LCD_CLK(0);
        SPI_LCD_CLK(1);
    }
}
#endif

  SPI_LCD_CS_HIGH();
}
/* 数据写函数 */
void write_data(unsigned char c)
{
  SPI_LCD_RS_HIGH();
  SPI_LCD_CS_LOW();
#if USE_SPI

  SPI_I2S_SendData(LCD_SPIx, c);

  while (SPI_I2S_GetFlagStatus(LCD_SPIx, SPI_I2S_FLAG_TXE) == RESET);
#else
{
    int i;
    for (i = 0; i < 8; i++) {
        SPI_LCD_DAT(0x01&(c >>(7 - i)));
        SPI_LCD_CLK(0);
        SPI_LCD_CLK(1);
    }
}
#endif

  SPI_LCD_CS_HIGH();
}
```

2. LCD 显示进程

实现了 LCD 的驱动之后，就需要实现 LCD 显示的进程。在本任务中，让 LCD 上分行显示不同颜色的“This is a LCD example 2013.10.17”，下面是 LCD 显示进程的实现源代码：

```
PROCESS(lcd_process, "lcd process");              //定义 LCD 显示进程
AUTOSTART_PROCESSES(&lcd_process);                //将 LCD 显示进程设置成自动启动
PROCESS_THREAD(lcd_process, ev, data)             //进程执行体
{
  PROCESS_BEGIN();
  char * tail = "LCD process example";
  char * head = "Contiki 2.6";
  Display_Clear_Rect(0, 0, LCDW, 12, 0xffff);     //将 LCD 顶端的 160 * 12 的区域清屏成白色
  Display_ASCII6X12((LCDW - strlen(head) * 6)/2, 1, 0x0000, head);
                                                  //将 LCD 顶端居中显示 head 的内容
  //以不同行、不同颜色显示 This is a LCD example 2013.10.17
  Display_ASCII6X12(1,24, 0xf800, "This");
  Display_ASCII6X12(1,36, 0x07e0, "is");
  Display_ASCII6X12(1,48, 0x0f0f0, "a");
  Display_ASCII6X12(1,60, 0xffff, "LCD");
  Display_ASCII6X12(1,72, 0x0000, "example");
  Display_ASCII6X12(1,84, 0x0000, "2013.10.17");
  //在 LCD 的底部 160 * 12 的区域清屏成白色
  Display_Clear_Rect(0, LCDH - 12, LCDW, 12, 0xffff);
  //LCD 底部居中黑色字体显示 LCD process example
  Display_ASCII6X12((LCDW - strlen(tail) * 6)/2, LCDH - 12, 0x0000, tail);
  PROCESS_END();
}
```

上述内容中实现了 LCD 显示进程，同时也可以看到在实现的过程调用了 Display_ASCII6X12 方法，这个方法的功能就是在 LCD 上的任意位置显示 6×12 点阵的字符，同时可以指定显示字符的颜色，下面便是这个方法的实现源代码：

```
void Display_ASCII6X12(unsigned int x0,unsigned int y0,unsigned int co, char * s)
{
    unsigned char ch = * s;
    unsigned char dot;
    int i, j;
#define CWIDTH 6
    while (ch != 0) {
        if (ch < 0x20 || ch > 0x7e) {
          ch = 0x20;
        }
        ch -= 0x20;
        for (i = 0; i < 12; i++) {
            dot = nAsciiDot6x12[ch * 12 + i];
            for (j = 0; j < 6; j++) {
```

```
                if (dot&0x80)Output_Pixel(x0 + j, y0 + i, co);
                dot <<= 1;
              }
          }
          x0 += CWIDTH;
          ch = * ++s;
      }
  }
```

根据 LCD 的工作原理可知，要在 LCD 上显示 6×12 点阵的字符，必须要在源代码中生成一个 6×12 点阵的字符库，否则将不能在 LCD 上显示正确的字符信息，下面是 6×12 点阵的部分 ASCII 字符库：

```
//------------------- ASCII 字模的数据表 -------------------------//
//码表从 0x20～0x7e                                               //
//----------------------------------------------------------------//
const unsigned char nAsciiDot6x12[] =                             //ASCII
{
    0x00,0x00,0x00,0x00,0x00,0x00,0x00,0x00,                      //- -
    0x00,0x00,0x00,0x00,
    0x00,0x10,0x10,0x10,0x10,0x10,0x00,0x00,                      //-!-
    0x10,0x00,0x00,0x00,
    0x00,0x6C,0x48,0x48,0x00,0x00,0x00,0x00,                      //-"-
    0x00,0x00,0x00,0x00,
    …
    0x00,0x38,0x44,0x44,0x44,0x44,0x44,0x44,                      //-0-
    0x38,0x00,0x00,0x00,
    0x00,0x30,0x10,0x10,0x10,0x10,0x10,0x10,                      //-1-
    0x7C,0x00,0x00,0x00,
    …
    0x00,0x30,0x10,0x28,0x28,0x28,0x7C,0x44,                      //-A-
    0xEC,0x00,0x00,0x00,
    0x00,0xF8,0x44,0x44,0x78,0x44,0x44,0x44,                      //-B-
    0xF8,0x00,0x00,0x00,
    0x00,0x3C,0x44,0x40,0x40,0x40,0x40,0x44,                      //-C-
    …
};
```

5.9.4 开发步骤

（1）正确地通过调试转接板将 J-Link 仿真器连接到 PC 和 ZXBee 无线节点板，将 LCD 正确插到 ZXBee 无线节点板上；

（2）打开例程，将本任务开发例程整个文件夹复制到源代码目录的“\zonesion\example\iar”文件夹下，打开任务工程；

（3）给连接好的硬件平台起电，将程序下载到 ZXBee 无线节点板中；

（4）下载完毕后选择 Debug→Go，运行程序；

5.9.5 总结与扩展

程序成功运行后，可观察到在 ZXBee 无线节点板上的 LCD 显示如图 5.20 所示的内容。

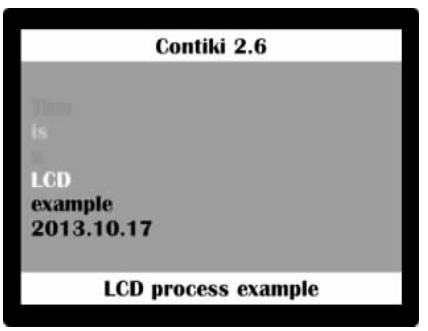

图 5.20 LCD 屏显示