基于 IAR 的 Contiki 系统在 STM32 上的移植

摘要

本文对 Contiki 操作系统进行简介和概述，说明了移植前的准备工作，对并 Contiki 源代码移植相关代码进行分析，指出关键点。然后给出了移植的具体代码实现，并建立 IAR 工程项目，给出工程代码结构和参数配置，最后编写一个测试程序，对移植后的代码进行测试，通过串口打印信息。

1. Contiki 操作系统简介

“Contiki 是一个小型的，开源的，极易移植的多任务操作系统。它专门设计以适用于一系列的内存优先的网络系统，包括从 8 位电脑到微型控制器的嵌入系统。它的名字来自于托尔·海尔达尔的康提基号。Contiki 只需几 kilobyte 的代码和几百字节的内存就能提供多任务环境和内建 TCP/IP 支持。”来自维基百科。（认真读才发现维基百科说的不是人话）

从这段介绍中可以得知 contiki 操作系统的三大特点——小型、开源、极易移植。和绝大多数开源的嵌入式操作系统不同，例如 uCOS 和 FreeRTOS，contiki 非常容易移植，使用事件驱动机制（先这么理解吧），运行时占用的内存很小。

虽然国内关于 contiki 的资料非常少，但是通过阅读 contiki 的例子和文档，还是可以很容易的完成移植工作。

2. 移植前的准备工作

首先建立一个最简单的工程。一个最简单的任务莫过于 LED 闪烁了，从学习 51 单片机开始，到 AVR，到 ARM，从移植 uCOS 到移植 contiki。LED 闪烁无疑是最棒的任务。假设这个任务就是 LED 点亮 1 秒，然后让 LED 熄灭 1 秒。

Contiki 的采用事件驱动机制，那么如何才能够产生“事件“呢。答案只有两个，第一，通过时钟定时，定时事件到就产生一个事件；第二，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。先来回顾一下 uCOS 在 STM32 移植，uCOS 的移植也就是做了两件事情，第一，在 PendSV 这个异常中断中，保存上下文；第二，使用 systick 提供系统时钟。由于 contiki 是非抢占的操作系统，所以移植时并不需要 PendSV 中保存上下文。那么剩下的时钟一定是需要的，移植 contiki 的移植重点就应该在 systick 上。（我个人觉得，如果有时间有精力的话，应该多学点东西，可以触类旁通。）

我个人的习惯，先上全部的代码，给大家一个整体的印象。

```
#include "stm32f10x.h"
#include <stdint.h>
#include <stdio.h>
#include <debug-uart.h>
#include <clock.h>
#include <sys/process.h>
#include <sys/procinit.h>
#include <etimer.h>
#include <sys/autostart.h>

unsigned int idle_count = 0;
void led_init();
```

```
PROCESS(blink_process, "Blink");
AUTOSTART_PROCESSES(&blink_process);
PROCESS_THREAD(blink_process, ev, data)
{
  PROCESS_BEGIN();
  while(1)
  {
    static struct etimer et;
    etimer_set(&et, CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    //打开LED
    GPIO_ResetBits(GPIOC,GPIO_Pin_6);
    printf("LED ON\r\n");
    etimer_set(&et, CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    //关闭LED
    GPIO_SetBits(GPIOC,GPIO_Pin_6);
    printf("LED OFF\r\n");
  }
   PROCESS_END();
}

int main()
{
  dbg_setup_uart();
  led_init();
  printf("Initialising\r\n");
  clock_init();
```

```
  process_init();
  process_start(&etimer_process, NULL);
  autostart_start(autostart_processes);
  //process_start(&blink_process,NULL);
  printf("Processes running\r\n");
  while(1) {
    do
    {
    }
    while(process_run() > 0);
    idle_count++;
    /* Idle! */
    /* Stop processor clock */
    /* asm("wfi"::); */
  }
  return 0;
}
void led_init()
{
  GPIO_InitTypeDef GPIO_InitStructure;
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOC,ENABLE);
  //PC6 推挽输出
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_6;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_Out_PP;
  GPIO_Init(GPIOC, &GPIO_InitStructure);
}
```

3. 从源代码获取移植线索

阅读contiki-2.5 源码中，stm32移植的相关内容分散在两个文件夹中，第一，cpu\arm\stm32f103，这个文件夹存放的 stm32 移植的相关文件；第二，platform\stm32test，这个文件夹中有一个不是那么完整的例子（主要是里面没有一个任务）。具体的源码如下：

```
#include <stm32f10x_map.h>
#include <stm32f10x_dma.h>
#include <gpio.h>
#include <nvic.h>
#include <stdint.h>
#include <stdio.h>
#include <debug-uart.h>
#include <sys/process.h>
#include <sys/procinit.h>
#include <etimer.h>
#include <sys/autostart.h>
#include <clock.h>

unsigned int idle_count = 0;

int main()
{
  dbg_setup_uart();
  printf("Initialising\n");
```

```
    clock_init();
    process_init();
    process_start(&etimer_process, NULL);
    autostart_start(autostart_processes);
    printf("Processes running\n");
    while(1) {
      do {
      } while(process_run() > 0);
      idle_count++;
      /* Idle! */
      /* Stop processor clock */
      /* asm("wfi"::); */
    }
    return 0;
  }
```

我们来简单的分析一下，首先文件中包含了一些头文件。看着有点熟悉，应该是 V2.0 库的头文件，后面的移植工作会全部替换掉，使用 V3.4 的库文件。在 main 函数中，第一步初始化串口并通过串口发送某些信息，如果熟悉 smt32 的话，配置一个串口不难。接下来，初始化时钟，通过跟踪源代码，发现 clock_init 函数位于 cpu\arm\stm32f103 文件夹中的 clock 文件夹中。具体的函数如下

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

这段代码的原理也非常的简单，初始化 systick 定时器。其功能是每秒发生 CLOCK_SECOND 次溢出。配置了 systick 也少不了 systick 中断了，systick 的

中断的源码如下：

```
void SysTick_handler(void) __attribute__ ((interrupt));


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("%d,%d\n", clock_time(),etimer_next_expiration_time    ()); */
  }
  if (--second_countdown == 0) {
    current_seconds++;
    second_countdown = CLOCK_SECOND;
  }
}
```

在 systick 中断中不断更新了 etimer，有了时钟 contiki 就可以安全运行了。但是非常遗憾的是，全部的移植文件都使用的操作寄存器的方法，对于我来说，还是使用库函数比较舒服，在移植的过程中会慢慢修改。

4. Contiki 移植

准备的时间虽然有点长，但是是必须的。移植的过程却是非常的简单。

先在 clock 源文件中添加头文件

```
#include "stm32f10x.h"
#include "stm32f10x_it.h"
```

删除原来的

```
#include <stm32f10x_map.h>
#include <nvic.h>
```

把 systick 初始化改成

```
void clock_init()
{
  if (SysTick_Config(SystemCoreClock / CLOCK_SECOND))
  {
    while (1);
  }
}
```

把 systick 中断改成

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

最后，把 stm32f10x_it.c 的 void SysTick_Handler(void){}删除。（为什么，函数只能出现一次吗，这里出现了其他地方就不能出现了，这个其实和 uCOS 的移植一样的。） 到此就基本完成了 contiki 的移植，非常的简单。

再来配置一下 debug 接口。配置串口位于 debug_uart 文件中，我把原代

码中的 DMA 相关代码删了个精光，只有串口初始化和 fputc 函数。具体的代码如下

```
void
dbg_setup_uart_default()
{
  USART_InitTypeDef USART_InitStructure;
  GPIO_InitTypeDef GPIO_InitStructure;
   //使能GPIOA时钟
  RCC_APB2PeriphClockCmd(RCC_APB2Periph_GPIOA \
                         | RCC_APB2Periph_USART1 ,ENABLE);
  //PA9 TX1  复用推挽输出
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_9;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_AF_PP;
  GPIO_Init(GPIOA, &GPIO_InitStructure);
  //PA10 RX1  浮动输入
  GPIO_InitStructure.GPIO_Pin = GPIO_Pin_10;
  GPIO_InitStructure.GPIO_Speed = GPIO_Speed_50MHz;
  GPIO_InitStructure.GPIO_Mode = GPIO_Mode_IN_FLOATING;
  GPIO_Init(GPIOA, &GPIO_InitStructure);

  USART_InitStructure.USART_BaudRate = 9600;
  USART_InitStructure.USART_WordLength = USART_WordLength_8b;
  USART_InitStructure.USART_StopBits = USART_StopBits_1;
  USART_InitStructure.USART_Parity = USART_Parity_No;
  USART_InitStructure.USART_HardwareFlowControl =
  USART_HardwareFlowControl_None;
  USART_InitStructure.USART_Mode = USART_Mode_Rx | USART_Mode_Tx;
```

```
  USART_Init(USART1, &USART_InitStructure);
  //使能USART1
  USART_Cmd(USART1, ENABLE);
}

int fputc(int ch, FILE * f)
{
  USART_SendData(USART1, (uint8_t)ch);
  while(USART_GetFlagStatus(USART1, USART_FLAG_TXE) == RESET );
  return ch;
}
```

5. 新建一个测试任务

通过上网搜索和阅读书籍，我写了以下任务。

```
PROCESS(blink_process, "Blink");
AUTOSTART_PROCESSES(&blink_process);
PROCESS_THREAD(blink_process, ev, data)
{
  PROCESS_BEGIN();
  while(1)
  {
    static struct etimer et;
    etimer_set(&et, CLOCK_SECOND);
    PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
    //打开LED
    GPIO_ResetBits(GPIOC,GPIO_Pin_6);
```

```
      printf("LED ON\r\n");
      etimer_set(&et, CLOCK_SECOND);
      PROCESS_WAIT_EVENT_UNTIL(etimer_expired(&et));
      //关闭LED
      GPIO_SetBits(GPIOC,GPIO_Pin_6);
      printf("LED OFF\r\n");
    }
     PROCESS_END();
  }
```

该任务是从 contiki-2.5 中一个例子修改而来的。任务非常的简单，打开 LED，通过串口发送提示信息，然后关闭 LED，通过串口发送提示信息。具体的代码不做过多的解释，因为前言中给出了链接，博主比我分析的恰当。在这里只说两点，

第一 PROCESS(blink_process, "Blink");相关于函数的声明

第二 AUTOSTART_PROCESSES(&blink_process);是指该任务自动启动，当然也可以调用 process_start 函数启动任务。

AUTOSTART_PROCESSES其实也是一个宏东定义，

```
#if ! CC_NO_VA_ARGS
#if AUTOSTART_ENABLE
#define AUTOSTART_PROCESSES(...)                        \
struct process * const autostart_processes[] = {__VA_ARGS__, NULL}
#else //AUTOSTART_ENABLE
#define AUTOSTART_PROCESSES(...)                        \
extern int _dummy
#endif //AUTOSTART_ENABLE
#else
#error "C compiler must support __VA_ARGS__ macro"
#endif
```

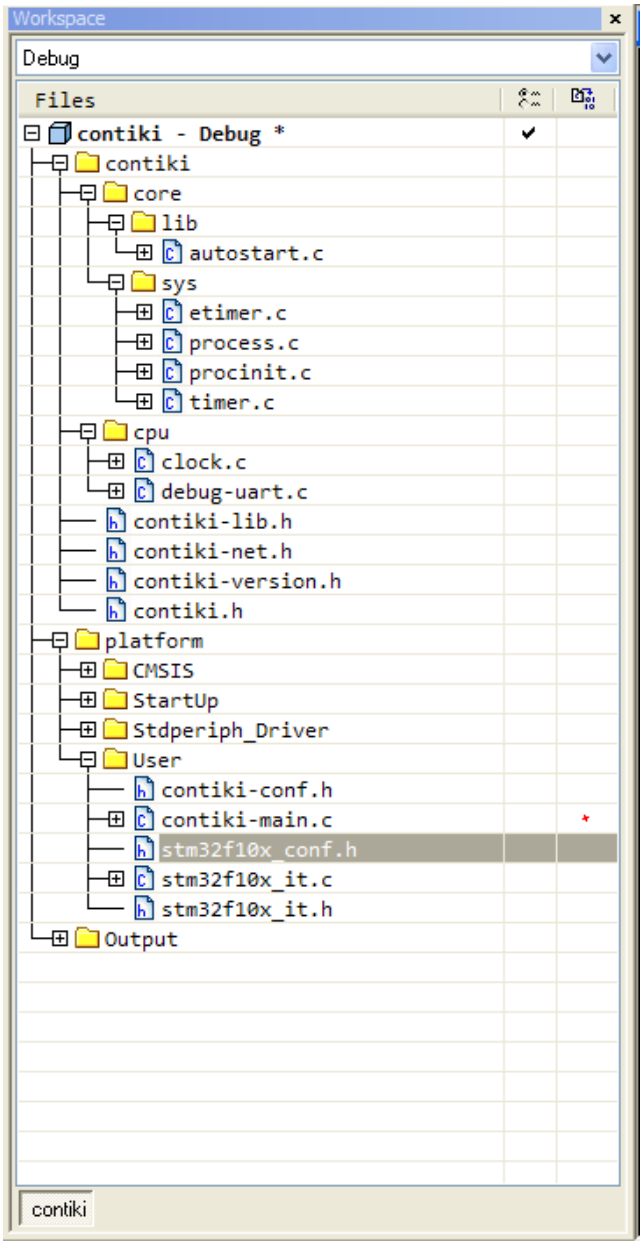
要想使用它的话，还需要再任何一个地方添加

```
#define AUTOSTART_ENABLE 1
```

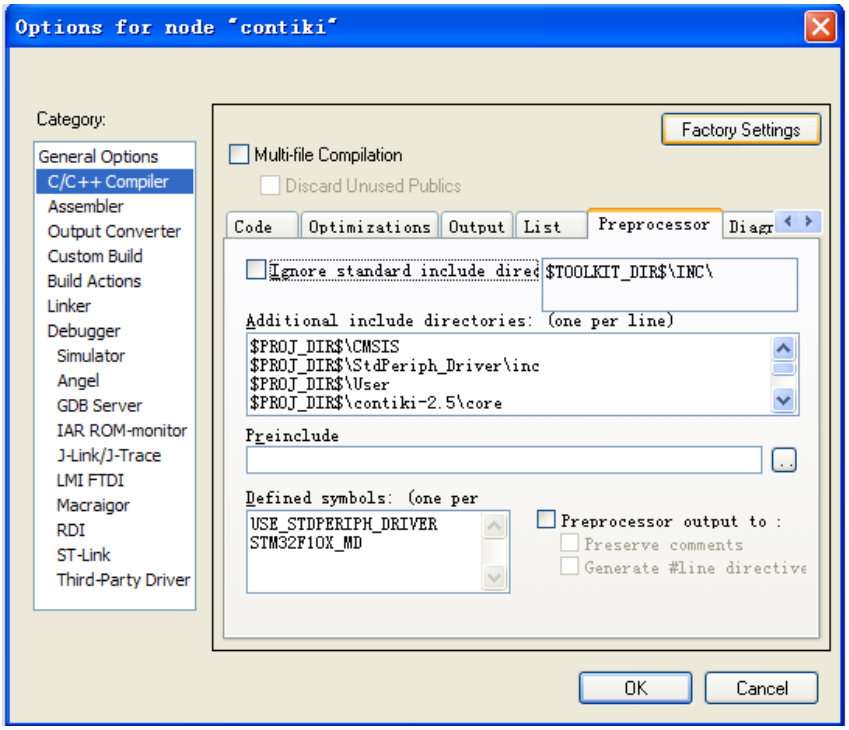
最后请大家不要忘记 LED 相关 IO 口的操作。请查看前文代码。

6. 建立 IAR 工程进行测试

先给出 contiki 的 IAR 工程目录和文件目录



再来一个头文件包含路径



$PROJ_DIR$\CMSIS

$PROJ_DIR$\StdPeriph_Driver\inc

$PROJ_DIR$\User

$PROJ_DIR$\contiki-2.5\core

$PROJ_DIR$\contiki-2.5\core\sys

$PROJ_DIR$\contiki-2.5\core\lib

$PROJ_DIR$\contiki-2.5\cpu

其他的也不做过多说明，我想相信聪明的你找找就知道了。

如果移植顺利的话，就可以看到以下实验结果。写到这里你会发现，contiki 的移植还是非常简单的。

- **参考资料**

[Contiki] Contiki operating system:

http://www.contiki-os.org/

[contiki/6Lowpan]：IPv6/Contiki 6Lowpan/Zigbee 物联网开发论坛

http://www.iotdev.net

[6LoWPAN] IETF 6LoWPAN Working Group:

http://datatracker.ietf.org/wg/6lowpan/charter/

[CoRE] IETF CoRE Working Group:

http://datatracker.ietf.org/wg/core/charter/