

The background features three large, overlapping blue circles of varying sizes. Thin blue lines connect the centers of these circles, forming a triangular network. The circles have a layered, concentric appearance with different shades of blue.

# Contiki 学习笔记

博客文章整理

Contiki 初体验|数据结构|实例|主函数剖析|进阶学习|编程|文件系统

**Jelline**

**2012/3/15**

[Contiki学习笔记：目录](#) (2012-01-07 17:02)

标签： [Coffee](#) [Contiki](#) [Protothread](#) [学习笔记](#) [目录](#) 分类： [Contiki](#)

### 摘要：

本文将 Contiki 学习笔记整理成篇，便于索引。文章的顺序系推荐阅读的顺序。

Contiki 初之体验：

[Contiki学习笔记：开发环境搭建及学习资料](#)

[Contiki学习笔记：事件驱动机制和protothread机制](#)

[Contiki学习笔记：实例hello world剖析](#)

Contiki 主要数据结构：

[Contiki学习笔记：主要数据结构之进程](#)

[Contiki学习笔记：主要数据结构之事件](#)

[Contiki学习笔记：主要数据结构之etimer](#)

[Contiki学习笔记：进程、事件、etimer关系](#)

Contiki 实例：

[Contiki学习笔记：创建两个交互进程](#)

Contiki 主函数剖析：

[Contiki学习笔记：main函数剖析](#)

[Contiki学习笔记：启动一个进程process\\_start](#)

[Contiki学习笔记：系统进程etimer\\_process](#)

[Contiki学习笔记：深入理解process\\_run函数](#)

Contiki 进阶学习：

[Contiki学习笔记：新事件产生及事件处理](#)

[Contiki学习笔记：时钟中断处理程序Systick\\_isr](#)

[Contiki学习笔记：protothread状态](#)

[Contiki学习笔记：进程状态](#)

Contiki 编程：

[Contiki学习笔记：编程模式](#)

[Contiki学习笔记：应用编程接口API](#)

Contiki 文件系统：

[Contiki学习笔记：Coffee文件系统概述及其学习资料](#)

[Contiki学习笔记：Coffee遇到若干问题\(解决及待解决\)](#)

[Contiki学习笔记：Coffee文件系统移植](#)

[Contiki学习笔记：Coffee文件组织及若干数据结构](#)

[Contiki学习笔记：Coffee文件系统flags标志位](#)

[Contiki学习笔记：Coffee文件系统file与file header及其联系](#)

[Contiki学习笔记：Coffee文件系统格式化cfs coffee format](#)

[Contiki学习笔记：Coffee文件系统创建文件](#)

[Contiki学习笔记：Coffee文件系统打开文件cfs open](#)

[Contiki学习笔记：Coffee文件系统读取文件cfs read](#)

[Contiki学习笔记：Coffee文件系统写入文件cfs write](#)

[Contiki学习笔记: Coffee文件系统关闭文件cfs\\_close](#)

[Contiki学习笔记: Coffee文件系统删除文件cfs\\_remove](#)

[Contiki学习笔记: Coffee文件系统垃圾回收collect\\_garbage](#)

注:

以上博文系本人通过阅读源码、官方资料、第三方资料的见解，限于知识水平有限，若有错误和不足之处，请务必指出。也欢迎一起讨论学习，E-MAIL: [Jelline@126.com](mailto:Jelline@126.com)

[Contiki学习笔记: 开发环境搭建及学习资料](#) (2011-10-18 11:08)

标签: [Contiki](#) [学习笔记](#) [源码下载](#) [环境搭建](#) [学习资料](#) 分类: [Contiki](#)

**摘要:**

本文讲述了 Contiki 源码下载、开发环境搭建，还给了一些学习资料。

## 一、源码下载

在线浏览源码: <http://contiki.git.sourceforge.net/git/gitweb.cgi?p=contiki/contiki;a=tree>

通过 Git 下载源码:

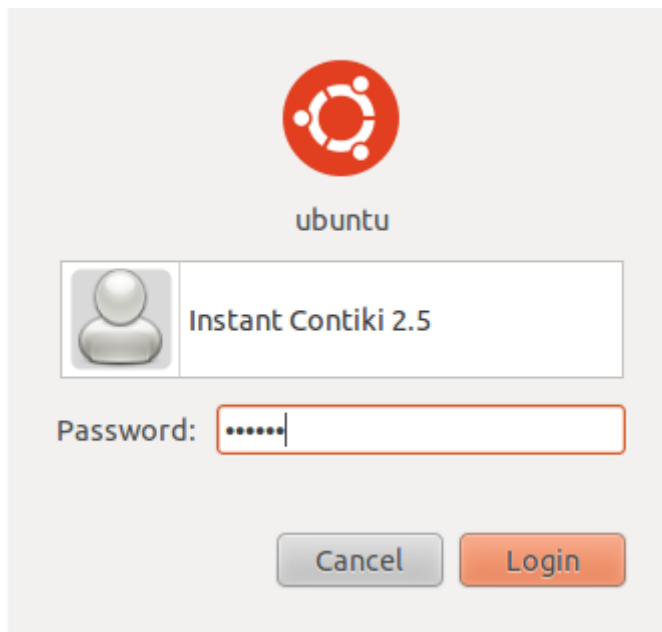
```
1. git clone git://contiki.git.sourceforge.net/gitroot/contiki/contiki
```

## 二、开发环境搭建

### 2.1 Instant Contiki

InstantContiki是官方配好的开发环境，基于Ubuntu，在<http://sourceforge.net/projects/contiki/>下载

InstantContiki，装在虚拟机VMware上，登入界面如下(密码是user):



我只是装好，没用这个作为开发平台。

## 2.2 IAR+J-Linker

Contiki 学习资料较少，搭建开发环境也比较困难。我们是基于 ARM Cortex M3 的板，没有现成的开发环境，自己在 IAR 上搭，对 Makefile 进行大量修改(这是项目组小强的工作)。

## 2.3 其他方式

### 三、学习资料

官方网站: <http://www.sics.se/contiki/> <http://contiki-os.blogspot.com/> (我一直登不了)

Sourceforge: <http://sourceforge.net/projects/contiki/>

Contiki Wiki: [http://www.sics.se/contiki/wiki/index.php/Main\\_Page](http://www.sics.se/contiki/wiki/index.php/Main_Page)

github: <https://github.com/contiki>

The Contiki Operating System: <http://dak664.github.com/contiki-doxygen/>

### 文档

 [Contiki 2.x Reference Manual.pdf](#)

[Contiki Programming Guide.ppt](#)

 [Contiki Programming Course Hands-On Session Notes.pdf](#)



[Contiki学习笔记：事件驱动机制和protothread机制](#) (2011-10-20 10:47)

标签： [Contiki](#) [Protothread](#) [事件驱动](#) [学习笔记](#) [机制](#) 分类： [Contiki](#)

摘要：

本文以通俗的语言阐述了 Contiki 的两个主要机制：事件驱动和 protothread 机制。

**Contiki 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。**

## 一、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，就跑去休眠了(降低功耗)，这就是所谓的事件驱动，类似于中断。

## 二、protothread 机制

### 2.1 概述

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 机制恰解决了这个问题，通过保存进程被阻塞处的行数(进程结构体的一个变量，`unsigned short` 类型，只需两个字节)，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。

### 2.2 特点

protothread(- Lightweight, Stackless Threads in C)最大特点就是轻量级，每个 protothread 不需要自己的堆栈，所有的 protothread 使用同一个堆栈，而保存程序断点用两个字节保存被中断的行数即可。具体特点如下[1]：

**Very small RAM overhead** - only two bytes per protothread and no extra stacks

**Highly portable** - the protothreads library is 100% pure C and no architecture specific assembly code

**Can be used with or without an OS**

**Provides blocking wait without full multi-threading or stack-switching**

**Freely available under a BSD-like open source license**

protothread 机制很早就有了，Contiki OS 只要运用这种机制，protothread 机制还可以用于以下情形[1]：

Memory constrained systems

Event-driven protocol stacks

Small embedded systems

Sensor network nodes

Portable C applications

### 2.3 编程提示

谨慎使用局部变量。当进程切换时，因 `protothread` 没有保存堆栈上下文(只使用两个字节保存被中断的行号)，故局部变量得不到保存。这就要求使用局部变量要特别小心，一个好的解决方法，**使用局部静态变量**(加 `static` 修饰符)，如此，该变量在整个生命周期都存在。

### 2.4 调度[2]

直接看原文吧。A protothread is driven by repeated calls to the function in which the protothread is running. Each time the function is called, the protothread will run until it blocks or exits. Thus the scheduling of protothreads is done by the application that uses protothreads.

如果还想进一步了解 `protothread`，可以到[3]下载源码，仔细研读。

#### 参考资料：

[1]Protothreads - Lightweight, Stackless Threads in C : <http://www.sics.se/~adam/pt/>

[2>About protothreads : <http://www.sics.se/~adam/pt/about.html>

[3]SourceForge--protothread : <http://sourceforge.net/projects/protothread/>

标签: [Contiki](#) [学习笔记](#) [hello\\_world](#) [PROCESS](#) [AUTOSTART\\_PROCESSES](#) 分类: [Contiki](#)

### 摘要:

本文剖析 Contiki 最简单的实例 hello\_world，深入源码分析，详解了本实例用到的各个宏，进而给出一份完整展开的代码。最后把本实例用到的宏总结成 API，并给出了创建一个进程的模型。

## 一、Hello World 概览

hello\_world.c 用于向串口打印“Hello World”，源代码如下，

```
1. //filename: hello_world.c
2.
3. #include "contiki.h"
4. #include "debug-uart.h" /* For usart_puts() */
5. #include <stdio.h> /* For printf() */
6.
7. PROCESS(hello_world_process, "Hello world"); /*参照二*/
8. AUTOSTART_PROCESSES(&hello_world_process); /*参照三*/
9.
10. /*Define the process code*/
11. PROCESS_THREAD(hello_world_process, ev, data) /*参见四*/
12. {
13.     PROCESS_BEGIN(); /*参照 5.1*/
14.
```

```
15.      usart_puts("Hello, world!\n"); /*向串口打印字符串"Hello, world"*/
16.
17.      PROCESS_END(); /*参考 5.2*/
18. }
```

接下来逐句分析。

## 二、PROCESS 宏

PROCESS 宏完成两个功能：

- (1) 声明一个函数，该函数是进程的执行体，即进程的 thread 函数指针所指的函数
- (2) 定义一个进程

源码展开如下：

```
1. //PROCESS(hello_world_process, "Hello world");
2.
3. #define PROCESS(name, strname) PROCESS_THREAD(name, ev, data); \
4. struct process name = { NULL, strname, process_thread_##name }
```

对应参数展开为：

```
1. #define PROCESS((hello_world_process, "Hello world")
2.
3. PROCESS_THREAD(hello_world_process, ev, data); \    /*分析见 2.1*/
```

- 4.
5. `struct process hello_world_process = { NULL, "Hello world", process_thread_hello_world_process }; /*分析见 2.2*/`

## 2.1 PROCESS\_THREAD 宏

PROCESS\_THREAD 宏用于定义进程的执行主体，宏展开如下

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

对应参数展开为：

1. `//PROCESS_THREAD(hello_world_process, ev, data);`
- 2.
3. `static PT_THREAD(process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data)); /*分析见 2.1.1*/`

### 2.1.1 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：

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

展开之后即为：

1. `//static PT_THREAD(process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data));`

- 2.
3. `static char process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data);`

这下就很清楚了，声明一个静态的函数 `process_thread_hello_world_process`，返回值是 `char` 类型。

另，`struct pt *process_pt` 可以直接理解成 `lc`，用于保存当前被中断的地方(保存程序断点)，以便下次恢复执行。

## 2.2 定义一个进程

PROCESS 宏展开的第二句，定义一个进程 `hello_world_process`，源码如下：

1. `struct process hello_world_process = { NULL, "Hello world", process_thread_hello_world_process };`

结构体 `process` 定义如下：

1. `struct process`
2. `{`
3.  `struct process *next;`
4.  `const char *name; /*此处略作简化，源代码包含了预编译#if。即可以通过配置，使得进程名称可有可无*/`
5.  `PT_THREAD((* thread)(struct pt *, process_event_t, process_data_t));`
6.  `struct pt pt;`
7.  `unsigned char state, needspoll;`
8. `};`

可见进程 `hello_world_process` 的 `lc`、`state`、`needspoll` 都默认置为 0。关于 `process` 结构体请参见博文 [《Contiki 学习笔记：主要数据结构之进程》](#)。

### 三、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：

1. `//AUTOSTART_PROCESSES(&hello_world_process);`
- 2.
3. `#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：`debug("Y = %d\n", y);` 被替换成 `printf("Y = %d\n", y);`。那么，`AUTOSTART_PROCESSES(&hello_world_process);` 实际上被替换成：

1. `struct process * const autostart_processes[] = {&hello_world_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，比如让 `hello_process` 和 `world_process` 这两个进程自启动，如下：

1. `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。

### 四、PROCESS\_THREAD 宏

`PROCESS(hello_world_process, "Hello world");` 展开成两句，其中有一句是也是 `PROCESS_THREAD(hello_world_process, ev, data);`。这里要注意到分号，是一个函数声明。而这 `PROCESS_THREAD(hello_world_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，展开过程参见上述 2.1。

```
1. static char process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data);
```

温馨提示：在阅读 Contiki 源码，手动展开宏时，要特别注意分号。

## 五、PROCESS\_BEGIN 宏和 PROCESS\_END 宏

原则上，所有代码都得放在 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，情况就比较复杂了，当然，不建议这样做)，看完下面宏展开，就知道为什么了。

### 5.1 PROCESS\_BEGIN 宏

PROCESS\_BEGIN 宏一步步展开如下：

```
1. #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(见四)，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：

```
1. #define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1; LC_RESUME((pt)->lc)
2.
3. #define LC_RESUME(s) switch(s) { case 0:
```

把参数替换，结果如下：

```
1. {
2.
3.     char PT_YIELD_FLAG = 1; /*将 PT_YIELD_FLAG 置 1，类似于关中断？？？*/
```



```
4.  
5.     switch(process_pt->lc) /*程序根据 lc 的值进行跳转，lc 用于保存程序断点*/  
6.  
7.     {  
8.  
9.         case 0: /*第一次执行从这里开始，可以放一些初始化的东东*/  
10.  
11.             ;
```

很奇怪是吧，PROCESS\_BEGIN 宏展开都不是完整的语句，别急，看完下面的 PROCESS\_END 就知道 Contiki 这些天才们是怎么设计的。

## 5.2 PROCESS\_END 宏

PROCESS\_END 宏一步步展开如下：

```
1. #define PROCESS_END() PT_END(process_pt)  
2.  
3. #define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0; \ PT_INIT(pt); return PT_ENDED; }  
4.  
5.  
6. #define LC_END(s) }  
7.  
8. #define PT_INIT(pt) LC_INIT((pt)->lc)  
9. #define LC_INIT(s) s = 0;  
10.  
11. #define PT_ENDED 3
```

整理下，实际上如下代码：

```
1.     }
2.     PT_YIELD_FLAG = 0;
3.
4.     process_pt = 0;
5.
6.     return 3;
7. }
8.
```

好了，现在回过头来看，PROCESS\_BEGIN 宏和 PROCESS\_END 宏是如此的般配，天生一对呀，祝福他们:-)

## 六、总结

### 6.1 宏全部展开

根据上述的分析，该实例全部展开的代码如下，再浏览下，是不是有种神清气爽的感觉:-)

```
1. #include "contiki.h"
2. #include "debug-uart.h" /* For usart_puts() */
3. #include <stdio.h> /* For printf() */
4.
5. static char process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data);
6.
7. struct process hello_world_process = { ((void *)0), "Hello world process", process_thread_hello_world_process };
8. struct process * const autostart_processes[] = { &hello_world_process, ((void *)0) };
```

```
9.
10. char process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data)
11. {
12.     {
13.         char PT_YIELD_FLAG = 1;
14.         switch((process_pt)->lc)
15.         {
16.             case 0:
17.                 ;
18.
19.                 usart_puts("Hello, world!\n");
20.
21.             };
22.         }
23.         PT_YIELD_FLAG = 0;
24.         (process_pt)->lc = 0;;
25.         return 3;
26. }
```

## 6.2 宏总结

对本实例用到的宏总结如上，以后就直接把宏当 API 用了。

### PROCESS(name, strname)

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

## AUTOSTART\_PROCESSES(...)

定义一个进程指针数组 autostart\_processes

## PROCESS\_THREAD(name, ev, data)

进程 name 的定义或声明，取决于宏后面是";"还是"{}"

## PROCESS\_BEGIN()

进程的主体函数从这里开始

## PROCESS\_END()

进程的主体函数从这里结束

## 6.3 编程模型

这实例虽说很简单，但却给出了定义一个进程的模型(还以 Hello world 为例)，实际编程过程中，只需要将 usart\_puts("Hello, world!\n"); 换成自己需要实现的代码。

1. //假设进程名称为 Hello world
2. #include "contiki.h"
3. #include <stdio.h> /\* For printf() \*/
- 4.
5. PROCESS(hello\_world\_process, "Hello world"); //PROCESS(name, strname)
6. AUTOSTART\_PROCESSES(&hello\_world\_process); //AUTOSTART\_PROCESS(...)

```
7.  
8. /*Define the process code*/  
9. PROCESS_THREAD(hello_world_process, ev, data) //PROCESS_THREAD(name, ev, data)  
10. {  
11.     PROCESS_BEGIN();  
12.  
13.     /**这里填入你的代码**/  
14.  
15.     PROCESS_END();  
16. }
```

博文

[Contiki学习笔记：主要数据结构之进程](#) (2011-10-20 15:43)

标签： [Contiki](#) [process](#) [学习笔记](#) [数据结构](#) [进程](#) 分类： [Contiki](#)

**摘要：**

本文介绍 Contiki 主要数据结构之进程，深入源码分析，并用图示直观表示进程链表。

## 一、进程

进程结构体源码如下：

```
1. struct process
2. {
3.     struct process *next; //指向下一个进程
4.
5.     /*****见 1.1 进程名称*****/
6.     #if PROCESS_CONF_NO_PROCESS_NAMES
7.         #define PROCESS_NAME_STRING(process) ""
8.     #else
9.         const char *name;
10.        #define PROCESS_NAME_STRING(process) (process)->name
11.    #endif
12.
13.    PT_THREAD((*thread)(struct pt *, process_event_t, process_data_t)); //见 1.2
```

```
14.    struct pt pt;                //见 1.3
15.    unsigned char state;         //见 1.4
16.    unsigned char needspoll;    //见 1.5
17.};
```

## 1.1 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。在以后讨论中，均假设配有进程名称。

## 1.2 PT\_THREAD 宏

PT\_THREAD 宏定义如下：

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

故该语句展开如下：

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

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，详情请参考博文《[Contiki 学习笔记：实例hello world剖析](#)》。另，声明一个进程包含在宏 `PROCESS(name, strname)` 里，通过宏 `AUTOSTART_PROCESSES(...)` 将进程加入自启动数组中。

## 1.3 pt

pt 结构体一步步展开如下：

```
1. struct pt
2. {
3.     lc_t lc;
4. };
5.
6. typedef unsigned short lc_t;
```

如此,可以把 struct pt pt 直接理解成 unsigned short lc,以后如无特殊说明,pt 就直接理解成 lc。lc(local continuations)用于保存程序被中断的行数(只需两个字节,这恰是 protothread 轻量级的集中体现),被中断的地方,保存行数 (s=\_\_LINE\_\_)接着是语句 case \_\_LINE\_\_。当该进程再次被调度时,从 PROCESS\_BEGIN()开始执行,而该宏展开含有这条语句 switch(process\_pt->pt),从而跳到上一次被中断的地方(即 case \_\_LINE\_\_),继续执行。

## 1.4 进程状态

进程共 3 个状态,宏定义如下:

```
1. #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*/
2. #define PROCESS_STATE_RUNNING  1      /*进程正在执行*/
3. #define PROCESS_STATE_CALLED   2      /*实际上是返回,并保存 lc 值*/
```

## 1.5 needspoll

简而言之,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,而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。



1. `static volatile unsigned char poll_requested;`

## 二、进程链表

基于上述分析，将代码展开或简化，得到如下进程链表 `process_list`：

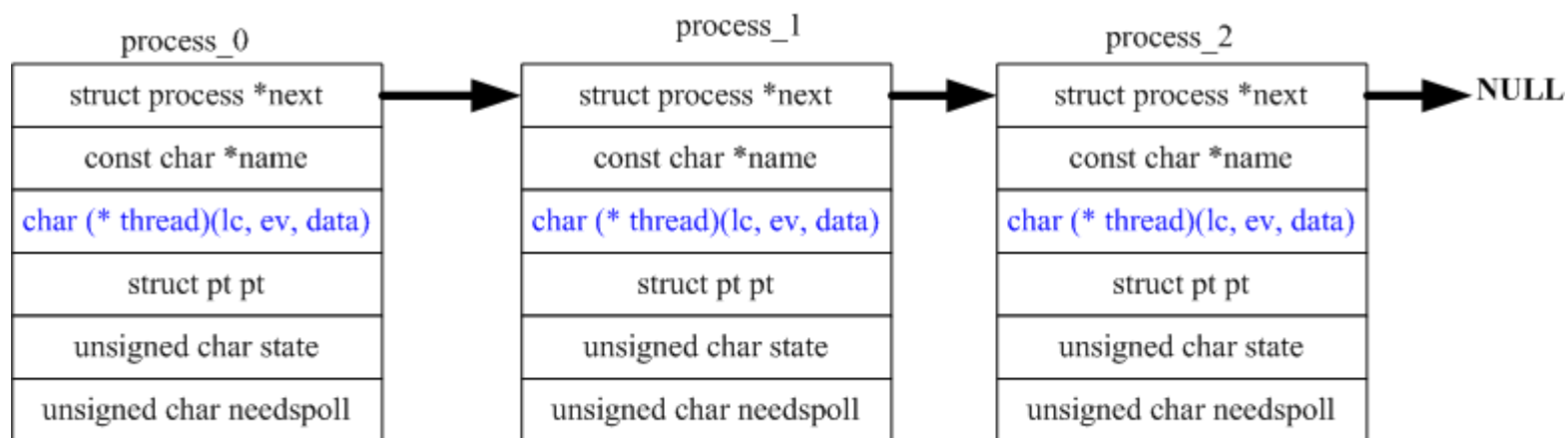


图 Contiki进程链表 (注：为直观显示，将部分宏展开)

[Contiki学习笔记：主要数据结构之事件](#) (2011-10-20 17:18)

标签： [Contiki](#) [事件](#) [事件队列](#) [学习笔记](#) [数据结构](#) 分类： [Contiki](#)

## 摘要：

本文介绍了 Contiki 主要数据结构之事件，深入源码分析事件结构体，图示事件队列，并介绍事件处理和新事件加入是如何影响事件队列的。

## 一、事件

### 1.1 事件结构体

事件也是 Contiki 重要的数据结构，其定义如下：

```
1. struct event_data
2. {
3.     process_event_t ev;
4.     process_data_t data;
5.     struct process *p;
6. };
7.
8. typedef unsigned char process_event_t;
9. 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

## 1.2 事件分类

事件可以被分为三类：时钟事件(timer events)、外部事件、内部事件。那么，Contiki 核心数据结构就只有进程和事件了，把 `etimer` 理解成一种特殊的事件。

## 二、 事件队列

Contiki 用环形队列组织所有事件(用数组存储)，如下：

1. `static struct event_data events[PROCESS_CONF_NUMEVENTS];`

图示事件队列如下：

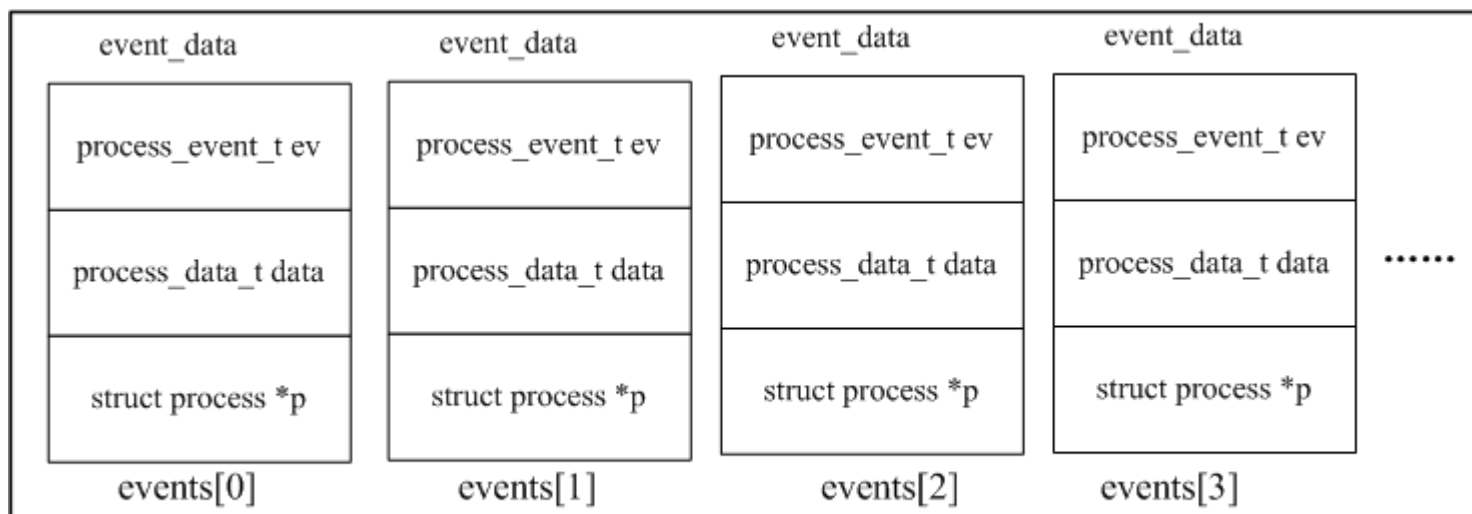


图 events图示

上图源文件，点这里下载  [Contiki事件队列图示.rar](#)

### 三、系统定义的事件

#### 3.1 系统事件

系统定义了 10 个事件，源码和注释如下：

1. `/*配置系统最大事件数*/`
2. `#ifndef PROCESS_CONF_NUMEVENTS`

```

3.     #define PROCESS_CONF_NUMEVENTS 32
4. #endif
5.
6. #define PROCESS_EVENT_NONE           0x80 //函数 dhcpc_request 调用 handle_dhcp(PROCESS_EVENT_NONE, NULL)
7. #define PROCESS_EVENT_INIT           0x81 //启动一个进程 process_start, 通过传递该事件
8. #define PROCESS_EVENT_POLL           0x82 //在 PROCESS_THREAD(etimer_process, ev, data)使用到
9. #define PROCESS_EVENT_EXIT           0x83 //进程退出, 传递该事件给进程主体函数 thread
10. #define PROCESS_EVENT_SERVICE_REMOVED 0x84
11. #define PROCESS_EVENT_CONTINUE       0x85 //PROCESS_PAUSE 宏用到这个事件
12. #define PROCESS_EVENT_MSG            0x86
13. #define PROCESS_EVENT_EXITED         0x87 //进程退出, 传递该事件给其他进程
14. #define PROCESS_EVENT_TIMER          0x88 //etimer 到期时, 传递该事件
15. #define PROCESS_EVENT_COM            0x89
16. #define PROCESS_EVENT_MAX            0x8a /*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*/

```

注: PROCESS\_EVENT\_EXIT 与 PROCESS\_EVENT\_EXITED 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)`。(助记: EXITED 是完成式, 发给进程, 让整个进程结束。而 g 一般式 EXIT, 发给进程主体 thread, 只是使其退出 thread)

### 3.2 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**(为么不用一个特殊事件来标识广播事件，而采用这种费解方式?)。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。详情见详情见博文《[Contiki学习笔记：深入理解process\\_run函数](#)》2.2，部分源码如下：

```
1. #define PROCESS_BROADCAST NULL //广播进程
2.
3. /*保存待处理事件的成员变量*/
4. ev = events[fevent].ev;
5. data = events[fevent].data;
6. receiver = events[fevent].p;
7.
8. if (receiver == PROCESS_BROADCAST)
9. {
10.     for (p = process_list; p != NULL; p = p->next)
11.     {
12.         if (poll_requested)
13.         {
14.             do_poll();
15.         }
16.         call_process(p, ev, data);
17.     }
18. }
```

[Contiki学习笔记：主要数据结构之etimer](#) (2011-10-21 14:57)

标签: [Contiki](#) [etimer](#) [event timer](#) [学习笔记](#) [数据结构](#) 分类: [Contiki](#)

## 摘要:

本文简单介绍了 Contiki 系统 5 种定时器用途, 进而着重介绍 etimer, 并图示 timerlist。

## 一、定时器概述

Contiki 包含一个时钟模型和 5 个定时器模型(timer, stimer, ctimer, etimer, and rtimer)[1], 5 种 timer 简述如下:

**timer**--be used to check if a time period has passed[1]

**stimer**--be used to check if a time period has passed[1]

**ctimer**--Active timer, calls a functionwhen it expires, Used by Rime[2]

**etimer**--Active timer, sends an event whenit expires[2]

**rtimer**--Real-time timer, calls a functionat an exact time[2]

## 注:

(1) timer 与 stimer 区别在于 the resolution of time: timers use system clock ticks while stimers use seconds to allow much longer time periods[1]。

(2) Unlike the other timers, the timer and stimer libraries can be safely used from interrupts which makes them especially useful in low level drivers.

## 二、etimer

### 2.1 etimer 结构体

etimer 提供一种 timer 机制产生 timed events, 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。etimer 结构体源码如下:

```
1. struct etimer
2. {
3.     struct timer timer;
4.     struct etimer *next;
5.     struct process *p;
6. };
7.
8. /*****timer 定义*****/
9. struct timer
10. {
11.     clock_time_t start;
12.     clock_time_t interval;
13. };
14.
15. typedef unsigned int clock_time_t;
```



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

## 2.2 timerlist

全局静态变量 timerlist，指向系统第一个 etimer，图示 timerlist 如下：

```
1. static struct etimer *timerlist;
```

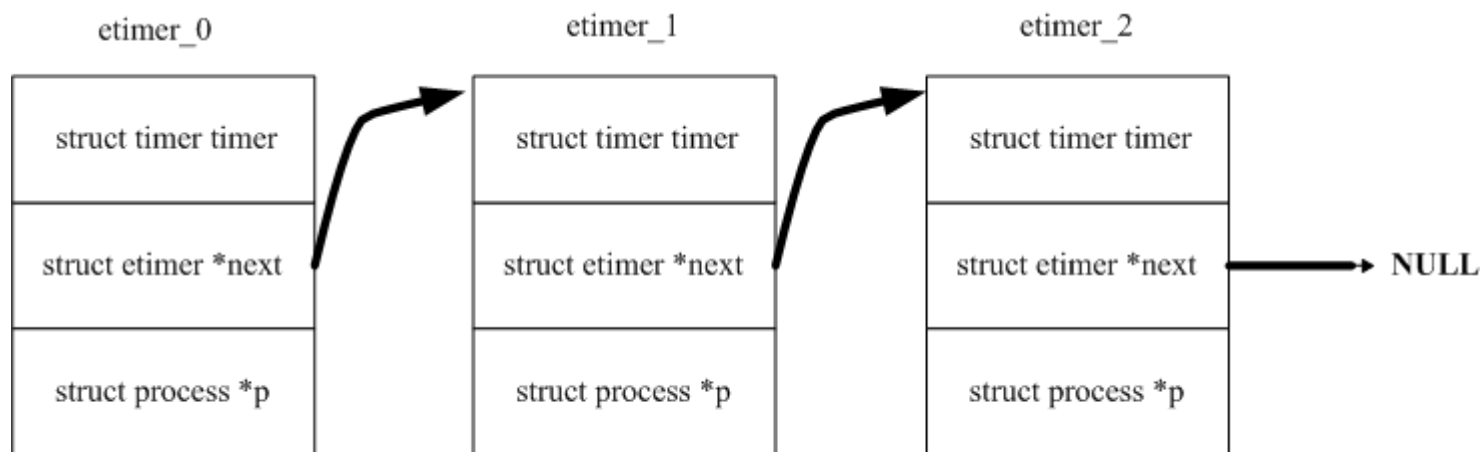


图 Contiki etimer链表timerlist

etimer相关的API：参考《[Contiki 2.5: core/sys/etimer.h File Reference](#)》

### 参考资料:

- [1] Timers - ContikiWiki : <http://www.sics.se/contiki/wiki/index.php/Timers>
- [2] 博文《[contiki代码学习之二：浅探Event-Driven模型【1】](#)》

[Contiki学习笔记：进程、事件、etimer关系](#) (2011-10-21 15:31)

标签: [Contiki](#) [学习笔记](#) [etimer](#) [事件](#) [进程](#) 分类: [Contiki](#)

### 摘要:

本文以通俗的语言介绍进程、事件、etimer 三者关系。

进程 process、事件 event\_data、etimer 都是 Contiki 的核心数据结构，理清这三者关系，将有助于对系统的理解。

## 一、事件与 etimer 关系

事件即可以分为同步事件、异步事件，也可以分为定时器事件、内部事件、外部事件。而 etimer 属于定时器事件的一种，可以理解成 Contiki 系统把 etimer 单列出来，方便管理(由 etimer\_process 系统进程管理)。

当etimer\_process执行时，会遍历etimer链表，检查etimer是否有到期的，凡有timer到期就把事件PROCESS\_EVENT\_TIMER加入到事件队列中，并将该etimer成员变量p指向PROCESS\_NONE。在这里，PROCESS\_NONE用于标识该etimer是否到期，函数etimer\_expired会根据etimer的p是否指向PROCESS\_NONE来判断该etimer是否到期。详情参考博文《[Contiki学习笔记：系统进程etimer\\_process](#)》。

## 二、进程与 etimer 关系

etimer 与 process 还不是一一对应的关系，一个 etimer 必定绑定一个 process，但 process 不一定非得绑定 etimer。etimer 只是一种特殊事件罢了。

## 三、进程与事件关系

当有事件传递给进程时，就新建一个事件加入事件队列，并绑定该进程，所以一个进程可以对应于多个事件(即事件队列有多个事件跟同一个进程绑定)，而一个事件可以广播给所有进程，即该事件成员变量 p 指向空。当调用 do\_event 函数时，将进程链表所有进程投入运行。

[Contiki学习笔记：创建两个交互进程](#) (2011-10-18 17:02)

标签： [Contiki](#) [交互进程](#) [学习笔记](#) 分类： [Contiki](#)

### 摘要：

本文给出两个交互进程的实例，先给出程序源代码，运行截图，接着解读该程序，最后深入源码详细分析。

创建两个进程，一个打印 Hello，另一个打印 World，使其交互打印。

## 一、源代码

声明进程 `print_hello_process` & `print_world_process`，所涉及宏参见博文《[Contiki学习笔记：实例hello\\_world剖析](#)》：

```
1. //filename:interact_two_process.c
2. #include "contiki.h"
3. #include "debug-uart.h"
4.
5. static process_event_t event_data_ready;
6.
7. /*声明两个进程*/
8. PROCESS(print_hello_process, "Hello");
9. PROCESS(print_world_process, "world");
10.
11. AUTOSTART_PROCESSES(&print_hello_process, &print_world_process); //让该两进程自启动
```

定义进程 `print_hello_process`:

```
1. /*定义进程 print_hello_process*/
2. PROCESS_THREAD(print_hello_process, ev, data)
3. {
4.     PROCESS_BEGIN();
5.     static struct etimer timer;
6.
```

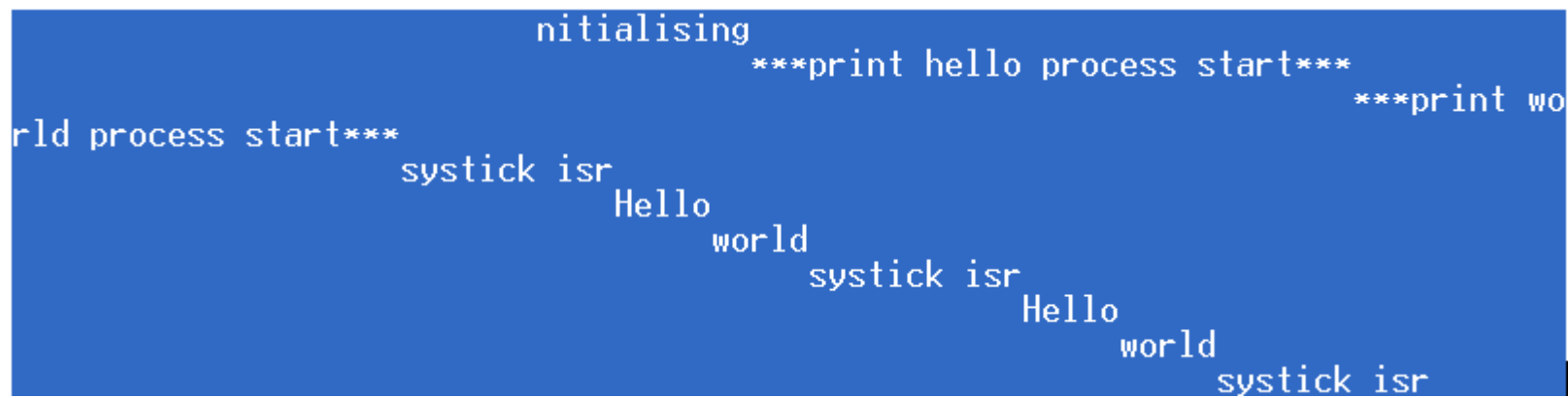
```
7.   etimer_set(&timer, CLOCK_CONF_SECOND / 10); //define CLOCK_CONF_SECOND 10 将 timer 的 interval 设为 1 详情见
    4.1
8.
9.   usart_puts("***print hello process start***\n");
10.
11.  event_data_ready = process_alloc_event(); //return lastevent++; 新建一个事件，事实上是用一组 unsigned char
    值来标识不同事件
12.
13.  while (1)
14.  {
15.      PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER); //详情见 4.2，即 etimer 到期继续执行下面内容
16.
17.      usart_puts("Hello\n");
18.
19.      process_post(&print_world_process, event_data_ready, NULL); //传递异步事件给 print_world_process，直到
    内核调度该进程才处理该事件。详情见 4.3
20.
21.      etimer_reset(&timer); //重置定时器，详情见 4.4
22.  }
23.
24.  PROCESS_END();
25. }
```

定义进程 print\_world\_process:

```
1. /*定义进程 print_world_process*/
```

```
2. PROCESS_THREAD(print_world_process, ev, data)
3. {
4.     PROCESS_BEGIN();
5.
6.     usart_puts("***print world process start***\n");
7.
8.     while (1)
9.     {
10.        PROCESS_WAIT_EVENT_UNTIL(ev == event_data_ready);
11.        usart_puts("world\n");
12.    }
13.
14.    PROCESS_END();
15. }
```

## 二、运行结果截图



```
nitialising
***print hello process start***
***print world process start***
systick isr
Hello
world
systick isr
Hello
world
systick isr
```

### 三、程序解读

系统启动, 执行一系列初始化(串口、时钟、进程等), 接着启动系统进程 `etimer_process`, 而后启动进程 `print_hello_process` 和 `print_world_process`。那么 `print_hello_process` 与 `print_world_process` 是怎么交互的呢?

进程 `print_hello_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_world_process`, 待执行到 `PROCESS_WAIT_EVENT_UNTIL(ev == event_data_ready)` 被挂起(因为 `print_hello_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 `print_hello_process`, 打印 Hello, 并传递事件 `event_data_ready` 给 `print_world_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_world_process`, 打印 world, 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。

### 四、源码详解

#### 4.1 etimer\_set 函数

```
1. //etimer_set(&timer, CLOCK_CONF_SECOND/10);
2. void etimer_set(struct etimer *et, clock_time_t interval)
3. {
4.     timer_set(&et->timer, interval); //设置定时器 timer, 源码见 2.1.1
5.     add_timer(et); //将这个 etimer 加入 timerlist
6. }
```

##### 4.1.1 timer\_set 函数

```
1. //用当前时间初始化 start, 并设置间隔 interval
2. void timer_set(struct timer *t, clock_time_t interval)
```

```
3. {
4.     t->interval = interval;
5.     t->start = clock_time(); //return current_clock;
6. }
7.
8. clock_time_t clock_time(void)
9. {
10.     return current_clock;
11. }
12.
13. static volatile clock_time_t current_clock = 0;
14. typedef unsigned int clock_time_t;
```

#### 4.1.2 add\_timer 函数

```
1. /*将 etimer 加入 timerlist, 并 update_time(), 即求出下一个到期时间 next_expiration*/
2. static void add_timer(struct etimer *timer)
3. {
4.     struct etimer *t;
5.
6.     etimer_request_poll(); //事实上执行 process_poll(&etimer_process), 即将进程的 needspoll 设为 1, 使其获得更高
    优先级, 详情见下方
7.
8.     /*参数验证, 确保该 etimer 不是已处理过的*/
9.     if (timer->p != PROCESS_NONE) //如果是 PROCESS_NONE, 则表示之前处理过, 该 etimer 已到期(注 1)
10.     {
```



```
11.     for (t = timerlist; t != NULL; t = t->next)
12.     {
13.         if (t == timer) /* Timer already on list, bail out. */
14.         {
15.             update_time(); //即求出下一个到期时间next_expiration(全局静态变量),即还有next_expiration时间,
就有 timer 到期
16.             return ;
17.         }
18.     }
19. }
20.
21. /*将 timer 加入 timerlist*/
22. timer->p = PROCESS_CURRENT();
23. timer->next = timerlist;
24. timerlist = timer;
25.
26. update_time(); //即求出下一个到期时间 next_expiration(全局静态变量), 即还有 next_expiration 时间, 就有 timer
到期
27. }
```

注 1: 关于PROCESS\_NONE可以参考博文《[Contiki学习笔记: 系统进程etimer\\_process](#)》。

函数 etimer\_request\_poll 直接调用 process\_poll, 一步步展开如下:

```
1. void etimer_request_poll(void)
2. {
```

```
3.     process_poll(&etimer_process);
4. }
5.
6. void process_poll(struct process *p)
7. {
8.     if (p != NULL)
9.     {
10.         if (p->state == PROCESS_STATE_RUNNING || p->state == PROCESS_STATE_CALLED)
11.         {
12.             p->needspoll = 1;
13.             poll_requested = 1; //全局静态变量, 标识是否有进程的 needspoll 为 1
14.         }
15.     }
16. }
```

#### 4.2 PROCESS\_WAIT\_EVENT\_UNTIL 宏

```
1. //PROCESS_WAIT_EVENT_UNTIL(ev == PROCESS_EVENT_TIMER);
2.
3. /*宏 PROCESS_WAIT_EVENT_UNTIL, 直到条件 c 为真时, 进程才得以继续向前推进*/
4. #define PROCESS_WAIT_EVENT_UNTIL(c) PROCESS_YIELD_UNTIL(c)
5.
6. #define PROCESS_YIELD() PT_YIELD(process_pt)
7.
8. /* Yield from the current protothread. */
9. #define PT_YIELD(pt) \
```

```
10. do { \
11. PT_YIELD_FLAG = 0; \
12. LC_SET((pt)->lc); \
13. if (PT_YIELD_FLAG == 0) { \
14. return PT_YIELDED; \
15. } \
16. } while(0)
17.
18. #define LC_SET(s) s = __LINE__; case __LINE__: //保存行数，即 lc 被设置成该 LC_SET 所在的行
```

### 4.3 process\_post 函数

精简后(去除一些用于调试的代码)源码如下，详情参考博文《[Contiki学习笔记：系统进程etimer\\_process](#)》2.3:

```
1. //process_post(&print_world_process, event_data_ready, NULL); 即把事件 event_data_ready 加入事件队列
2. int process_post(struct process *p, process_event_t ev, process_data_t data)
3. {
4.     static process_num_events_t snum;
5.
6.     if (nevents == PROCESS_CONF_NUMEVENTS)
7.     {
8.         return PROCESS_ERR_FULL;
9.     }
10.
11.     snum = (process_num_events_t) (fevent + nevents) % PROCESS_CONF_NUMEVENTS;
12.     events[snum].ev = ev;
```

```
13.  events[snum].data = data;
14.  events[snum].p = p;
15.  ++nevents;
16.
17.  #if PROCESS_CONF_STATS
18.      if (nevents > process_maxevents)
19.      {
20.          process_maxevents = nevents;
21.      }
22.  #endif
23.
24.  return PROCESS_ERR_OK;
25. }
```

#### 4.4 etimer\_reset 函数

```
1.  /*Reset the timer with the same interval.*/
2.  void etimer_reset(struct etimer *et)
3.  {
4.      timer_reset(&et->timer);
5.      add_timer(et); //详情见 4.1.2
6.  }
7.
8.  void timer_reset(struct timer *t)
9.  {
10.     t->start += t->interval;    //为什么不 t->start = clock_time()? 为了程序可预测性?
```

11. }

[Contiki学习笔记: main函数剖析](#) (2011-10-26 20:04)

标签: [Contiki](#) [etimer\\_process](#) [main](#) [process\\_start](#) [学习笔记](#) 分类: [Contiki](#)

### 摘要:

本文旨在剖析 main 函数，从更高层次理解 Contiki 系统。先是给出源码，接着总结功能，最后深入源码分析。

### 一、main 函数源码

```
1. //filename:contiki-main.c
2. #include <stm32f10x.h>
3. #include <stm32f10x_dma.h>
4. #include <stdint.h>
5. #include <stdio.h>
6. #include <debug-uart.h>
7. #include <sys/process.h>
8. #include <sys/procinit.h>
9. #include <etimer.h>
10. #include <sys/autostart.h>
```

```
11. #include <clock.h>
12.
13. int main()
14. {
15.     dbg_setup_uart(); //串口初始化, 与硬件相关(注 1)
16.     usart_puts("Initialising\n"); //向串口打印字符串"Initialising"
17.
18.     clock_init(); //时钟初始化, 与硬件相关, 见 3.3
19.
20.     process_init(); //进程初始化, 详情见 3.1
21.     process_start(&etimer_process, NULL); //启动 etimer_process, 详情见 3.2
22.     autostart_start(autostart_processes); //启动指针数组 autostart_processes[] 里的所有进程, 详情见 3.2
23.
24.     while (1)
25.     {
26.         /*执行完所有 needspoll 为 1 的进程及处理完所有队列, 详情见 3.2*/
27.         do
28.         {
29.         }
30.         while (process_run() > 0);
31.     }
32.     return 0;
33. }
```

注 1:

串口初始化和时钟初始化与硬件相关(我用的 MCU 是 STM32F103RBT6)，不深入讨论，主要基于以下两个原因：其一，这部分对理解系统影响不大；其二，我对硬件不是很熟。

## 二、程序解读

系统先进行一系列初始化(串口、时钟、进程)，接着启动系统进程 `etimer_process` 和指针数组 `autostart_processes[]` 里的所有进程，到这里就启动了所有的系统进程(当然，后续的操作还可能动态产生新进程)。接下来的工作，就是系统反复处理所有 `needspoll` 标记为 1 的进程及事件队列中所有事件。而处理事件过程中(典型情况是让某些进程继续执行)又可能产生新的事件，再处理事件，系统如此反复运行。

## 三、源码详解

### 3.1 进程初始化

```
1.  /****进程初始化****/
2.  void process_init(void)
3.  {
4.      /*初始化事件队列*/
5.      lastevent = PROCESS_EVENT_MAX;
6.      nevents = fevent = 0;
7.
8.      #if PROCESS_CONF_STATS
9.          process_maxevents = 0;
10.     #endif
11.
12.     process_current = process_list = NULL; //初始化进程链表
13. }
```

### 3.2 注明

有关etimer\_process参考博文《[Contiki学习笔记：系统进程etimer\\_process](#)》

有关process\_start参考博文《[Contiki学习笔记：启动一个进程process\\_start](#)》

有关process\_start参考博文《[Contiki学习笔记：深入理解process\\_run函数](#)》

### 3.3 时钟初始化

clock\_init 用于配置系统产生嘀嗒的间隔，即每隔 72000000/CLOCK\_SECOND 产生一次系统嘀嗒(72000000 是指 1s 可以产生最大的嘀嗒数)，也就是说 1 秒钟可以产生 CLOCK\_SECOND 次时钟中断。

```
1. void clock_init()
2. {
3.     SysTick_Config(72000000 / CLOCK_SECOND);
4. }
5.
6. /*CLOCK_SECOND 宏定义*/
7. #define CLOCK_CONF_SECOND 10
8.
9. #ifdef CLOCK_CONF_SECOND
10.     #define CLOCK_SECOND CLOCK_CONF_SECOND
11. #else
12.     #define CLOCK_SECOND (clock_time_t)32
13. #endif
```



SysTick\_Config 函数展开如下，与硬件相关(我用的 MCU 是 STM32F103RBT6)：

```
1. /**
2.  * @brief Initialize and start the SysTick counter and its interrupt.
3.  *
4.  * @param ticks number of ticks between two interrupts
5.  * @return 1 = failed, 0 = successful
6.  *
7.  * Initialise the system tick timer and its interrupt and start the
8.  * system tick timer / counter in free running mode to generate
9.  * periodical interrupts.
10. */
11. static __INLINE uint32_t SysTick_Config(uint32_t ticks)
12. {
13.     if (ticks > SysTick_LOAD_RELOAD_Msk) return (1);           /* Reload value impossible */
14.
15.     SysTick->LOAD = (ticks & SysTick_LOAD_RELOAD_Msk) - 1;     /* set reload register */
16.     NVIC_SetPriority (SysTick_IRQn, (1<<__NVIC_PRIO_BITS) - 1); /* set Priority for Cortex-M0 System Interrupts
17. */
18.     SysTick->VAL = 0;      /* Load the SysTick Counter Value */
19.     SysTick->CTRL = SysTick_CTRL_CLKSOURCE_Msk |
20.                   SysTick_CTRL_TICKINT_Msk |
21.                   SysTick_CTRL_ENABLE_Msk;    /* Enable SysTick IRQ and SysTick Timer */
22.     return (0);          /* Function successful */
23. }
```

[Contiki学习笔记：启动一个进程process\\_start](#) (2011-09-22 14:53)

标签： [Contiki](#) [学习笔记](#) [启动进程](#) [process\\_start](#) [call\\_process](#) 分类： [Contiki](#)

## 摘要：

本文深入源码，详细分析 Contiki 启动一个进程的过程。先是总结了 process\_start() 都做了些什么事，进而跟踪代码进行详细分析。

## 引言

process\_start 函数用于启动一个进程，将进程加入进程链表(事先验证参数，确保进程不在进程链表中)，初始化进程(将进程状态设为运行状态及 lc 设为 0)。给进程传递一个 PROCESS\_EVENT\_INIT 事件，让其开始执行(事先参数验证，确保进程已被设为运行态

并且进程的函数指针 `thread` 不为空), 事实上是执行进程结构体中的 `thread` 函数指针所指的函数, 而这恰恰是 `PROCESS_THREAD(name, ev, data)` 函数的实现。判断执行结果, 如果返回值表示退出、结尾或者遇到 `PROCESS_EVENT_EXIT`, 进程退出, 否则进程被挂起, 等待事件。

理解系统最好的方法就是阅读源码, 分享 Linus 的一句话, 接下来跟踪代码详细分析 Contiki 系统是如何启动一个进程的。

1. Read the Fucking Source Code ——Linus Torvalds

整个调用过程如下:

1. `process_start`——>`process_post_synch`——>`call_process`——>`exit_process`

## 一、`process_start` 函数

将进程加入进程链表(事先验证参数, 确保进程不在进程链表中), 初始化进程(将进程状态设为运行状态及 lc 设为 0)。给进程传递一个 PROCESS\_EVENT\_INIT 事件, 让其开始执行。

```
1.  /******启动一个进程******/
2.  void process_start(struct process *p, const char *arg) //可以传递 arg 给进程 p, 也可以不传, 直接 "NULL"
3.  {
4.      struct process *q;
5.      /*参数验证: 确保进程不在进程链表中*/
6.      for(q = process_list; q != p && q != NULL; q = q->next);
7.      if(q == p)
8.      {
9.          return;
10.     }
11.     /*把进程加到进程链表首部*/
12.     p->next = process_list;
13.     process_list = p;
14.
15.     p->state = PROCESS_STATE_RUNNING;
16.     PT_INIT(&p->pt); //将 p->pt->lc 设为 0, 使得进程从 case 0 开始执行
17.
18.     PRINTF("process: starting '%s' \n", PROCESS_NAME_STRING(p));
19.
```

```
20.      //给进程传递一个 PROCESS_EVENT_INIT 事件，让其开始执行
21.      process_post_synch(p, PROCESS_EVENT_INIT, (process_data_t)arg);
22. }
```

## 二、process\_post\_synch 函数

process\_post\_synch() 直接调用 call\_process()，期间需要保存 process\_current，这是因为当调用 call\_process 执行这个进程 p 时，process\_current 就会指向当前进程 p，而进程 p 可能会退出或者被挂起等待一个事件[1]。

```
1. //process_post_synch(p, PROCESS_EVENT_INIT, (process_data_t)arg);
2. void process_post_synch(struct process *p, process_event_t ev, process_data_t data) //typedef void * process_data_t;
3. {
4.     struct process *caller = process_current; //相当于 PUSH，保存现场 process_current
5.
6.     call_process(p, ev, data);
7.
8.     process_current = caller; //相当于 POP，恢复现场 process_current
9. }
```

## 三、call\_process 函数

如果进程 process 的状态为 PROCESS\_STATE\_RUNNING，并且进程中的 thread 函数指针（相当于该进程的主函数）不为空的话，就执行该进程。如果返回值表示退出、结尾或者遇到 PROCESS\_EVENT\_EXIT，进程退出，否则进程被挂起，等待事件。

```
1. //call_process(p, PROCESS_EVENT_INIT, (process_data_t)arg);
2. static void call_process(struct process *p, process_event_t ev, process_data_t data)
3. {
4.     int ret;
5.
6. #if DEBUG
7.     if(p->state == PROCESS_STATE_CALLED)
8.     {
9.         printf("process: process '%s' called again with event %d\n", PROCESS_NAME_STRING(p), ev);
10.    }
11. #endif /* DEBUG */
12.
13.    if((p->state & PROCESS_STATE_RUNNING) && p->thread != NULL) //thread 是函数指针
14.    {
15.        PRINTF("process: calling process '%s' with event %d\n", PROCESS_NAME_STRING(p), ev);
16.        process_current = p;
17.        p->state = PROCESS_STATE_CALLED;
18.        ret = p->thread(&p->pt, ev, data); //才真正执行 PROCESS_THREAD(name, ev, data) 定义的内容
19.        if(ret == PT_EXITED || ret == PT_ENDED || ev == PROCESS_EVENT_EXIT)
```

```
20.         {
21.             exit_process(p, p); //如果返回值表示退出、结尾或者遇到 PROCESS_EVENT_EXIT, 进程退出
22.         } else
23.         {
24.             p->state = PROCESS_STATE_RUNNING; //进程挂起等待事件
25.         }
26.     }
27. }
```

#### PT 四种状态

1. #define PT\_WAITING 0 /\*被阻塞, 等待事件发生\*/
2. #define PT\_YIELDED 1 /\*主动让出\*/
3. #define PT\_EXITED 2 /\*退出\*/
4. #define PT\_ENDED 3 /\*结束\*/

#### 进程状态

1. #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\*/
2. #define PROCESS\_STATE\_RUNNING 1 /\*进程正在执行\*/
3. #define PROCESS\_STATE\_CALLED 2 /\*被阻塞, 运行? \*/

#### 四、exit\_process 函数

先进行参数验证，确保进程在进程链表中并且不是 PROCESS\_STATE\_NONE 状态，向所有进程发一个同步事件 PROCESS\_EVENT\_EXITED，通知他们将要退出，让与该进程相关的进程进行相应处理。

用[1]中的一个例子：如果一个程序要退出的话，就会给 etimer\_process 进程发送一个 PROCESS\_EVENT\_EXITED 事件，那么收到这个事件之后，etimer\_process 就会查找 timerlist 看看哪个 timer 是与这个进程相关的，就把它从 timerlist 中清除。

1. `//struct process *p` 指要退出的进程
2. `//struct process *fromprocess` 指当前的进程 ?
3. `//exit_process(p, p)`
4. `static void exit_process(struct process *p, struct process *fromprocess)`
5. `{`
6.  `register struct process *q;`
7.  `struct process *old_current = process_current;`



```
8.      PRINTF("process: exit_process '%s' \n", PROCESS_NAME_STRING(p));
9.
10.     /*参数验证：确保要退出的进程在进程链表中*/
11.     for(q = process_list; q != p && q != NULL; q = q->next);
12.     if(q == NULL)
13.     {
14.         return;
15.     }
16.     if(process_is_running(p)) //return p->state != PROCESS_STATE_NONE;
17.     {
18.         p->state = PROCESS_STATE_NONE;
19.
```

```
20.          /*向所有进程发一个同步事件 PROCESS_EVENT_EXITED, 通知他们将要退出, 使得与该进程相  
            关的进程进行相应处理*/  
21.          for(q = process_list; q != NULL; q = q->next)  
22.          {  
23.              if(p != q)  
24.              {  
25.                  call_process(q, PROCESS_EVENT_EXITED, (process_data_t)p);  
26.              }  
27.          }  
28.  
29.          if(p->thread != NULL && p != fromprocess) /*退出的进程不是当前进程*/  
30.          {  
31.              /* Post the exit event to the process that is about to exit. */
```

```
32.         process_current = p;
33.         p->thread(&p->pt, PROCESS_EVENT_EXIT, NULL); //给进程 p 传递一个
               PROCESS_EVENT_EXIT 事件，通常用于退出死循环(如 PROCESS_WAIT_EVENT 函数)，从而使其从主体函数退出
34.     }
35. }
36.
37. /*将进程 p 从进程链表删除*/
38. if(p == process_list) //进程 p 恰好在进程链表首部
39. {
40.     process_list = process_list->next;
41. }
42. else //进程 p 不在进程链表首部
43. {
```

```
44.         for(q = process_list; q != NULL; q = q->next) //遍历进程链表，寻找进程 p，删除之
45.         {
46.             if(q->next == p)
47.             {
48.                 q->next = p->next;
49.                 break;
50.             }
51.         }
52.     }
53.     process_current = old_current;
54. }
```

PS:本文系阅读源码、论坛资料、官方资料的个人见解，如有错误的地方，烦请您指出，也欢迎交流 [Jelline@126.com](mailto:Jelline@126.com)。

参考资料:

[1] 贴子《[contiki 代码学习之一：浅探 protothread 进程控制模型【2】](#)》

[Contiki学习笔记：系统进程etimer\\_process](#) (2011-10-23 16:17)

标签: [Contiki](#) [etimer\\_process](#) [etimer进程](#) [学习笔记](#) 分类: [Contiki](#)

**摘要:**

系统进程 `etimer_process` 管理 `timelist`, 本文先给出 `etimer_process` 的启动, 而后深入源码分析 `etimer_process` 的执行主体, 追踪了其所涉及到的宏和函数。

## 一、启动 `etimer_process`

在 `main` 函数中, 先是进行一系列初始化, 接着启动系统进程 `etimer_process` 和指针数组 `autostart_processes[]` 里的所有进程。源代码如下:

```
1. int main()
```

```
2. {
3.     dbg_setup_uart();
4.     clock_init();
5.     process_init();
6.
7.     process_start(&etimer_process, NULL); //启动 etimer_process
8.     autostart_start(autostart_processes); //启动指针数组 autostart_processes[]里的所有进程
9.
10.    while (1)
11.    {
12.        do
13.        {}while (process_run() > 0);
14.    }
15.    return 0;
16.}
```

关于process\_start启动一个进程，可以参考博文《[Contiki学习笔记：启动一个进程process\\_start](#)》。

## 二、etimer\_process 执行主体剖析

在博文《Contiki 学习笔记：启动一个进程 process\_start》中，我们得知，进程退出时，需向所有进程(当然也包括 etimer\_process)发送事件 PROCESS\_EVENT\_EXITED。etimer\_process 收到该事件，就会遍历 timerlist，并把与该退出进程相关的 etimer 从 timerlist 删除。紧接着，遍历 timerlist，检查 etimer 是否有到期的，凡有 timer 到期就把事件 PROCESS\_EVENT\_TIMER 加入到事件队列中，并将该 etimer 成员变量 p 指向 PROCESS\_NONE。在这里，PROCESS\_NONE 用于标识该 etimer 是否到期，即由 etimer\_expired 函数根据 etimer 的 p 是否指向 PROCESS\_NONE 来判断该 etimer 是否到期。源代码如下：

```
1. //PROCESS(etimer_process, "Event timer");
2. PROCESS_THREAD(etimer_process, ev, data)
3. {
4.     struct etimer *t, *u;
5.
6.     PROCESS_BEGIN();
7.
8.     timerlist = NULL;
9.
10.    while (1)
11.    {
12.        PROCESS_YIELD(); /*见 2.1*/
13.
14.        /*进程退出时, 需向所有进程(当然也包括 etimer_process)发送事件 PROCESS_EVENT_EXITED。etimer_process 收到
        该事件, 就会遍历 timerlist, 并把与该退出进程相关的 etimer 从 timerlist 删除*/
15.        if (ev == PROCESS_EVENT_EXITED)
16.        {
17.            struct process *p = data;    //此时通过 data 传递将要退出的进程, data 是 void *类型
18.
19.            /*遍历 timerlist, 查找是否有 etimer 绑定该退出进程*/
20.            while (timerlist != NULL && timerlist->p == p)
21.            {
22.                timerlist = timerlist->next;
23.            }
24.
25.            /*有 etimer 绑定该退出进程, 将 etimer 从 timerlist 删除*/
```

```
26.         if (timerlist != NULL)
27.         {
28.             t = timerlist;
29.             while (t->next != NULL)
30.             {
31.                 if (t->next->p == p)
32.                 {
33.                     t->next = t->next->next;
34.                 }
35.                 else
36.                     t = t->next;
37.             }
38.         }
39.
40.         continue; //删除所有与退出进程绑定的 etimer
41.     }
42.     else if (ev != PROCESS_EVENT_POLL)    //不解?
43.     {
44.         continue;
45.     }
46.
47.     again:
48.
49.     u = NULL;
50.     for (t = timerlist; t != NULL; t = t->next)
51.     {
```



```
52.         if (timer_expired(&t->timer)) //检查该 etimer 的 timer 是不是到期, 返回 1 表示过期, 详情见 2.2
53.         {
54.             if (process_post(t->p, PROCESS_EVENT_TIMER, t) == PROCESS_ERR_OK) //详情见 2.3, 即把事件
PROCESS_EVENT_TIMER 加入到事件队列
55.             {
56.                 /*成功加入事件队列*/
57.
58.                 t->p = PROCESS_NONE; //如果 etimer 的 p 指向的是 PROCESS_NONE, 则表示该 etimer 已到期。用于后
续 etimer_expired() 函数判断该 etimer 是否到期, etimer_expired(), 也见 2.2
59.
60.                 if (u != NULL)
61.                 {
62.                     u->next = t->next;
63.                 }
64.                 else
65.                 {
66.                     timerlist = t->next;
67.                 }
68.                 t->next = NULL;
69.                 update_time(); /*见 2.4, 即求出下一个到期时间 next_expiration (全局静态变量), 即还有
next_expiration 时间, 就有 timer 到期*/
70.                 goto again;
71.             }
72.             else
73.             {
```

```
74.             etimer_request_poll(); //详情见 2.5 若加入事件 PROCESS_EVENT_TIMER 出错(可能是事件队列已满),
    执行 etimer_request_poll(), 即 process_poll(&etimer_process), 使其有更高的优先级
75.         }
76.     }
77.     u = t;
78. }
79.
80. }
81.
82. PROCESS_END();
83. }
```

## 2.1 PROCESS\_YIELD 宏

```
1. /*即进程主动让出执行权*/
2. #define PROCESS_YIELD() PT_YIELD(process_pt)
3.
4. /*继续展开如下, Yield from the current protothread*/
5. #define PT_YIELD(pt) \
6. do { \
7. PT_YIELD_FLAG = 0; \
8. LC_SET((pt)->lc); \
9. if(PT_YIELD_FLAG == 0) { \
10. return PT_YIELDED; \
11. } \
12. } while(0)
```

13.

14. `#define LC_SET(s) s = __LINE__; case __LINE__:` /\*保存被中断的行数\*/

## 2.2 timer\_expired 和 etimer\_expired

### 2.2.1 timer\_expired

```
1. /*即检查 timer 是不是到期了，若到期返回 1，否则返回 0*/
2. int timer_expired(struct timer *t)
3. {
4.     clock_time_t diff = (clock_time() - t->start) + 1;
5.     return t->interval < diff;
6. }
```

注：为什么不直接 `return diff >= t->interval`，源代码注释给出原因，如下：

1. Note: Can not return `diff >= t->interval` so we add 1 to diff and return `t->interval < diff` - required to avoid an internal error in mspgcc.

### 2.2.2 etimer\_expired

`etimer_expired` 跟 `timer_expired` 不一样，后者到期是指定时器超时(通过比较当前时间与 `start+interval` 来判断 timer 是否到期)，而前者到期 是指已发送事件 `PROCESS_EVENT_TIMER` 给 `etimer` 绑定的进程(通过 `etimer` 的 `p` 指向是否为 `PROCESS_NONE` 来判断)。源代码如下：

```
1. int etimer_expired(struct etimer *et)
2. {
```

```
3.     return et->p == PROCESS_NONE;
4. }
```

### 2.3 process\_post 函数

```
1. //即把把事件加入到事件队列
2. int process_post(struct process *p, process_event_t ev, process_data_t data)
3. {
4.     static process_num_events_t snum;
5.
6.     /*调试信息, 直接无视*/
7.     if (PROCESS_CURRENT() == NULL)
8.     {
9.         PRINTF(
10.             "process_post: NULL process posts event %d to process '%s', nevents %d\n",
11.             ev, PROCESS_NAME_STRING(p), nevents);
12.     }
13.     else
14.     {
15.         PRINTF(
16.             "process_post: Process '%s' posts event %d to process '%s', nevents %d\n",
17.             PROCESS_NAME_STRING(PROCESS_CURRENT()), ev, p == PROCESS_BROADCAST ?
18.             "<broadcast>": PROCESS_NAME_STRING(p), nevents);
19.     }
20.
21.     /*事件队列已满, 返回 PROCESS_ERR_FULL, 即 1*/
```

```
22.  if (nevents == PROCESS_CONF_NUMEVENTS)
23.  {
24.      #if DEBUG
25.          if (p == PROCESS_BROADCAST)
26.          {
27.              printf(
28.                  "soft panic: event queue is full when broadcast event %d was posted from %s\n", ev,
PROCESS_NAME_STRING(process_current));
29.          }
30.          else
31.          {
32.              printf(
33.                  "soft panic: event queue is full when event %d was posted to %s frpm %s\n", ev,
PROCESS_NAME_STRING(p), PROCESS_NAME_STRING(process_current));
34.          }
35.          #endif /* DEBUG */
36.          return PROCESS_ERR_FULL;
37.      }
38.
39.      /*将事件加入事件队列，并绑定进程 p*/
40.      snum = (process_num_events_t) (fevent + nevents) % PROCESS_CONF_NUMEVENTS;
41.      //事件队列用环形组织
42.      events[snum].ev = ev;
43.      events[snum].data = data;
44.      events[snum].p = p;
45.      ++nevents;
```

```
46.  
47.  #if PROCESS_CONF_STATS  
48.      if (nevents > process_maxevents)  
49.      {  
50.          process_maxevents = nevents;  
51.      }  
52.  #endif /* PROCESS_CONF_STATS */  
53.  
54.  return PROCESS_ERR_OK;  
55. }
```

## 2.4 update\_time 函数

```
1.  /*即求出下一个到期时间 next_expiration(全局静态变量)，即还有 next_expiration 时间，就有 timer 到期*/  
2.  static void update_time(void)  
3.  {  
4.      clock_time_t tdist;  
5.      clock_time_t now;  
6.      struct etimer *t;  
7.  
8.      if (timerlist == NULL)  
9.      {  
10.         next_expiration = 0;  
11.     }  
12.     else  
13.     {
```

```
14.     now = clock_time();
15.     t = timerlist;
16.
17.     /*遍历 timerlist, 找出最近到期的 timer, 并求得下一个到期时间 next_expiration*/
18.     tdist = t->timer.start + t->timer.interval - now; //还有 tdist 就到期了
19.     for (t = t->next; t != NULL; t = t->next)
20.     {
21.         if (t->timer.start + t->timer.interval - now < tdist)
22.         {
23.             tdist = t->timer.start + t->timer.interval - now;
24.         }
25.     }
26.     next_expiration = now + tdist;
27. }
28. }
```

## 2.5 etimer\_request\_poll 函数

```
1. void etimer_request_poll(void)
2. {
3.     process_poll(&etimer_process);
4. }
5.
6. //将进程 p 的 needspoll 设为 1, 使其获得更高优先级, 即让 etimer_process 更快地再次获得执行权
7. void process_poll(struct process *p)
8. {
```

```
9.     if (p != NULL)
10.    {
11.        if (p->state == PROCESS_STATE_RUNNING || p->state == PROCESS_STATE_CALLED)
12.        {
13.            p->needspoll = 1;
14.            poll_requested = 1; //全局变量，标识是否有 needspoll 为 1 的进程
15.        }
16.    }
17. }
```

[Contiki学习笔记：深入理解process\\_run函数](#) (2011-10-26 18:48)

标签： [Contiki](#) [do\\_event](#) [do\\_poll](#) [process\\_run](#) [学习笔记](#) 分类： [Contiki](#)

### 摘要：

process\_run 用于处理系统所有 needspoll 标记为 1 的进程及处理事件队列的下一个事件。本文深入原码，详细分析，也包括 do\_poll 和 do\_event 函数。

## 一、运行 process\_run

```
1. int main()
```



```
2. {
3.     dbg_setup_uart();
4.     usart_puts("Initialising\n");
5.
6.     clock_init();
7.     process_init();
8.     process_start(&etimer_process, NULL);
9.     autostart_start(autostart_processes);
10.
11.     while (1)
12.     {
13.         /*执行完所有 needspoll 为 1 的进程及处理完所有队列*/
14.         do
15.         {
16.         }
17.         while (process_run() > 0);
18.     }
19.     return 0;
20. }
```

## 二、process\_run 剖析

process\_run 处理系统所有 needspoll 标记为 1 的进程及处理事件队列的下一个事件，源代码如下：

```
1. static volatile unsigned char poll_requested; //全局静态变量，标识系统是否有 needspoll 为 1 的进程
2.
```

```
3. int process_run(void)
4. {
5.     if (poll_requested) //进程链表有 needspoll 为 1 的进程
6.     {
7.         do_poll(); //见 2.1
8.     }
9.
10.    do_event(); //见 2.2
11.
12.    return nevents + poll_requested; //若和为 1，则表示处理完系统的所有事件，并且没有 needspoll 为 1 的进程
13.}
```

透过上述的源代码，可以直观看出 needspoll 标记为 1 的进程可以优先执行。并且每执行一次 process\_run，将处理系统所有 needspoll 标记为 1 的进程，而只处理事件队列的一个事件。

## 2.1 do\_poll 函数

复位全局变量 poll\_requested，遍历整个进程链表，将 needspoll 标记为 1 的进程投入运行，并将相应的 needspoll 复位。源代码如下：

```
1. static void do_poll(void)
2. {
3.     struct process *p;
4.     poll_requested = 0; //复位全局变量
5.
6.     for (p = process_list; p != NULL; p = p->next) //处理所有 needspoll 为 1 的进程
7.     {
```

```
8.         if (p->needspoll) //将 needspoll 为 1 的进程投入执行
9.         {
10.            p->state = PROCESS_STATE_RUNNING;
11.            p->needspoll = 0;
12.            call_process(p, PROCESS_EVENT_POLL, NULL);
13.        }
14.    }
15. }
```

call\_process函数剖析见博文《[Contiki学习笔记：启动一个进程process\\_start](#)》第三部分。

## 2.2 do\_event 函数

do\_event 处理事件队列的一个事件，有两种事件需特殊处理：PROCESS\_BROADCAST 和 PROCESS\_EVENT\_INIT。前者是广播事件，需处理所有进程，后者是初始化事件，需将进程状态设为 PROCESS\_STATE\_RUNNING，以确保该进程被进程了。源代码如下：

```
1. static process_num_events_t nevents; /*事件队列的总事件数 */
2. static process_num_events_t fevent; /*指向下一个要传递的事件的位置*/
3. static struct event_data events[PROCESS_CONF_NUMEVENTS]; /*事件队列，用数组存储，逻辑上是环形队列*/
4.
5. static void do_event(void)
6. {
7.     /*以下 3 个变量恰为 struct event_data 的成员，用于暂存即将处理(fevent 事件)的值*/
8.     static process_event_t ev;
9.     static process_data_t data;
10.    static struct process *receiver;
11. }
```

```
12. static struct process *p;
13.
14. if (nevents > 0)
15. {
16.     /*提取将要处理事件的成员变量*/
17.     ev = events[fevent].ev;
18.     data = events[fevent].data;
19.     receiver = events[fevent].p;
20.
21.     fevent = (fevent + 1) % PROCESS_CONF_NUMEVENTS; //更新 fevent (指向下一个待处理的事件, 类型于微机的 PC)
22.     --nevents; //事件队列被组织成环形队列, 所以取余数
23.
24.     if (receiver == PROCESS_BROADCAST) //如果事件是广播事件 PROCESS_BROADCAST, 则处理所有进程
25.     {
26.         for (p = process_list; p != NULL; p = p->next)
27.         {
28.             if (poll_requested)
29.             {
30.                 do_poll();
31.             }
32.             call_process(p, ev, data); //jelline note: call the receiver process twice??
33.         }
34.     }
35.     else
36.     {
37.         if (ev == PROCESS_EVENT_INIT) //若事件是初始化, 设置进程状态, 确保进程状态为 PROCESS_STATE_RUNNING
```

```
38.         {  
39.             receiver->state = PROCESS_STATE_RUNNING;  
40.         }  
41.         call_process(receiver, ev, data);  
42.     }  
43. }  
44. }
```

call\_process函数剖析见博文《[Contiki学习笔记：启动一个进程process start](#)》第三部分。

[Contiki学习笔记：新事件产生及事件处理](#) (2011-10-27 15:47)

标签： [Contiki](#) [学习笔记](#) [新事件产生](#) [事件处理](#) 分类： [Contiki](#)

### 摘要：

本文介绍了 Contiki 系统怎么产生新事件，以及怎样处理事件。

## 一、事件加入及处理

理解以下内容，先注意下如下两个全局静态变量：

1. `static process_num_events_t nevents;` /\*未处理的事件总数\*/
2. `static process_num_events_t fevent;` /\*标识下一个要传递的事件的位置，事实上是数组的下标\*/
- 3.
4. `typedef unsigned char process_num_events_t;`

## 二、新事件产生

将事件加入到事件队列主要由`process_post()`函数完成，详情见博文《[Contiki学习笔记：系统进程timer\\_process](#)》2.3，代码精简后如下：

1. `static process_num_events_t snum;`
2. `snum = (process_num_events_t)(fevent + nevents) % PROCESS_CONF_NUMEVENTS;`
3. `events[snum].ev = ev;`
4. `events[snum].data = data;`
5. `events[snum].p = p;`
6. `++nevents;`

## 三、处理事件

事件处理由`do_event()`函数完成，`do_event()`把下一个事件（`fevent` 指向的事件）从事件队列里取出来，然后传递给它对应的监听进程。把`fevent`往前移一位置（即++），把`nevents`减1。详情见博文《[Contiki学习笔记：深入理解process\\_run函数](#)》2.2。

标签: [Contiki](#) [Systick\\_isr](#) [学习笔记](#) [时钟中断](#) [处理程序](#) 分类: [Contiki](#)

## 摘要:

本文基于 STM32F103RBT6 微控制器, 深入源码, 详细分析了时钟中断处理程序 Systick\_isr。

## 一、Systick\_isr 源代码

```
1. static volatile clock_time_t current_clock = 0;
2. static volatile unsigned long current_seconds = 0;
3. static unsigned int second_countdown = CLOCK_SECOND; //详情见 2.1
4.
5. /**时钟中断处理程序***/
6. void SysTick_Handler(void)
7. {
8.     usart_puts("systick_isr\n");
9.
10.    SCB->ICSR = SCB_ICSR_PENDSTCLR; //详情见 2.2
11.
12.    current_clock++;
13.    if (etimer_pending() && etimer_next_expiration_time() <= current_clock) //timerlist 不为空且还没有 etimer 到
    期, 则执行 etimer_request_poll。详情见 2.3
14.    {
15.        etimer_request_poll(); //详情见 2.4
16.    }
```

```
17.  if (--second_countdown == 0)
18.  {
19.      current_seconds++;
20.      second_countdown = CLOCK_SECOND; //重置 second_countdown
21.  }
22. }
```

注：时钟中断处理程序与硬件相关，我用的是 STM32F103RBT6。

## 二、源码分析

### 2.1 CLOCK\_SECOND 宏

CLOCK\_SECOND 定义了 1 秒钟产生时钟中断的次数，默认情况是 32 次，可以通过宏 CLOCK\_CONF\_SECOND 来配置。源代码如下：

```
1.  /*CLOCK_SECOND 宏定义*/
2.  #ifdef CLOCK_CONF_SECOND
3.      #define CLOCK_SECOND CLOCK_CONF_SECOND
4.  #else
5.      #define CLOCK_SECOND (clock_time_t)32
6.  #endif
7.
8.  #define CLOCK_CONF_SECOND 10 /*define systick clock interrupt times per second*/
```

### 2.2 SCB & ICSR



该语句主要对寄存器进行设置,SCB 指 System Control Block,ICSR 指 Interrupt Control State Register,SCB\_ICSR\_PENDSTCLR 是 Clear pending SysTick bit。

```
1. #define SCB_ICSR_PENDSTCLR ((uint32_t)0x02000000) /*Clear pending SysTick bit */
```

### 2.3 etimer\_pending 函数和 etimer\_next\_expiration\_time 函数

etimer\_pending 函数检查 timerlist 是否为空(若返回 true 表示不为空), etimer\_next\_expiration\_time 函数返回 next\_expiration(即到了 next\_expiration 才有 etimer 到期。另,若 timerlist 为空则返回 0),并与系统当前时间 current\_clock 比较,若 next\_expiration 小于等于 current\_clock,则表明还没有 etimer 到期,于是把系统进程 etimer\_process 的 needspoll 设为 1,使其具有更快地再次获得执行。

#### 2.3.1 etimer\_pending 函数

检查 timerlist 是否为空,若不为空则返回 true,否则返回 false,源码如下:

```
1. /*Check if there are any non-expired event timers*/  
2. int etimer_pending(void) //如果返回 true 则表明 timerlist 不为空  
3. {  
4.     return timerlist != NULL;  
5. }
```

#### 2.3.2 etimer\_next\_expiration\_time 函数

next\_expiration 是指 timerlist 下一个到期的时间,即到了 next\_expiration 就会有 etimer 到期,由 update\_time 计算,详情见博文《[Contiki 学习笔记: 系统进程 etimer\\_process](#)》。

```
1. /*得到 next_expiration, 如果 timerlist 为空, 则返回 0*/
2. clock_time_t etimer_next_expiration_time(void)
3. {
4.     return etimer_pending() ? next_expiration : 0;
5. }
```

## 2.4 etimer\_request\_poll 函数

将系统进程 etimer\_process 的 needspoll 设为 1, 使其获得更高优先级, 即让 etimer\_process 更快地再次获得执行。详情见博文《[Contiki 学习笔记: 系统进程 etimer\\_process](#)》。

[Contiki 学习笔记: protothread 状态](#) (2011-10-28 16:52)

标签: [Contiki](#) [学习笔记](#) [protothread](#) [PT\\_YIELDED](#) [PT\\_EXITED](#) 分类: [Contiki](#)

### 摘要:

本文结合 Contiki OS 实例分析 protothread 四种状态: PT\_WAITING、PT\_YIELDED、PT\_EXITED、PT\_ENDED, 并给出 Contiki 事件相关函数与 protothread 状态, 最后给出系统编程的参考 API。

## 一、PT 四种状态

1. #define PT\_WAITING 0
2. #define PT\_YIELDED 1
3. #define PT\_EXITED 2
4. #define PT\_ENDED 3

Contiki 事件相关函数与 protothread 状态关系如下图(注-没有包含所有函数):

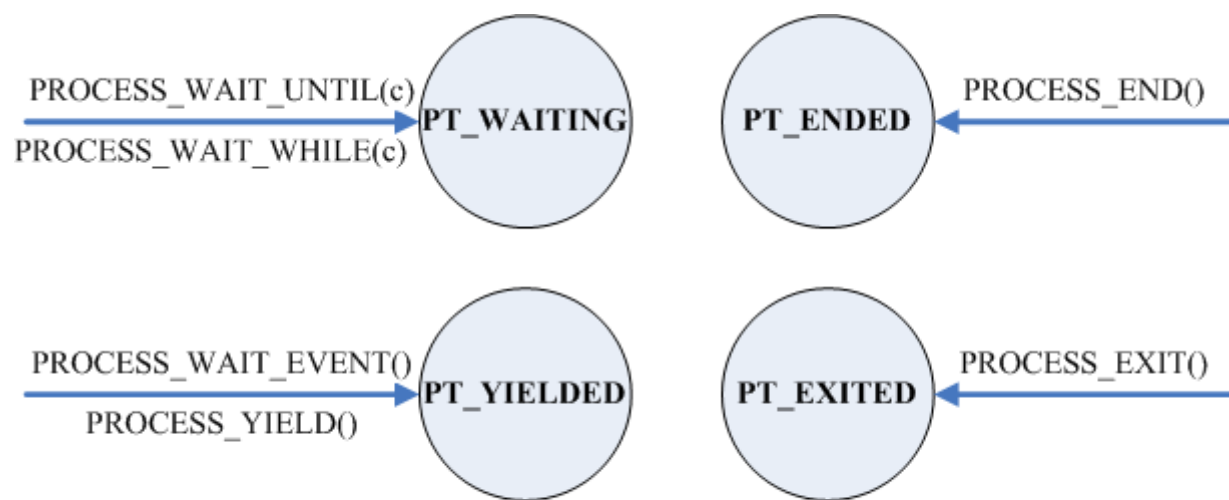


图 Contiki事件相关函数与protothread状态关系

### 1.1 PT\_WAITING

宏 PROCESS\_WAIT\_UNTIL(c)用到了 PT\_WAITING, 该宏用于挂起进程直到条件 c 成立, 源码如下:

1. `#define PROCESS_WAIT_UNTIL(c) PT_WAIT_UNTIL(process_pt, c)`

宏 `PROCESS_WAIT_UNTIL(c)` 不保证进程会让出执行权，源码注释如下：

1. This macro does not guarantee that the process yields, and should therefore be used with care. In most cases, `PROCESS_WAIT_EVENT()`, `PROCESS_WAIT_EVENT_UNTIL()`, `PROCESS_YIELD()` or `PROCESS_YIELD_UNTIL()` should be used instead.

宏 `PT_WAIT_UNTIL` 展开如下：

```

1. #define PT_WAIT_UNTIL(pt, condition) \
2.     do \
3.     { \
4.         LC_SET((pt)->lc); \
5.         if(!(condition)) \
6.         { \
7.             return PT_WAITING; \
8.         } \
9.     } while(0)

```

在 `PT_RESTART` 宏也用到了 `PT_WAITING`，源码如下：

```

1. //Restart the protothread.
2. #define PT_RESTART(pt) \
3.     do \
4.     { \
5.         PT_INIT(pt); \
6.         return PT_WAITING; \

```

7.       } while(0)

## 1.2 PT\_YIELDED

(1) PT\_YIELD\_FLAG = 0

表示条件不满足，主动让出执行权。PT\_END(pt)、PT\_YIELD(pt)、PT\_YIELD\_UNTIL(pt, cond)将 PT\_YIELD\_FLAG 设置成 0

(2) PT\_YIELD\_FLAG =1

表示条件满足，继续执行后续代码。在 PT\_BEGIN(pt)将 PT\_YIELD\_FLAG 设置成 1

为了便好理解 PT\_YIELD\_FLAG，以 PROCESS\_WAIT\_EVENT 宏作为分析对象，源代码如下：

```
1. #define PROCESS_WAIT_EVENT()      PROCESS_YIELD()
2. #define PROCESS_YIELD()          PT_YIELD(process_pt)
3.
4. #define PT_YIELD(pt) \
5. do
6. { \
7. PT_YIELD_FLAG = 0; \
8. LC_SET((pt)->lc); \
9. if(PT_YIELD_FLAG == 0)
10. { \
11. return PT_YIELDED; \
12. } \
13. } while(0)
```

14.

15. #define LC\_SET(s) s = \_\_LINE\_\_; case \_\_LINE\_\_: //保存程序断点，下次再运行该进程直接跳到 case \_\_LINE\_\_

PROCESS\_WAIT\_EVENT宏用于等待一个事件发生，如果检测到PT\_YIELD\_FLAG为1，就继续执行PROCESS\_WAIT\_EVENT宏后面的代码。若PT\_YIELD\_FLAG为0了，就直接返回PT\_YIELDED，从数 《Contiki学习笔记:启动一个进程process\_start》第三部分call\_process函数，可知并没有退出进程，只是把进程状态设为PROCESS\_STATE\_RUNNING。call\_process(struct process \*p, process\_event\_t ev, process\_data\_t data)函数部分代码如下：

```
1. ret = p->thread(&p->pt, ev, data); //才真正执行 PROCESS_THREAD(name, ev, data)定义的内容
2. if (ret == PT_EXITED || ret == PT_ENDED || ev == PROCESS_EVENT_EXIT)
3. {
4.     exit_process(p, p); //如果返回值表示退出、结尾或者遇到 PROCESS_EVENT_EXIT，进程退出
5. }
6. else
7. {
8.     p->state = PROCESS_STATE_RUNNING; //进程挂起等待事件
9. }
```

这 有个疑问，什么时候把 PT\_YIELD\_FLAG 设为 1，才能以继续执行 PROCESS\_WAIT\_EVENT 宏后面的代码。我也纳闷，在整个源码，只找到 PT\_BEGIN 宏将其设 1。从源码分析，该进 程没有机会执行 PROCESS\_WAIT\_EVENT 宏后面的内容，只有其他进程传递一个事件 PROCESS\_EVENT\_EXIT 让其退出，即 process\_post(&hello\_world\_process, PROCESS\_EVENT\_EXIT, NULL)。但实际编辑，只要向该进程传递一个普通事件，即可执行 PROCESS\_WAIT\_EVENT 宏后面的代码。很是不解，求助中……。 (实例见附 录)

### 1.3 PT\_EXITED

宏 PROCESS\_EXIT(让当前进程退出，自己结束自己的生命)用到了 PT\_EXITED，源码如下：

```
1. #define PT_EXIT(pt) \
2.     do \
3.     { \
4.         PT_INIT(pt); \
5.         return PT_EXITED; \
6.     } while(0)
```

执行进程主体 thread 时 (call\_process 函数)，会返回结果，如果结果是 PT\_EXITED 或者 1.4 的 PT\_ENDED，则进程退出 (exit\_process 函数)，部分源码如下：

#### 1.4 PT\_ENDED

这个就不陌生了，在宏 PROCESS\_END 展开就有这么一句 return PT\_ENDED，一层层展开如下：

```
1. #define PROCESS_END() PT_END(process_pt)
2.
3. #define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0; \
4.     PT_INIT(pt); return PT_ENDED; }
```

## 二、编程指南

理解上述内容后，相信对多个进程间事件如何交互比较清楚了，在实际编程过程，可以参考如下两个网址，分别列出了 pt 及 process 头文件的 API：

- [1] [Contiki 2.5: core/sys/process.h File Reference](#)
- [2] [Contiki 2.5: core/sys/pt.h File Reference](#)

附 PROCESS\_EVENT\_EXIT 测试用例:

```
1. //filename:process_wait_event_test.c
2. #include "contiki.h"
3. #include "debug-uart.h"
4.
5. PROCESS(hello_world_process, "Hello world");
6. PROCESS(post_event_process, "Post event");
7.
8. AUTOSTART_PROCESSES(&hello_world_process, &post_event_process);
9.
10. PROCESS_THREAD(hello_world_process, ev, data)
11. {
12.     PROCESS_BEGIN();
13.     usart_puts("Hello, world!\n");
14.
15.     while(1)
16.     {
17.         //usart_puts("while.Before wait event!\n");
18.         PROCESS_WAIT_EVENT();
19.         usart_puts("Have been posted an event!\n");
20.     }
21.
22.     PROCESS_END();
```



```
23. }
24.
25.
26. PROCESS_THREAD(post_event_process, ev, data)
27. {
28.     PROCESS_BEGIN();
29.     usart_puts("Post event!\n");
30.
31.     static process_event_t event_post;
32.     event_post = process_alloc_event();
33.
34.     process_post(&hello_world_process, event_post, NULL);
35.     //process_post(&hello_world_process, PROCESS_EVENT_EXIT, NULL);
36.
37.     PROCESS_END();
38. }
```

[Contiki学习笔记：进程状态](#) (2011-10-30 15:18)

标签： [Contiki](#) [学习笔记](#) [进程状态](#) [PROCESS\\_STATE\\_NONE](#) [PROCESS\\_STATE\\_RUNNI](#) 分类： [Contiki](#)

摘要：

本文分析了 Contiki OS 进程 3 种状态 `PROCESS_STATE_NONE`、`PROCESS_STATE_RUNNING`、`PROCESS_STATE_CALLED`，并给出进程状态转换图。

## 一、进程状态概述

Contiki 是事件驱动内核，并基于 protothread 机制提供类线程编程风格 (a thread-like programming style)，在博文 [《Contiki 学习笔记：protothread 状态》](#) 已介绍了 protothread 状态，但进程的状态与 protothread 不尽相同，只有 3 种状态，如下：

1. `#define PROCESS_STATE_NONE 0`
2. `#define PROCESS_STATE_RUNNING 1`
3. `#define PROCESS_STATE_CALLED 2`

`PROCESS_STATE_NONE` 是指进程不处于运行状态，而 `PROCESS_STATE_RUNNING` 和 `PROCESS_STATE_CALLED` 都属于运行状态，区别在于：前者确实实在执行，即获得执行权；后者并没有真正在执行，可能是阻塞到某个事件了(???)。进程状态转换图如下，详情见下述分析：

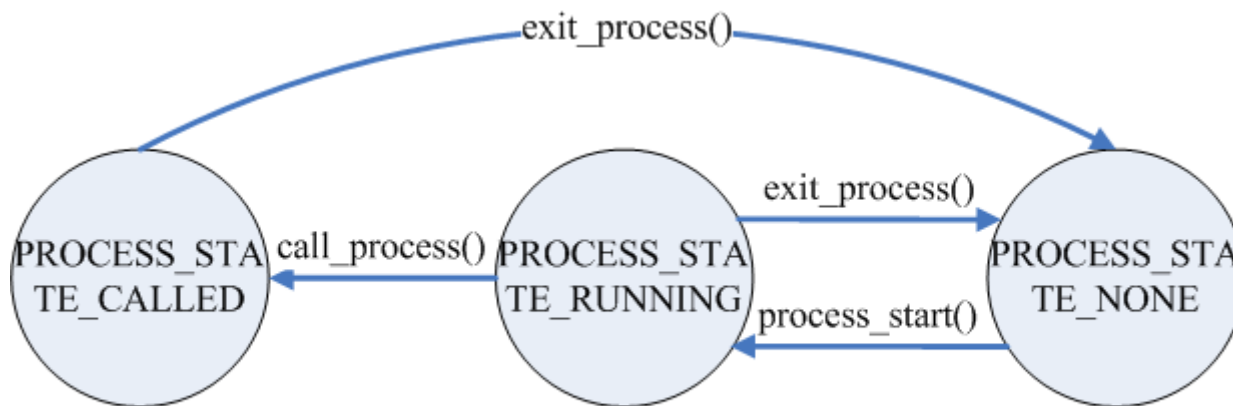


图1 Contiki进程状态转换图

## 二、PROCESS STATE

### 2.1 PROCESS\_STATE\_NONE

进程退出(但此时还没从进程链表删除),先将进程状态设为PROCESS\_STATE\_NONE,而后再从进程链表删除,详情见博文《[Contiki 学习笔记: 启动一个进程process\\_start](#)》第四部分exit\_process函数。部分源码如下:

```
1. if (process_is_running(p))
2. {
3.     p->state = PROCESS_STATE_NONE;
```

process\_is\_running 函数用于判断进程是否处于运行状态(包括 PROCESS\_STATE\_RUNNING 和 PROCESS\_STATE\_CALLED)，看源码就很清楚了：

```
1. int process_is_running(struct process *p)
2. {
3.     return p->state != PROCESS_STATE_NONE;
4. }
```

除此之外，声明一个进程时，进程状态默认也为 PROCESS\_STATE\_NONE。在博文《[Contiki学习笔记：实例hello world剖析](#)》讨论中，宏PROCESS用于声明一进程函数主体和定义一进程，宏展开如下：

```
1. #define PROCESS(name, strname) PROCESS_THREAD(name, ev, data); \
2. struct process name = { NULL, strname, process_thread_##name }
```

定义进程name，只初始化前3个变量，其余的缺省为0，当然也包括进程状态，而PROCESS\_STATE\_NONE又被define为0。关于进程结构体可参见博文《[Contiki学习笔记：主要数据结构之进程](#)》。

## 2.2 PROCESS\_STATE\_RUNNING

启动一个进程(由函数process\_start完成)，将进程加入进程链表process\_list，而后把进程状态设为PROCESS\_STATE\_RUNNING，call\_process函数(真正执行进程主体thread)会根据进程状态决定是否执行进程主体 thread，详情见博文《[Contiki学习笔记：启动一个进程process\\_start](#)》。即进程获得执行权的时候，先把进程状态设为PROCESS\_STATE\_RUNNING，而后再正在执行进程主体thread。

## 2.3 PROCESS\_STATE\_CALLED

在 call\_process 函数中，在执行进程主体 thread 前，先把进程设为 PROCESS\_STATE\_CALLED，部分源码如下：

```
1. if ((p->state &PROCESS_STATE_RUNNING) && p->thread != NULL)
2. {
3.     process_current = p;
4.     p->state = PROCESS_STATE_CALLED;
5.
6.     ret = p->thread(&p->pt, ev, data);
7.     if (ret == PT_EXITED || ret == PT_ENDED || ev == PROCESS_EVENT_EXIT)
8.     {
9.         exit_process(p, p);
10.    }
11.    else
12.    {
13.        p->state = PROCESS_STATE_RUNNING;
14.    }
15. }
```

源代码很少地方用到 PROCESS\_STATE\_CALLED，我也不太懂其中的含义，会不会用于标识进程被某一事件再次调用的标志，佐证如下：

```
1. //call_process 函数的部分代码
2. #if DEBUG
3.     if (p->state == PROCESS_STATE_CALLED)
4.     {
5.         printf("process: process '%s' called again with event %d\n",
6.             PROCESS_NAME_STRING(p), ev);
7.     }
```

## 8. #endif

[Contiki学习笔记: 编程模式](#) (2011-10-30 18:57)

标签: [Contiki](#) [代码框架](#) [学习笔记](#) [程序基本结构](#) [编程模式](#) 分类: [Contiki](#)

### 摘要:

本文先给出 Contiki 编程的代码框架, 接着, 介绍了用宏实现三种程序基本结构(即顺序、选择、循环)的基本框架, 最后介绍了挂起进程相关的 API。

## 一、代码框架

Contiki 实际编程通常只需替代 Hello world 的内容, main 函数内容甚至无需修改, 大概的代码框架如下(以两个进程为例):

1. `/*步骤 1: 包含需要的头文件*/`
2. `#include "contiki.h"`
3. `#include "debug-uart.h"`
- 4.
5. `/*步骤 2: 用 PROCESS 宏声明进程执行主体, 并定义进程*/`
6. `PROCESS(example_1_process, "Example 1"); //PROCESS(name, strname)`
7. `PROCESS(example_2_process, "Example 1");`

```
8.
9. /*步骤 3: 用 AUTOSTART_PROCESSES 宏让进程自启动*/
10. AUTOSTART_PROCESSES(&example_1_process, &example_2_process); //AUTOSTART_PROCESSES(...)
11.
12. /*步骤 4: 定义进程执行主体 thread*/
13. PROCESS_THREAD(example_1_process, ev, data) //PROCESS_THREAD(name, ev, data)
14. {
15.     PROCESS_BEGIN(); /*代码总是以宏 PROCESS_BEGIN 开始*/
16.
17.     /*example_1_process 的代码*/
18.
19.     PROCESS_END(); /*代码总是以宏 PROCESS_END 结束*/
20. }
21.
22. PROCESS_THREAD(example_2_process, ev, data)
23. {
24.     PROCESS_BEGIN();
25.
26.     /*example_2_process 的代码*/
27.
28.     PROCESS_END();
29. }
```

可参考博文《[Contiki学习笔记: 实例hello world剖析](#)》以获得更直观的认识, 接下来给出三种程序基本结构(即顺序、选择、循环)的模式。

## 二、顺序&选择&循环

### 2.1 顺序

```
1. PROCESS_BEGIN();
2. (*...*)
3. PROCESS_WAIT_UNTIL(cond1); //注 1
4. (*...*)
5. PROCESS_END();
```

### 2.2 选择

```
1. PROCESS_BEGIN();
2. (*...*)
3. while (cond1)
4.     PROCESS_WAIT_UNTIL(cond1 or cond2); //注 1
5. (*...*)
6. PROCESS_END();
```

### 2.3 循环

```
1. PROCESS_BEGIN();
2. (*...*)
3. if (condtion)
4.     PROCESS_WAIT_UNTIL(cond2a); //注 1
5. else
6.     PROCESS_WAIT_UNTIL(cond2b); //注 1
```



7. `(*...*)`
8. `PROCESS_END()`;

注 1: 这里不一定非得用宏 `PROCESS_WAIT_UNTIL`, 事实上有很多选择, 比如宏 `PROCESS_WAIT_EVENT_UNTIL(c)`、宏 `PROCESS_WAIT_EVENT()`、宏 `PROCESS_YIELD_UNTIL(c)` 等(请参见本文第三部分), 实际编程应根据实际情况加以选择。

### 三、挂起进程相关 API

#### 3.1 概述

表 1 给出挂起进程相关 API 的功能描述, 事实上, 实际编程所关心的是, 什么时候继续执行宏后面的内容, 3.2~3.6 给出了详见分析。

表 1 Contiki 挂起进程相关 API[1]

<code>PROCESS_WAIT_EVENT()</code>	Wait for an event to be posted to the process.
<code>PROCESS_YIELD()</code>	Yield the currently running process.
<code>PROCESS_WAIT_EVENT_UNTIL(c)</code>	Wait for an event to be posted to the process, with an extra condition.
<code>PROCESS_YIELD_UNTIL(c)</code>	Yield the currently running process until a condition occurs.
<code>PROCESS_WAIT_UNTIL(c)</code>	Wait for a condition to occur.
<code>PROCESS_WAIT_WHILE(c)</code>	Block and wait while condition is true.
<code>PROCESS_PT_SPAWN(pt, thread)</code>	Spawn a protothread from the process.
<code>PROCESS_PAUSE()</code>	Yield the process for a short while.

### 3.2 PROCESS\_WAIT\_EVENT 和 PROCESS\_YIELD

从代码展开来看，宏 PROCESS\_WAIT\_EVENT 和宏 PROCESS\_YIELD 实现的功能是一样的(或者说 PROCESS\_WAIT\_EVENT 只是 PROCESS\_YIELD 的一个别名)，只是两种不同的描述。也就是说当 PT\_YIELD\_FLAG 为 1 时，才执行宏后面的代码，否则返回。代码展开如下：

```

1. #define PROCESS_WAIT_EVENT() PROCESS_YIELD()
2. #define PROCESS_YIELD() PT_YIELD(process_pt)
3.
4. #define PT_YIELD(pt) \
5.     do \
6.     { \
7.         PT_YIELD_FLAG = 0; \
8.         LC_SET((pt)->lc); \
9.         if(PT_YIELD_FLAG == 0) \
10.        { \
11.            return PT_YIELDED; \
12.        } \
13.    } while(0)

```

### 3.3 PROCESS\_WAIT\_EVENT\_UNTIL 和 PROCESS\_YIELD\_UNTIL

跟 3.2 类似，宏 PROCESS\_WAIT\_EVENT\_UNTIL 和宏 PROCESS\_YIELD\_UNTIL 是一组，在 3.2 的基础是增加了额外的一个条件，也就是说当 PT\_YIELD\_FLAG 为 1 或条件为 true 时，才执行宏后面的代码，否则返回。代码展开如下：

```

1. #define PROCESS_WAIT_EVENT_UNTIL(c) PROCESS_YIELD_UNTIL(c)
2. #define PROCESS_YIELD_UNTIL(c) PT_YIELD_UNTIL(process_pt, c)

```

```

3.
4.     #define PT_YIELD_UNTIL(pt, cond)          \
5.     do                                         \
6.     {                                         \
7.         PT_YIELD_FLAG = 0;                   \
8.         LC_SET((pt)->lc);                     \
9.         if((PT_YIELD_FLAG == 0) || !(cond))   \
10.        {                                     \
11.            return PT_YIELDED;                 \
12.        }                                     \
13.    } while(0)

```

### 3.4 PROCESS\_WAIT\_UNTIL 和 PROCESS\_WAIT\_WHILE

从代码展开来看, 宏 PROCESS\_WAIT\_UNTIL 和宏 PROCESS\_WAIT\_WHILE 判断的条件相反, PROCESS\_WAIT\_UNTIL 宏当条件为真时 (即某个事件发生), 执行宏后面的内容。而 PROCESS\_WAIT\_WHILE 宏当条件为假时, 执行宏后面的内容 (即当条件为真时, 阻塞 该进程)。

```

1. /*PROCESS_WAIT_UNTIL 宏展开*/
2. #define PROCESS_WAIT_UNTIL(c) PT_WAIT_UNTIL(process_pt, c)
3.
4. /*PROCESS_WAIT_WHILE 宏展开*/
5. #define PROCESS_WAIT_WHILE(c) PT_WAIT_WHILE(process_pt, c)
6. #define PT_WAIT_WHILE(pt, cond) PT_WAIT_UNTIL((pt), !(cond))
7.
8. /*PT_WAIT_UNTIL 宏展开*/
9. #define PT_WAIT_UNTIL(pt, condition)          \

```

```

10.  do
11.  {
12.      LC_SET((pt)->lc);
13.      if(!(condition))
14.      {
15.          return PT_WAITING;
16.      }
17.  } while(0)

```

### 3.5 PROCESS\_PT\_SPAWN

PROCESS\_PT\_SPAWN 用于产生一个子 protothread，若执行完 thread 并退出 PT\_EXITED，则继续执行宏 PROCESS\_PT\_SPAWN 后面的内容。宏一层层展开如下：

```

1. #define PROCESS_PT_SPAWN(pt, thread) PT_SPAWN(process_pt, pt, thread)
2. #define PT_SPAWN(pt, child, thread) \
3.  do
4.  {
5.      PT_INIT((child));
6.      PT_WAIT_THREAD((pt), (thread));
7.  } while(0)
8.
9. #define PT_WAIT_THREAD(pt, thread) PT_WAIT_WHILE((pt), PT_SCHEDULE(thread))
10.
11. #define PT_WAIT_WHILE(pt, cond) PT_WAIT_UNTIL((pt), !(cond))
12.

```

13. #define PT\_SCHEDULE(f) ((f) < PT\_EXITED)

### 3.6 PROCESS\_PAUSE

宏 PROCESS\_PAUSE 用于向进程传递事件 PROCESS\_EVENT\_CONTINUE，并等到任一事件发生，即被挂起了(但我并没有看到哪个地方对事件 PROCESS\_EVENT\_CONTINUE 优先处理了)。宏展开如下如下：

```
1. #define PROCESS_PAUSE()
2. do
3. {
4.     process_post(PROCESS_CURRENT(), PROCESS_EVENT_CONTINUE, NULL);
5.     PROCESS_WAIT_EVENT();
6. } while(0)
```

### 3.7 PROCESS\_WAIT\_EVENT\_UNTIL 与 PROCESS\_WAIT\_UNTIL 区别

从上述 3.3 和 3.4 代码展开可以看出，宏 PROCESS\_WAIT\_EVENT\_UNTIL 和宏 PROCESS\_WAIT\_UNTIL 的区别在于：前者除了要判断条件外，还需判断 PT\_YIELD\_FLAG；而后者只需判断条件。并且返回值也不一样。

参考资料：

[1] [Contiki 2.5: core/sys/process.h File Reference](#)

[Contiki学习笔记：应用编程接口API](#) (2011-10-30 20:12)

标签： [API](#) [Contiki](#) [学习笔记](#) [应用编程接口](#) 分类： [Contiki](#)

## 摘要：

本文按自己理解的方式，对进程、事件、etimer 的 API 进行整理，以便编程。

网上有一份根据源码生成的文档[Contiki 2.5: The Contiki Operating System](#)(类似于ava SE 6 API Documentation)，分门别类给出接口(系统和启动代码调用的接口、硬件驱动调用的接口、应用编程接口)，本文只给出应用编程接口，并进行整理。另以下的大部分API在之前的博文几乎都有分析。

## 一、进程

### 1.1 进程声明和定义[1]

`PROCESS_THREAD(name, ev, data)`-----Define the body of a process.

`PROCESS_NAME(name)`-----Declare the name of a process.

`PROCESS(name, strname)`-----Declare a process.

### 1.2 process protothread 方法

`PROCESS_BEGIN()`-----Define the beginning of a process.

`PROCESS_END()`-----Define the end of a process.

`PROCESS_EXIT()`-----Exit the currently running process.

`PROCESS_CURRENT()`-----Get a pointer to the currently running process. `PROCESS_CONTEXT_BEGIN(p)`-----Switch context to another process. `PROCESS_CONTEXT_END(p)`-----End a context switch.

`PROCESS_POLLHANDLER(handler)`-----Specify an action when a process is polled. `PROCESS_EXITHANDLER(handler)`-----Specify an action when a process exits.

### 1.3 进程内核函数[2]

`void process_start (struct process *p, const char *arg)`-----Start a process.

`void process_exit (struct process *p)`-----Cause a process to exit.

### 1.4 挂起进程相关[1][3]

`PROCESS_WAIT_EVENT()`-----Wait for an event to be posted to the process. `PROCESS_WAIT_EVENT_UNTIL(c)`-----Wait for an event to be posted to the process, with an extra condition.

`PROCESS_YIELD()`-----Yield the currently running process.

`PROCESS_YIELD_UNTIL(c)`-----Yield the currently running process until a condition occurs.

`PROCESS_WAIT_UNTIL(c)`-----Wait for a condition to occur.

`PROCESS_WAIT_WHILE(c)`-----`PT_WAIT_WHILE(process_pt, c)`

`PROCESS_PT_SPAWN(pt, thread)`-----Spawn a protothread from the process.

`PROCESS_PAUSE()`-----Yield the process for a short while.

## 二、事件

### 2.1 新建事件[1]

`process_event_t process_alloc_event (void)`-----Allocate a global event number.

### 2.2 传递事件[1][2]

`int process_post (struct process *p, process_event_t ev, void *data)`-----Post an asynchronous event.

`void process_post_synch (struct process *p, process_event_t ev, void *data)`-----Post a synchronous event to a process.

## 三、etimer[4][5]

`void etimer_set (struct etimer *et, clock_time_t interval)`-----Set an event timer.

`void etimer_reset (struct etimer *et)`-----Reset an event timer with the same interval as was previously set.

`void etimer_restart (struct etimer *et)`-----Restart an event timer from the current point in time.



`void etimer_adjust (struct etimer *et, int td)`-----Adjust the expiration time for an event timer.

`void etimer_stop (struct etimer *et)`-----Stop a pending event timer.

`int etimer_expired (struct etimer *et)`-----Check if an event timer has expired.

`clock_time_t etimer_expiration_time (struct etimer *et)`-----Get the expiration time for the event timer.

`clock_time_t etimer_start_time (struct etimer *et)`-----Get the start time for the event timer.

#### 参考资料:

- [1] [Contiki 2.5: core/sys/process.c File Reference](#)
- [2] [Contiki 2.5: core/sys/process.h File Reference](#)
- [3] 博文《[Contiki学习笔记：编程模式](#)》
- [4] [Contiki 2.5: core/sys/etimer.h File Reference](#)
- [5] [Contiki 2.5: core/sys/etimer.c File Reference](#)

[Contiki学习笔记：Coffee文件系统概述及其学习资料](#) (2011-11-17 17:34)

标签: [CFS](#) [Coffee](#) [Contiki](#) [学习笔记](#) [学习资料](#) 分类: [Contiki](#)

## 摘要:

本文循序渐进讲解 Coffee 文件系统的缘由,即为什么需要文件系统、Flash 文件系统特性、WSN 文件系统的特性,最后列出一些学习 Coffee 文件系统可参考的资料。

## 一、概述

### 1.1 为什么需要文件系统

#### 1.1.1 便于数据管理和组织

无线传感器网络主要用于采集数据,并且数据通过网络传递给观察者。基于以下两个需求,需要将采集到的数据进行本地存储。

- (1) 数据传递时中断(受硬件和环境因素影响,发生概率还是很高的),需要重传
- (2) 合并多次采集到的数据,集中发出,以减少通信,进而降低功耗

#### 1.1.2 重编程的需要

考虑这样的情况,节点需要升级,一种思路就是将需要升级的节点全部回收回来,升级过后,再布置回去。这种思路缺点很明显也很致命,费时又费力,极端的情况,布置的结点就无法再取回了或者节点的工作不能被中断。

节点重编程恰解决了这个问题(也可以理解成动态加载),即通过无线传感器网络 WSN 将需要系统映像或者配置文件甚至是一个新的服务,传递到节点,进而更新。

## 1.2 Flash 文件系统特性

嵌入式系统(当然也包括传感器节点)的外存是Flash,相对于磁盘,Flash有其自身的特性,这也决定了Flash文件系统需要考虑额外的问题,比如耗损平衡(Flash擦除块数限制)、坏块管理、掉电保护、垃圾回收、映射机制等,详情见博文[《FLASH文件系统设计需考虑FLASH固有特性》](#)。

## 1.3 WSN 文件系统特性

无线传感器网络文件系统除了要考虑上述 Flash 文件系统的特性外(WSN 节点的外存通常也是 Flash),还需结合 WSN 的特点进行设计。

- (1) 内存受限,这就要求不能设计很大的数据结构驻留内存,也不能划出很大一块来做 cache。
- (2) 被采集数据特点,被采集数据量的大小、频率等都会影响整个文件系统的设计。

## 1.4 Coffee

现 有的基于 Flash 文件系统主要应用于嵌入式系统(内存往往没有那么受限、外存容量也比较大),不太适合无线传感器网络节点。我想,Contiki 大概也 是考虑到这些,所以自己写了个文件系统 Coffee File System。其最大的两个特点:减少内存使用、大规模存储。

## 二、学习资料

- [1] [Enabling Large-Scale Storage in Sensor Networks with the Coffee File System.pdf](#)
- [2] [File Systems - ContikiWiki](#)
- [3] [Coffee Filesystem Guide - ContikiWiki](#)
- [4] [Contiki 2.5: The Contiki file system interface](#)
- [5] 源代码(主要集中在 `contiki/core/cfs`、`contiki/cpu/platform` 文件夹下)

文档[1]是实现 Coffee 的人发表的论文，[4]是 Coffee 文件系统的 API 描述，而[2]则是 API 详细描述，还介绍部分原理，[3]集中讲 CFS 测试的例子。资料[1]~[4] (我目前所能找到的)远不够了解 Coffee，只能阅读源代码[5]。

接下来，就深入源码，结合文档，一探究竟……(欢迎一起讨论交流学习，Jelline@126.com)

[Contiki学习笔记: Coffee遇到若干问题\(解决及待解决\)](#) (2011-11-22 14:29)

标签: [Coffee](#) [Contiki](#) [创建文件](#) [学习笔记](#) [问题](#) 分类: [Contiki](#)

#### 摘要:

本文给出若干在分析 Coffee 文件系统遇到的问题(解决及待解决)，如找不到创建新文件的方法，语法方面的问题。

## 一、创建新文件方法

最近在分析 Coffee 文件系统, **没找到创建一个新文件的方法**, 很是纳闷。有函数 `cfs_open`、`cfs_close`、`cfs_read`、`cfs_write`、`cfs_seek`、`cfs_remove`, 但没有创建文件相关的函数。通常的设计, 会在打开文件进行判断, 若文件不存在会创建新文件, 但 coffee 文件系统的 `cfs_open(const char *name, int flags)` 只有 3 个标志: `CFS_READ`、`CFS_WRITE`、`CFS_APPEND`, 也没有类似 Linux 中打开文件 `O_CREAT` 的标志。于是深入源码, `cfs_open` 函数(文件路径 `contiki/core/cfs/cfs_coffee.c`)在找不到 `name` 对应文件的情况下, 就退出了。文件都创建不了, 何谈读出写入。

### 解决:

感谢网友回复, 提示我 `cfs_open` 包含着创建文件。我已深入分析 `cfs_open` 源码, 并整理成文档:

《[Contiki学习笔记: Coffee文件系统打开文件cfs\\_open](#)》

《[Contiki学习笔记: Coffee文件系统创建文件](#)》

### 新问题:

但调试过程中, 发现新的问题。创建新文件主体函数 `reserve` 调用 `find_contiguous_pages` 函数, 寻找满足的页, `find_contiguous_pages` 调用 `HDR_FREE(hdr)` 宏判断物理页是否可用(若物理页文件头 `file_header` 中 `flags` 的 A 位为 0, 则返回真), 问题就在这里, Coffee 格式化(擦除)时整个文件头都是 `0xFF`, 也就是说格式化后的 `flags` 每一位都是 1, 显然 `HDR_FREE(hdr)` 返回 1, 没法创建文件。

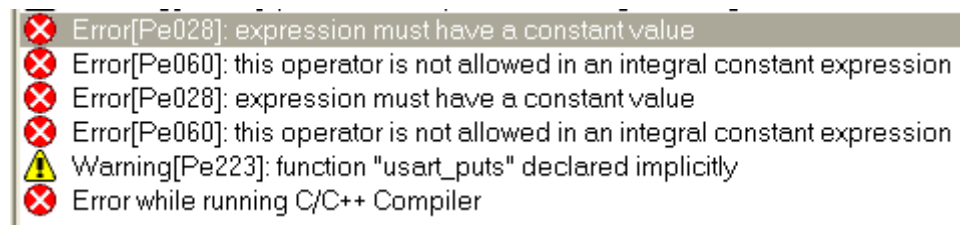
我尝试着把 `HDR_FREE(hdr)` 改成 `!HDR_FREE(hdr)`, 返回正确 `fd`。但读写时又有新的问题, 写入数据与读出数据不同(我怀疑压根就没写入)。所以呢, 接下来把 `cfs_read` 和 `cfs_write` 也分析清楚, 必要时将 FLASH 读写擦函数也分析下。

## 二、语法问题

在 `contiki\core\cfs\cfs_coffee.c` 文件下的 `merge_log` 函数，有这么一条语句，以致编译出错，如下：

```
1. char buf[hdr.log_record_size == 0 ? COFFEE_PAGE_SIZE : hdr.log_record_size];
```

错误提示如下(开发环境是 IAR)：



C 语言语法规则，声明数组应该是：数组名[常量表达式]，显然上面不是常量表达式。奇怪的是，我在 InstantContiki 2.5 环境下，居然可以编译通过(随便找一个硬件平台编译)。如此看来，这个问题应该是跟编译器相关了。

像这样的问题，在 Contiki 源码中，多处出现，罗列如下：

1. `uint16_t indices[batch_size];` //在文件 `cfs-coffee.c` 的 `get_record_index` 函数
- 2.
3. `uint16_t indices[preferred_batch_size];` //在文件 `cfs-coffee.c` 的 `find_next_record` 函数
- 4.
5. `char copy_buf[log_record_size];`

### 初步解决:

我的解决方法是，索性将这条语句简化为：`char buf[COFFEE_PAGE_SIZE]`；编译就通过了，但可能会带来副作用。

### 三、CFS\_APPEND 并不包含 CFS\_WRITE

Coffee 官方论文[1]声称指定了 CFS\_APPEND 意味着指向 CFS\_WRITE，从源码分析，显然必须得指定 CFS\_WRITE 才能写入。Coffee 写入文件 `cfs_write`，首先得判断 `fd` 的有效性和写权限检查，如下：

```
1. if(!(FD_VALID(fd) && FD_WRITABLE(fd)))
2. {
3.     return - 1;
4. }
5.
6. #define FD_WRITABLE(fd) (coffee_fd_set[(fd)].flags & CFS_WRITE)
```

FD\_WRITABLE 判断该文件是否以 CFS\_WRITE 打开，可见如果想从文件末尾写，则需同时指定 CFS\_WRITE 和 CFS\_APPEND。

### 解决:

(1) 记住这个用法，当需要从文件末尾写时，同时指定 CFS\_WRITE 和 CFS\_APPEND

(2) 修改 FD\_WRITABLE 宏，使 CFS\_APPEND 隐含着 CFS\_WRITE，修改后如下：

```
1. #define FD_WRITABLE(fd) (coffee_fd_set[(fd)].flags & CFS_WRITE & CFS_APPEND)
```

其他没发现的问题……

请不吝给予指教！可以发邮件[Jelline@126.com](mailto:Jelline@126.com)，也可以直接在本文末尾回复(匿名也可回复)，不胜感激。

### 参考资料:

[1] Tsiftes Nicolas, Dunkels Adam, He Zhitao. Enabling large-scale storage in sensor networks with the coffee file system[J]. International Conference on Information Processing in Sensor Networks. 2009, 349-360

[Contiki学习笔记: Coffee文件系统移植](#) (2011-12-05 15:54)

标签: [Coffee](#) [Contiki](#) [学习笔记](#) [文件系统](#) [移植](#) 分类: [Contiki](#)

### 摘要:

本文讲述 Contiki 文件系统 Coffee 的移植，包括参数配置和宏映射，并解释了 Coffee 一些宏的含义。

## 一、概述



Coffee 文件系统实现代码在 `contiki/core/cfs/cfs-coffee.c`, 与平台无关。移植 Coffee 文件系统主要是修改 `contiki/cpu/platform` (比如我的是 `arm/stm32f103`) / `cfs-coffee-arch.h` 文件, 包括针对特定硬件进行一些参数配置和将宏 `COFFEE_WRITE`、`COFFEE_READ`、`COFFEE_ERASE` 映射到对应的函数。

## 二、参数配置

### 2.1 硬件相关

我板子的MCU是STM32F103RBT6, 片上Flash是 128KB (0x0800 0000 ~ 0x0801 FFFF), 页大小是 1KB。如果不知道怎么看, 可参见博文《[STM32 片上Flash内存映射、页面大小、寄存器映射](#)》。

1. `#define FLASH_START 0x8000000 /*Flash 起始地址*/`
2. `#define FLASH_PAGE_SIZE 1024 /*最小擦除单元*/`
3. `#define FLASH_PAGES 125 /*Last 3 pages reserved for NVM*/`

最后 3 页留给 NVM, NVM 是指非易失性存储器 (non-volatile memory) 还是指别管 (Never Mind)? 个人认为, 应该预留不使用, 因为在链接映像文件 `contiki.map` (Contiki\EWARM\Debug\List\ ) 找不到该范围的地址。在 IAR 链接脚本 `iar-cfg-coffee.icf` (Contiki\cpu\stm32w108), 给了内存区域 (Memory Regions) 的定义, 如下:

1. `/*-Memory Regions-*/`
2. `define symbol __ICFEDIT_region_NVM_start__ = 0x0801F800;`
3. `define symbol __ICFEDIT_region_NVM_end__ = 0x0801FFFF;`

### 2.2 COFFEE\_PAGE\_SIZE 和 COFFEE\_SECTOR\_SIZE

尽管 FLASH 按页写, 按块擦除, 但文件系统不一定非得按这个来 (当然, 落实到实际写还是按页写)。Coffee 逻辑页大小和逻辑区大小可以自定义, `COFFEE_SECTOR_SIZE` 用于应付大的存储设备 (比如 SD 卡), 在这种情况下, 将其设置大一点可加快顺序扫描速

度。[1]建议将 COFFEE\_PAGE\_SIZE 和 COFFEE\_SECTOR\_SIZE 设小一点, 可以获得更合理的性能(reasonable performance)。我的设置如下:

1. `/* Minimum reservation unit for Coffee. It can be changed by the user. */`
2. `#define COFFEE_PAGE_SIZE (FLASH_PAGE_SIZE/4) //即 256B`
- 3.
4. `/* These must agree with the parameters passed to makefsdata */`
5. `#define COFFEE_SECTOR_SIZE FLASH_PAGE_SIZE //即 1KB`

### 2.3 COFFEE\_ADDRESS

COFFEE\_ADDRESS, 指文件系统从 FLASH 地址 COFFEE\_ADDRESS 处开始管理, 尽管 FLASH 大小是 128KB (以 STM32F103RBT6 为例), 但需要腾出一部分空间来存放代码。从链接映射文件 (Generate linker map file, 更具体信息得参考《IAR C/C++ Development Guide for ARM》), 我编译链接完代码段占用空间为 0x0800 0000 ~ 0x08004210。源码如下:

1. `/* If using IAR, COFFEE_ADDRESS reflects the static value in the linker script iar-cfg-coffee.icf, so it can't be passed as a parameter for Make.*/`
2. `#ifdef __ICCARM__`
3. `#define COFFEE_ADDRESS 0x8010c00`
4. `#endif`

在链接脚本 iar-cfg-coffee.icf 中, 有这么一项, 如下:

1. `/*-Memory Regions-*/`
2. `define symbol __ICFEDIT_region_CFS_start__ = 0x08010c00; /* Reserved for contiki flash file system. COFFEE_ADDRESS must be changed also in cfs-coffee-arch.h */`
- 3.

4. `define symbol __ICFEDIT_region_CFS_end__ = 0x0801F3FF;`

## 2.4 若干宏含义

1. `#define COFFEE_PAGES ((FLASH_PAGES*FLASH_PAGE_SIZE-(COFFEE_ADDRESS-FLASH_START))/COFFEE_PAGE_SIZE)`
2. `#define COFFEE_START (COFFEE_ADDRESS & ~(COFFEE_PAGE_SIZE-1))`
3. `#define COFFEE_SIZE (COFFEE_PAGES*COFFEE_PAGE_SIZE)`

**COFFEE\_PAGES (Coffee 页面总数):**  $(125*1024)-(0x8010c00-0x80000000)$ ，再除以 256，最后结果是 232

**COFFEE\_START (Coffee 起始地址):** 即屏蔽掉 COFFEE\_ADDRESS 后 8 位 (COFFEE\_PAGE\_SIZE 为 256B)，结果还是 0x08010c00

**COFFEE\_SIZE (Coffee 总大小):** COFFEE\_PAGES\*COFFEE\_PAGE\_SIZE，等于 59392Byte，即 58KB

以下是在 Linux 运行 (单独把 cfs-coffee-arch.h 抽出来，略在修改)，打印出来的结果：

```
jelline@sbserver-desktop:~$ ./tmp
COFFEE_PAGE_SIZE:      256
COFFEE_PAGES:          232
COFFEE_SIZE:           59392
COFFEE_START:          0x8010c00
COFFEE_ADDRESS:        0x8010c00
end_flash:             0x8020000
```

## 2.5 其他选项

以下这些选项根据实际应用设置：

```
1. #define COFFEE_NAME_LENGTH 20
2.
3. #define COFFEE_MAX_OPEN_FILES 4
4. #define COFFEE_FD_SET_SIZE 8
5. #define COFFEE_DYN_SIZE (COFFEE_PAGE_SIZE*1)
6. #define COFFEE_MICRO_LOGS 0
7. #define COFFEE_LOG_TABLE_LIMIT 16 //It doesn't matter as COFFEE_MICRO_LOGS is 0.
8. #define COFFEE_LOG_SIZE 128
```

### 三、映射宏

将 Coffee 文件系统读、写、擦除映射到平台相关的函数，如下所示，并在 cfs-coffee-arch.c 实现。

```
1. #define COFFEE_WRITE(buf, size, offset) stm32_flash_write(COFFEE_START + offset, buf, size)
2.
3. #define COFFEE_READ(buf, size, offset) stm32_flash_read(COFFEE_START + offset, buf, size)
4.
5. #define COFFEE_ERASE(sector) stm32_flash_erase(sector)
```

注：

本文只是根据[1]和源代码分析，旨在对 Coffee 若干宏有个了解，方便日后分析。具体移植，可能还需更多工作。

**参考资料:**

[1] [File Systems - ContikiWiki](#)

[Contiki学习笔记: Coffee文件组织及若干数据结构](#) (2011-12-07 11:04)

标签: [Coffee](#) [Contiki](#) [file\\_header](#) [学习笔记](#) [文件组织](#) 分类: [Contiki](#)

**摘要:**

本文介绍了 Coffee 文件系统的物理上和内存上的组织, 还详细介绍了若干关键数据结构, 如文件头 flie\_header、protected\_mem\_t、文件描述符 file\_desc、文件 file。

## 一、物理组织

### 1.1 概述

Coffee 一个文件在 FLASH 的组织示意图如下[1], 原始文件包括文件头、数据区, 还有可能部分空闲区(文件没那么大, 一页用不完)。当修改(modified)一个文件时, 不是在原文件修改(不现实, 因为 FLASH 先擦后写), 而是创建一个微日志结构的文件(micro log file), 并链接到原始文件[1]。原文是 modified, 但我觉得应该是追加的意思。如果对原始文件进行修改, 应该是先把文件内容加载到 RAM, 修改完再写到新的页上, 并标记原始文件无效, 这只是猜测, 还需结合源代码加以验证。

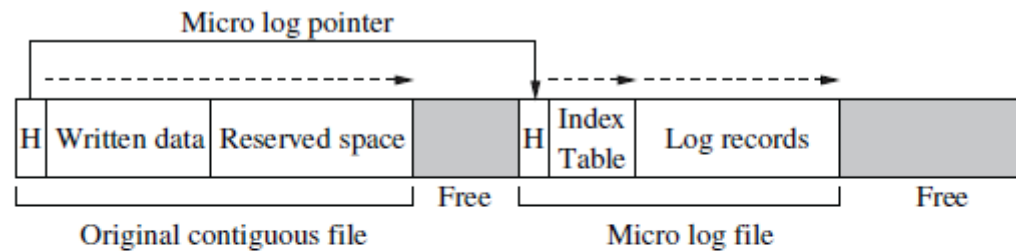


Figure 1: The file layout in the flash memory. Files consist of reserved consecutive pages and start with a header (H). Modified files are linked with a micro log file containing the most recent data.

索引表(Index table)如下图[1],Coffee 文件系统采用 Extent 索引机制,很多文件系统采用这种机制是为了减少碎片(reduce fragmentation),但 Coffee 则是为了降低文件系统的复杂性[1]。同时,采用 Extent 机制让文件结构体更简单,从而减少元数据的缓存。

索引机制是文件系统的核心技术[3],extent 和 blockmap 是两类典型的索引实现方案,前者基于片断索引(如 NTFS、Vxfs、JFS、Ext4),后者基于分配块的位图索引(如 UFS、SCO HTFS、Ext2/3)[3]。具体说来,Extent 索引是按分配的片断记录,只记录起始块、连续块数、文件内部块位置[3],如果一个文件由多组不连续的块组成,则需要多条记录。blockmap 索引机制,文件的每个分配块都有一个索引与之一一对应。blockmap 可以快速定位到文件特定的块,但需要很大的索引空间(极端情况,缓冲机制无法一次读入,效率便会下降),也浪费磁盘空间,所以现在 blockmap 机制已经很少使用了。而 Extent 需要索引空间小,连续读写会有优势,但算法复杂度略高[3]。

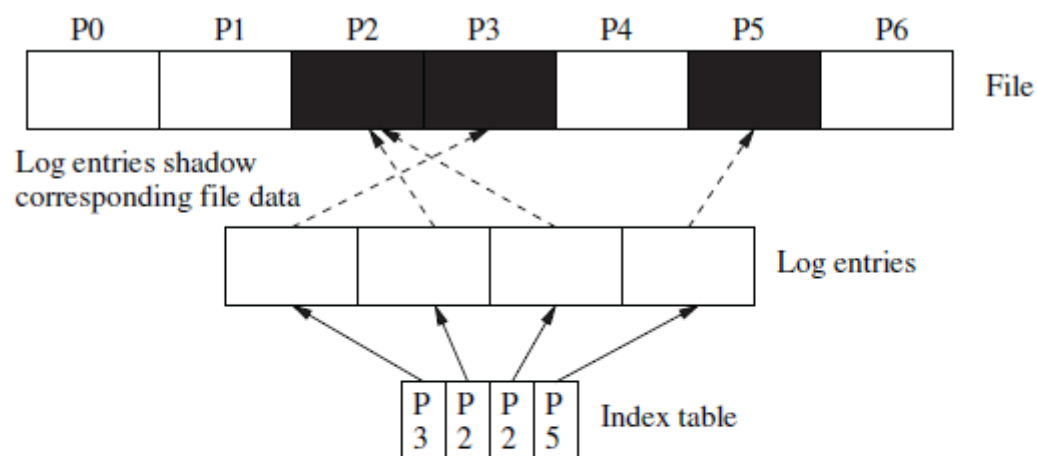


Figure 3: A modified Coffee file consists of data stored both in the original file and in a micro log. Writes that do not append data to the original file are added as records in the micro log files.

## 1.2 file\_header

file\_header 用于描述一个文件的元数据，即描述了一个文件的基本信息(比如占用页情况)，file\_header 位于文件(log file 或 micro log file)第一页的开始处。源代码如下：

1. /\*The file header structure mimics the representation of file headers in the physical storage medium\*/
2. struct file\_header
3. {

```
4.    coffee_page_t log_page;
5.    uint16_t log_records;
6.    uint16_t log_record_size;
7.    coffee_page_t max_pages;
8.    uint8_t deprecated_eof_hint;
9.    uint8_t flags;
10.   char name[COFFEE_NAME_LENGTH];
11.};
```

file\_header 结构体可以直观表示成下图[1]:



```

      0                               1
    0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Log file start page           |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Log record amount             |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Log record size               |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|           Maximum pages                 |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| EOF hint  | A | L | O | M | I | | | V |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+

```

Figure 2: The file metadata contains micro log information and file status indicators.

各成员变量含义如下:

`log_page`

如果配置了微日志(见 `cfs-coffee-arch.h` 文件配置 `#define COFFEE_MICRO_LOGS 0`), 那么 `log_page` 指向微日志的第一页, 见 Figure 1。

`log_records`

表示日志可以容纳的记录数量(log records denotes the number of records that the log can hold)

### **log\_record\_size**

log\_record\_size 表示微日志文件大小, 如果为 0, 则设置成默认值(见 cfs-coffee-arch.h 文件配置#define COFFEE\_LOG\_SIZE 128)。

### **max\_pages**

max\_pages 指为文件保留着页面数(The max pages field specifies the amount of pages that have been reserved for the file)

### **deprecated\_eof\_hint**

因为文件头不能存储文件长度(缘于文件长度经常变化着), 所以用 deprecated\_eof\_hint 指向文件的最后一个字节。文件关闭时, 如果文件长度增加则需更新 deprecated\_eof\_hint(先擦后写, 如何更新?)。

### **flags**

flags 反映了文件当前状态(The flag field tells us the current state), 页有 3 种状态空闲(free, 即可以用来写)、有效(active, 即数据有效)、无效(obsolete, 即数据无效且还没擦除, 不能用 来写)。如 Figure2, flags 用了 6 个位, 分别是 ALOMIV, 接下来详细解释之:

#### **A(allocated)**

如果该位被设置, 表示文件正在使用。反之, 当前页及所有保留页(直到下一个逻辑区的边界)是空闲的。

### **O(obsolete)**

当文件被删除时，O 标记保留页是无效的 (obsolete) [2]。不是吧，这些页不用擦除就可以直接使用的，怎么说也应该标记成空闲 free？

### **M(modified)**

资料[1][2]居然没提这个标志，得根据源代码推测了：-（在 cfs-coffee.c 的 flags 宏定义可得知，M 表示文件已被修改，日志存在 (Modified file, log exists)。

### **L(log)**

L(log) 标记文件已被修改，与微日志文件存在有关 (and that a related log file exists)。不同于 M，应该是指微日志的修改标志 (待读源码确认)。

### **I(isolated)**

I 标记孤立的页，所有 Coffee 算法每次都会处理孤立的页面 (Isolated pages are processed one at a time by all Coffee algorithms, and are treated the same way as obsolete files)

### **V(valid)**

V(valid) 标记文件头是完整的 (helps by marking that the header data is complete), To discover garbled headers—typically caused by a system reboot during a header write operation.

为方便操作，Coffee 将这些 flags 单独定义成宏，源代码如下 (在 cfs-coffee.c 文件)：

```
1. /* File header flags. */
2. #define HDR_FLAG_VALID          0x1      /* Completely written header. */
3. #define HDR_FLAG_ALLOCATED      0x2      /* Allocated file. */
4. #define HDR_FLAG_OBSOLETE       0x4      /* File marked for GC. */
5. #define HDR_FLAG_MODIFIED       0x8      /* Modified file, log exists. */
6. #define HDR_FLAG_LOG            0x10     /* Log file. */
7. #define HDR_FLAG_ISOLATED       0x20     /* Isolated page. */
```

注：这6个标志是有优先顺序的，依次是 the valid flag (V)、the isolated flag (I)、the obsolete flag (O)、the log flag (L)、the allocated flag (A)。(不晓得 M 该处于哪个位置)

**name**

文件名，其中文件长度 COFFEE\_NAME\_LENGTH 可配置，在 contiki/cpu/平台 (如 arm/stm32f103)/cfs-coffee-arch.h

## 二、内存组织

为了提高性能，Coffee 缓存了文件元数据，并用文件描述符 file\_desc 与之映射，Coffee 文件系统在内存组织逻辑图如下：

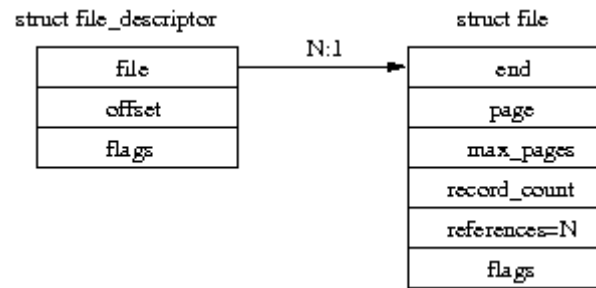


图 Coffee 文件系统在内存组织逻辑图[2]

Coffee 将文件描述符 `file_desc` 组织成一个数组，作为 `protected_mem_t` 的一个成员变量，数组个数(即系统缓存 `file_desc` 最大数目)，可以在 `cfs-coffee-arch.h` 进行配置，默认情况是 8 条，如下：

1. `#define COFFEE_FD_SET_SIZE 8`

`protected_mem_t` 结构体源代码如下：

```

1. /*The protected memory consists of structures that should not be overwritten during system checkpointing because
   they may be used by the checkpointing implementation. These structures need not be protected if checkpointing is
   not used.*/
2. static struct protected_mem_t
3. {
4.     struct file coffee_files[COFFEE_MAX_OPEN_FILES];
5.     struct file_desc coffee_fd_set[COFFEE_FD_SET_SIZE];
6.     coffee_page_t next_free;
7.     char gc_wait;
8. } protected_mem;
  
```

9.

```
10. static struct file *const coffee_files = protected_mem.coffee_files;  
11. static struct file_desc *const coffee_fd_set = protected_mem.coffee_fd_set;  
12. static coffee_page_t *const next_free = &protected_mem.next_free;  
13. static char *const gc_wait = &protected_mem.gc_wait;
```

## 2.1 文件描述符结构体 file\_desc

系统缓存 file\_desc 示意图如下：

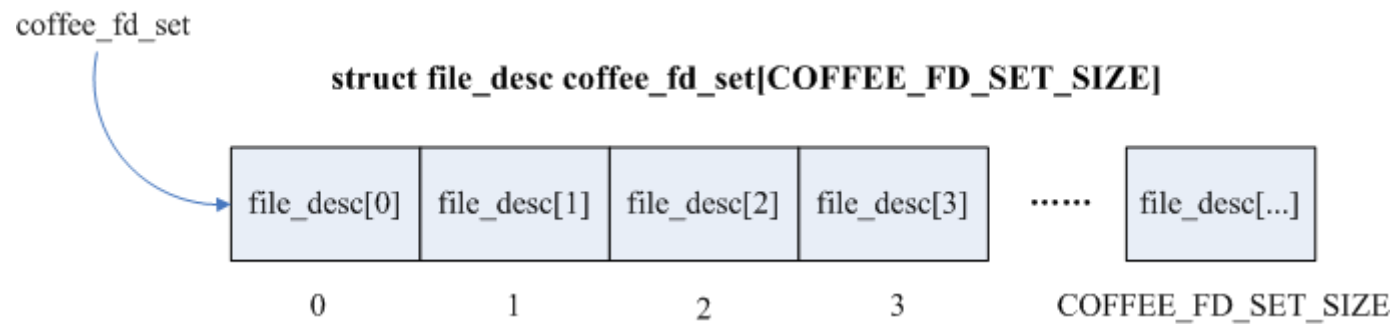


图 Coffee 缓存 file\_desc 示意图

这跟 Linux 文件系统类似，offset 存储文件偏移量，file 指针指向 2.2 的 file，源码如下：

```
1. /* The file descriptor structure. */  
2. struct file_desc  
3. {  
4.     cfs_offset_t offset;
```

```
5.     struct file *file;
6.     uint8_t flags;
7.     #if COFFEE_IO_SEMANTICS
8.         uint8_t io_flags;
9.     #endif
10.};
```

Coffee 文件系统出现多次 flags，有些含义相同，有些则不同。file\_desc 的 flags 与 1.2 的 flags 含义不同，共 4 个取值，如下，用于标记文件的权限，即读、写、追加、空闲。

```
1. #define COFFEE_FD_FREE           0x0
2. #define COFFEE_FD_READ           0x1
3. #define COFFEE_FD_WRITE          0x2
4. #define COFFEE_FD_APPEND         0x4
```

对于某些存储器，为了优化文件访问(optimize file access on certain storage types)，可以配置 COFFEE\_IO\_SEMANTICS。Coffee 定义了 io\_flags 的两种值，

```
1. #define CFS_COFFEE_IO_FLASH_AWARE 0x1
2. #define CFS_COFFEE_IO_FIRM_SIZE   0x2
```

当 写超过预留的大小时，Coffee 文件系统不会再扩展文件。当文件有固定大小的限制，CFS\_COFFEE\_IO\_FIRM\_SIZE 必须设置，保护不被 写超了(protect against writes beyond this limit)。CFS\_COFFEE\_IO\_FLASH\_AWARE，还不晓得什么意思，先贴出源码注释，

## 2.2 file

系统缓存 file 示意图如下：

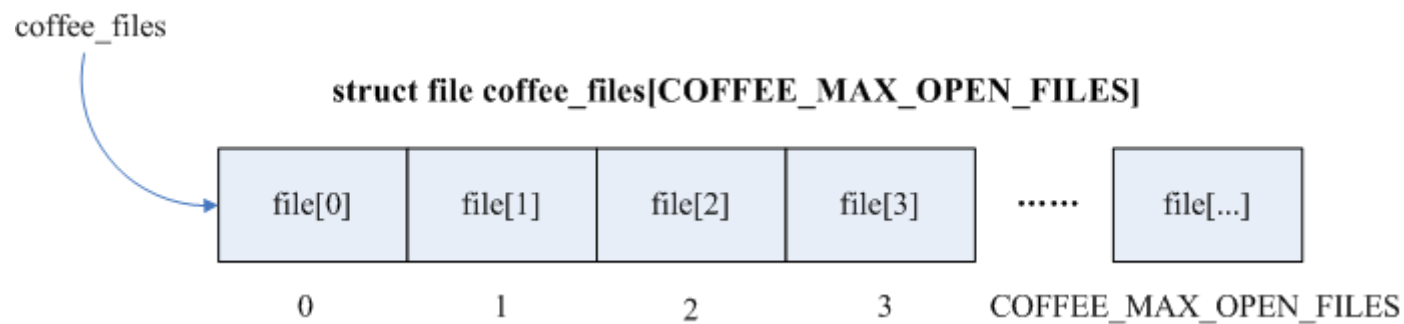


图 Coffee 缓存 file 示意图

file 源代码如下：

```
1. /* The structure of cached file objects. */
2. struct file
3. {
4.     cfs_offset_t end;
5.     coffee_page_t page;
6.     coffee_page_t max_pages;
7.     int16_t record_count;
8.     uint8_t references;
9.     uint8_t flags;
10.};
```



file 包容了元数据文件头 file\_header 一些信息(可以理解成 file 缓存了 file\_header)，值得一提的是 end 和 references。end 存放文件的最后一个字节的偏移量，当打开一个文件时，Coffee 用蛮力法找到文件末尾(即 Extent 末尾向前找每一个非空字节 [1]???)。references 记录文件的引用次数，Contiki 提供类多线程编程环境，会有这样的情况，多个线程同时打开一个文件，需要记录引用次数。

### 参考资料:

- [1] [Enabling Large-Scale Storage in Sensor Networks with the Coffee File System.pdf](#)
- [2] [File Systems - ContikiWiki](#)
- [3] 博文《[文件系统的两种文件索引模式extent和blockmap](#)》

[Contiki学习笔记: Coffee文件系统flags标志位](#) (2011-12-24 18:27)

标签: [Coffee](#) [Contiki](#) [file\\_header->flags](#) [flags](#) [学习笔记](#) 分类: [Contiki](#)

### 摘要:

Coffee 文件系统多处地方出现 flags，包括 file\_header->flags、file\_desc->flags、file\_desc->io\_flags、file->flags，含义不相同，本文整理这些 flags。

## 一、cfs\_open 的 flags

Coffee 打开文件 cfs\_open 函数需要指定 flags, flags 可以取 CFS\_READ、CFS\_WRITE、CFS\_APPEND 之一或者组合, 宏定义如下:

1. `//int cfs_open(const char *name, int flags)`
2. `#define CFS_READ 1`
3. `#define CFS_WRITE 2`
4. `#define CFS_APPEND 4`

CFS\_READ、CFS\_WRITE、CFS\_APPEND 正好对应于 COFFEE\_FD\_READ、COFFEE\_FD\_WRITE、COFFEE\_FD\_APPEND。纵观源代码, 后 3 个都没用到, 而是用前 3 个代替, 看以下 cfs\_open 函数语句就懂了:

1. `//int cfs_open(const char *name, int flags)`
2. `fdp->flags |= flags;`

## 二、file\_desc->flags

file\_desc->flags 存储文件访问权限, 系统定义了四个值: COFFEE\_FD\_FREE、COFFEE\_FD\_READ、FFEE\_FD\_WRITE、COFFEE\_FD\_APPEND, 源代码如下:

1. `#define COFFEE_FD_FREE 0x0`
2. `#define COFFEE_FD_READ 0x1`

3. `#define COFFEE_FD_WRITE 0x2`
4. `#define COFFEE_FD_APPEND 0x4`

事实上，后 3 个都没用到，而是用 CFS\_READ、CFS\_WRITE、CFS\_APPEND 代替。系统缓存 file\_desc 组织成一个数组 coffee\_fd\_set[COFFEE\_FD\_SET\_SIZE]，当 coffee\_fd\_set[i]->flags 为 COFFEE\_FD\_FREE，表示该 FD 可使用。

### 三、file->flags

file的flags只有两种取值：0 和COFFEE\_FILE\_MODIFIED，将物理文件缓存时(load\_file函数，详情见博文《[Contiki学习笔记：Coffee文件系统打开文件cfs\\_open](#)》2.5)，如果物理文件元数据file\_header的flags的M位为 1 的话(即物理文件被修改，日志存在)，则file->flags设为COFFEE\_FILE\_MODIFIED，否则设为 0。部分代码如下：

1. `//load_file 函数部分代码`
- 2.
3. `file->flags = 0;`
4. `if(HDR_MODIFIED(*hdr))`
5. `{`
6.  `file->flags |= COFFEE_FILE_MODIFIED; //如果文件被修改(表示日志存在)，#define COFFEE_FILE_MODIFIED 0x1`
7. `}`

再看下面的代码就很清楚了，如果 file->flags 为 COFFEE\_FILE\_MODIFIED，返回真。

1. `#define FILE_MODIFIED(file) (((file)->flags & COFFEE_FILE_MODIFIED)`
- 2.
3. `#define COFFEE_FILE_MODIFIED 0x1`

## 四、file\_desc->io\_flags

### 4.1 COFFEE\_IO\_SEMANTICS

如果定义了 COFFEE\_IO\_SEMANTICS，则在 file\_desc 结构体会多一个成员变量 io\_flags，源码如下：

```
1. struct file_desc
2. {
3.     cfs_offset_t offset;
4.     struct file *file;
5.     uint8_t flags;
6.     #if COFFEE_IO_SEMANTICS
7.         uint8_t io_flags;
8.     #endif
9. };
```

配置了 COFFEE\_IO\_SEMANTICS 可以优化某些存储设备的文件访问(optimize file access on certain storage types)，系统默认没有配置 COFFEE\_IO\_SEMANTICS，源码如下：

### 4.2 file\_desc->io\_flags

系统定义了 io\_flags 两个值，即 CFS\_COFFEE\_IO\_FLASH\_AWARE 和 CFS\_COFFEE\_IO\_FIRM\_SIZE。

#### (1)CFS\_COFFEE\_IO\_FLASH\_AWARE

1. `/*Instruct Coffee that the access pattern to this file is adapted to flash I/O semantics by design, and Coffee should therefore not invoke its own micro logs when file modifications occur.This semantical I/O setting is useful when implementing flash storage algorithms on top of Coffee. */`
- 2.
3. `#define CFS_COFFEE_IO_FLASH_AWARE 0x1`

CFS\_COFFEE\_IO\_FLASH\_AWARE 没看懂，需要阅读更多代码，后续补充。

## (2)CFS\_COFFEE\_IO\_FIRM\_SIZE

1. `/* A case when this is necessary is when the file has a firm size limit,and a safeguard is needed to protect against writes beyond this limit. */`
- 2.
3. `#define CFS_COFFEE_IO_FIRM_SIZE 0x2`

如果写入文件超过预留的大小，Coffee 不会继续扩展文件。当文件有固定大小限制时，设置 CFS\_COFFEE\_IO\_FIRM\_SIZE 可以保护写超过。

file\_desc 的 io\_flags 可以通过 cfs\_coffee\_set\_io\_semantics() 函数来设置，源码如下，其中法 flag 是 CFS\_COFFEE\_IO\_FLASH\_AWARE 和 CFS\_COFFEE\_IO\_FIRM\_SIZE 两者之一或全部。源码如下：

1. `#if COFFEE_IO_SEMANTICS`
2. `int cfs_coffee_set_io_semantics(int fd, unsigned flags)`
3. `{`
4. `if(!FD_VALID(fd))`
5. `{`
6. `return - 1;`

```
7.      }  
8.  
9.      coffee_fd_set[fd].io_flags |= flags;  
10.  
11.     return 0;  
12.    }  
13. #endif
```

## 五、file\_header->flags

file\_header 的 flags 反应了物理文件的状态，用了 6 位，分别是 ALOMIV。但官方论文[1]与实际源码各位的位置不同，不过不影响分析，忠于源码就是了。file\_header 的 flags 两种版本示意图如下：

图 1 file\_header 的 flags 两种版本

A(allocated)

如果该位被设置，表示文件正在使用。反之，当前页及所有保留页(直到下一个逻辑区的边界)是空闲的。

### **O(obsolete)**

当文件被删除时，O 标记保留页是无效的(obsolete) [2]。不是吧，这些页不用擦除就可以直接使用的，怎么说也应该标记成空闲 free?

### **M(modified)**

资料[1][2]居然没提这个标志，得根据源代码推测了：-( 在 cfs-coffee.c 的 flags 宏定义可得知，M 表示文件已被修改，日志存在(Modified file, log exists)。

### **L(log)**

L(log) 标记文件已被修改，与微日志文件存在有关(and that a related log file exists)。不同于 M，应该是指微日志的修改标志(待读源码确认)。

### **I(isolated)**

I 标记孤立的页，所有 Coffee 算法每次都会处理孤立的页面(Isolated pages are processed one at a time by all Coffee algorithms, and are treated the same way as obsolete files)

### **V(valid)**

V(valid) 标记文件头是完整的(helps by marking that the header data is complete), To discover garbled headers-typically caused by a system reboot during a header write operation.

注：这6个标志是有优先顺序的，依次是 the valid flag(V)、the isolated flag(I)、the obsolete flag(O)、the log flag(L)、the allocated flag(A)。(不晓得 M 该处于哪个位置)

为方便操作，Coffee 将这些 flags 单独定义成宏，源代码如下(在 cfs-coffee.c 文件)：

```
1. /* File header flags. */
2. #define HDR_FLAG_VALID      0x1 /* Completely written header. */
3. #define HDR_FLAG_ALLOCATED  0x2 /* Allocated file. */
4. #define HDR_FLAG_OBSOLETE   0x4 /* File marked for GC. */
5. #define HDR_FLAG_MODIFIED   0x8 /* Modified file, log exists. */
6. #define HDR_FLAG_LOG        0x10 /* Log file. */
7. #define HDR_FLAG_ISOLATED   0x20 /* Isolated page. */
```

Coffee 定义了若干宏来判断文件的状态(依据 file\_header 中的 flags)，如下：

```
1. /* File header macros. */
2. #define CHECK_FLAG(hdr, flag) ((hdr).flags & (flag))
3.
4. #define HDR_VALID(hdr) CHECK_FLAG(hdr, HDR_FLAG_VALID) //若 V 标志置 1，则返回真
5. #define HDR_ALLOCATED(hdr) CHECK_FLAG(hdr, HDR_FLAG_ALLOCATED) //若 A 标志置 1，则返回真
6. #define HDR_OBSOLETE(hdr) CHECK_FLAG(hdr, HDR_FLAG_OBSOLETE) //若 O 标志置 1，则返回真
7. #define HDR_MODIFIED(hdr) CHECK_FLAG(hdr, HDR_FLAG_MODIFIED) //若 M 标志置 1，则返回真
8. #define HDR_LOG(hdr) CHECK_FLAG(hdr, HDR_FLAG_LOG) //若 L 标志置 1，则返回真
9. #define HDR_ISOLATED(hdr) CHECK_FLAG(hdr, HDR_FLAG_ISOLATED) //若 I 标志置 1，则返回真
10.
11. #define HDR_FREE(hdr) !HDR_ALLOCATED(hdr) //若 A 标志置 0，则返回真
```



12. `#define HDR_ACTIVE(hdr) (HDR_ALLOCATED(hdr) && !HDR_OBSOLETE(hdr) && !HDR_ISOLATED(hdr))` //若A为1,0为0, I为0, 则返回真

有个疑问, 在 Coffee 格式化 `cfs_coffee_format` 中, 将所有的逻辑区全部擦除, 即 `file_header` 各成员变量填充为 1, 所以 `file_header` 的 `flags` 各位都是 1。再结合上述宏定义, 不免会发现矛盾之处。

#### 参考资料:

[1] Tsiftes Nicolas, Dunkels Adam, He Zhitao. Enabling large-scale storage in sensor networks with the coffee file system[J]. International Conference on Information Processing in Sensor Networks. 2009, 349-360

[2] Contiki 源代码

[Contiki学习笔记: Coffee文件系统file与file header及其联系](#) (2012-01-07 16:52)

标签: [Coffee](#) [Contiki](#) [file](#) [file header](#) [学习笔记](#) 分类: [Contiki](#)

#### 摘要:

本文重在讲述 file 和 file\_header 的联系与区别，首先分别分析了 file 与 file\_header 结构体，而后介绍两者的联系，最后分析几组易混的成员变量，包括 page 与 log\_page、file 与 file\_header 的 flags、deprecated\_eof\_hint 与 end、record\_count 与 log\_records。

## 一、file

结构体 file 源码如下：

```
1. struct file
2. {
3.     cfs_offset_t end;
4.     coffee_page_t page;
5.     coffee_page_t max_pages;
6.     int16_t record_count;
7.     uint8_t references;
8.     uint8_t flags;
9. };
```

end

指向存放文件的最后一个字节的偏移量，当打开一个文件时，Coffee用蛮力法找到文件末尾(file\_header是不存储文件末尾的位置，因为文件长度经常改变)。详情见博文 [《Contiki学习笔记：Coffee文件系统打开文件cfs\\_open》](#)。

page

指向物理文件的第一页，即存放文件元数据 `file_header` 的页。通常的用法如下：

1. //用法 1: `static struct file *find_file(const char *name)`
2. `read_header(&hdr, coffee_files[i].page);` //读取文件元数据 `file_header` 到 `hdr`
- 3.
4. //用法 2: `static int remove_by_page(coffee_page_t page, int remove_log, int close_fds, int gc_allowed)`
5. `coffee_files[i].page = INVALID_PAGE;` //将 `file->page` 标识为 `INVALID_PAGE`，表示该缓存 `file` 项可分配给其他文件用
6. `#define INVALID_PAGE ((coffee_page_t)-1)` //注 1

### **max\_pages**

`max_pages` 含义跟 `file_header->max_pages` 相同，即为该文件保留的页面数，加载文件 `load_file` 函数将 `file_headermax_pages` 赋给 `file` 的 `max_pages`。

### **record\_count**

`record_count` 表示实际的微日志记录数量，不同于 `file_header` 的 `log_records`。

### **references**

Contiki 提供类多线程编程环境，会有这样的情况，多个线程同时打开一个文件，需要记录引用次数。打开文件 `cfs_open` 引用次数++，关闭文件 `cfs_close` 引用次数--。

### **flags**

`flags` 两种取值：0 和 `COFFEE_FILE_MODIFIED`，即标识该文件是否含有微日志文件。详情见博文《[Contiki学习笔记: Coffee文件系统flags标志位](#)》。

**注 1:**

这里用了个小技巧，使得程度更具移植性。因为在 cfs-coffee.arch.h, coffee\_page\_t 有可能是 8 位，也有可能是 16 位，源码如下：

```
1. #if COFFEE_PAGES <= 127
2.     #define coffee_page_t u8_t
3. #elif COFFEE_PAGES <= 0x7FFF
4.     #define coffee_page_t u16_t
5. #endif
```

把 INVALID\_PAGE 定义为 coffee\_page\_t-1，如果 coffee\_page\_t 是 8 位，那么 INVALID\_PAGE 为 0xFF，如果 coffee\_page\_t 是 16 位，那么 INVALID\_PAGE 为 0xFFFF，可以自适应 coffee\_page\_t 类型的变化，更具移植性。关于 这个技巧可参见博文《[具有可移植的无穷大定义](#)》。

## 二、file\_header

结构体 file\_header 源码如下：

```
1. struct file_header
2. {
3.     coffee_page_t log_page;
4.     uint16_t log_records;
5.     uint16_t log_record_size;
6.     coffee_page_t max_pages;
```

```
7.     uint8_t deprecated_eof_hint;  
8.     uint8_t flags;  
9.     char name[COFFEE_NAME_LENGTH];  
10.};
```

### **log\_page**

不同于 `file->page`, `log_page` 指向微日志的第一页(如果配置了微日志)。

### **log\_records**

表示日志可以容纳的记录数量(log records denotes the number of records that the log can hold)

### **log\_record\_size**

`log_record_size` 表示微日志文件大小, 如果为 0, 则设置成默认值(见 `cfs-coffee-arch.h` 文件配置 `#define COFFEE_LOG_SIZE 128`)。

### **max\_pages**

`max_pages` 指为文件保留着页面数(The max pages field specifies the amount of pages that have been reserved for the file)

### **deprecated\_eof\_hint**

因 为文件头不能存储文件长度(缘于文件长度经常变化着), 所以用 deprecated\_eof\_hint 指向文件的最后一个字节。文件关闭时, 如果文件长度 增加则需更新 deprecated\_eof\_hint。事实上, 纵观源码, deprecated\_eof\_hint 都没用到, 而是用 file->end 存储文件末尾的位置(打开文件时, 用蛮力法扫描得到该文件的末尾位置)。

## flags

flags反映了文件当前状态(The flag field tells us the current state), 用了 6 个位, 分别是ALOMIV, 详情请见博文《[Contiki 学习笔记: Coffee文件系统flags标志位](#)》。

## name

文件名, 其中文件长度 COFFEE\_NAME\_LENGTH 可配置, 在 contiki/cpu/平台 (如 arm/stm32f103)/cfs-coffee-arch.h。

## 三、file 与 file\_header 联系

### 3.1 联系

file\_header 存放物理文件的元数据(即描述该物理文件), 而 file 可以理解成物理文件元数据的内存表示, 但又不是完全缓存, 因为 file 与 file\_header 成员变量差别甚大。为了提高性能, Contiki 具体是这样做的: 所有打开的文件都通过文件描述符 fd(非负整数, 类似于 Linux)引用, 将 file\_desc 组织成一个数组 file\_desc coffee\_fd\_set[COFFEE\_FD\_SET\_SIZE], 每个 fd 对应于一个 file\_desc(以数组下标的形式), 每个 file\_desc 对应于一个 file。通过数组 file\_desc coffee\_fd\_set[COFFEE\_FD\_SET\_SIZE]下标(即文件描述符 fd)就可以访问 file, 而 file 存储物理文件的一些信息, 这样也就可以访问物理文件了。

### 3.2 page 与 log\_page

file\_page 是指向物理文件的第一页，即存放文件元数据 file\_header 的页。而 file\_header->log\_pages 指向微日志的第一页(如果配置了微日志)。

### 3.3 file 与 file\_header 的 flags

file\_header->flags记录的是整个物理文件的相关信息(即元数据)，而file->flags只用来标识文件是否有微日志存在。详情请见博文《[Contiki学习笔记: Coffee文件系统flags标志位](#)》。

### 3.4 deprecated\_eof\_hint 与 end

file\_header->deprecated\_eof\_hint 与 file->end 都存储文件的末尾的位置，但事实上，纵观源码，file\_header->deprecated\_eof\_hint 是没有用到的(这点与官方论文有出入)，也就是说物理上没有存储文件的末尾位置，而是每次打开文件时，通过蛮力扫描，得到文件末尾位置，存入 file->end。

### 3.5 record\_count 与 log\_records

这两个关系，我也不是搞得很清楚。给我感觉就是，file\_header->log\_records 是指文件创建时的指定的微日志记录数量，默认的情况会是 COFFEE\_LOG\_SIZE/log\_record\_size(log\_record\_size 默认情况会是 COFFEE\_PAGE\_SIZE，见 adjust\_log\_config 函数)。而 file->record\_count 是实际上的微日志记录数量。

[Contiki学习笔记: Coffee文件系统格式化cfs coffee format](#) (2011-12-07 16:44)

标签: [Coffee](#) [Contiki](#) [format](#) [学习笔记](#) [格式化](#) 分类: [Contiki](#)

摘要:

本文深入源码讲述了 Contiki 文件系统 Coffee 的格式化，即擦除整个 FLASH，并初始化结构体 protected\_mem。

## 一、格式化

### 1.1 硬盘格式化

#### (1) 低级格式化

低级格式化就是将空白的磁盘划分出柱面和磁道，再将磁道划分为若干个扇区，每个扇区又划分出标识部分 ID、间隔区 GAP 和数据区 DATA 等。可见，低级格式化是高级格式化之前的一件工作，低级格式化只能针对一块硬盘而不能支持单独的某一个分区。每块硬盘在出厂时，已由硬盘生产商进行低级格式化，因此通常使用者无需再进行低级格式化操作。其实，我们对一张软盘进行的全面格式化就是一种低级格式化。对于硬盘上出现逻辑坏道或者软性物理坏道，用户可以试试使用低级格式化来达到屏蔽坏道的作用，这样能在一定程度上保证用户数据的可靠性，但坏道却会随着硬盘分区、格式化次数的增长而扩散蔓延[3]。

#### (2) 高级格式化

高级格式化又称逻辑格式化，它是指根据用户选定的文件系统（如 FAT12、FAT16、FAT32、NTFS、EXT2、EXT3 等），在磁盘的特定区域写入特定数据，以达到初始化磁盘或磁盘分区、清除原磁盘或磁盘分区中所有文件的一个操作。高级格式化包括对主引导记录中分区表相应区域的重写、根据用户选定的文件系统，在分区中划出一片用于存放文件分配表、目录表等用于文件管理的磁盘空间，以使用户使用该分区管理文件[4]。

### 1.2 Coffee 格式化



跟 硬盘格式化类似, FLASH 低级格式化(可以如此理解, 也许根本就没有, 由产家弄好了)已经把 FLASH 按块-页分好了, Coffee 格式化类似于硬盘的 高级格式化, 但不尽相同。第一使用 Coffee 文件系统, 必须进行格式化, 将整个 FLASH 擦除(FLASH 物理特性, 先擦除后写)。

## 二、cfs\_coffee\_format

cfs\_coffee\_format 主要工作将整个 FLASH 擦除, 以及初始化结构体 protected\_mem, 源码如下:

```
1. int cfs_coffee_format(void)
2. {
3.     unsigned i;
4.     PRINTF("Coffee: Formatting %u sectors", COFFEE_SECTOR_COUNT);
5.
6.     *next_free = 0;
7.     for (i = 0; i < COFFEE_SECTOR_COUNT; i++)
8.     {
9.         COFFEE_ERASE(i);
10.        PRINTF(". ");
11.    }
12.
13.    memset(&protected_mem, 0, sizeof(protected_mem)); /* Formatting invalidates the file information.*/
14.
15.    PRINTF(" done!\n");
16.    return 0;
17.}
```

## 2.1 next\_free

next\_free 是 protected\_mem\_t 结构体的一个成员变量，指向下一个空闲的页，初始化为 0，源码如下：

```
1. static struct protected_mem_t
2. {
3.     struct file coffee_files[COFFEE_MAX_OPEN_FILES];
4.     struct file_desc coffee_fd_set[COFFEE_FD_SET_SIZE];
5.     coffee_page_t next_free;
6.     char gc_wait;
7. } protected_mem;
8.
9. static struct file *const coffee_files = protected_mem.coffee_files;
10. static struct file_desc *const coffee_fd_set = protected_mem.coffee_fd_set;
11. static coffee_page_t *const next_free = &protected_mem.next_free;
12. static char *const gc_wait = &protected_mem.gc_wait;
```

## 2.2 COFFEE\_ERASE

Coffee 格式化是按区擦除，可以把 COFFEE\_SECTOR\_SIZE 设成几倍的块大小(片外 FLASH，即 NAND FLASH)或者几倍的页大小(片上 FLASH，即 NOR FLASH，按页擦除或整个 FLASH 擦除)，这将有利于容量大的存储设备。[2]是这么说的，COFFEE\_SECTOR\_SIZE 用于应付大的存储设备(比如 SD 卡)，在这种情况下，将其设置大一点可加快顺序扫描速度。

COFFEE\_ERASE 宏被定义到硬件相关的擦除函数(这也是 Coffee 文件移植需做的事，在 cfs-coffee-arch.h 文件中)，如下：

```
1. #define COFFEE_ERASE(sector) stm32_flash_erase(sector)
```

stm32\_flash\_erase 函数源代码(在 cfs-coffee-arch.c 中)如下:

```
1. void stm32_flash_erase(u8_t sector)
2. {
3.     u32_t addr = COFFEE_START + (sector) *COFFEE_SECTOR_SIZE;
4.     u32_t end = addr + COFFEE_SECTOR_SIZE;
5.
6.     /* This prevents from accidental write to CIB. */
7.     if (!(addr >= FLASH_START && end <= (FLASH_START+FLASH_PAGES*COFFEE_SECTOR_SIZE)))
8.     {
9.         return ;
10.    }
11.    FLASH_Unlock();
12.    FLASH_ErasePage(addr);
13. }
```

咋一看, 上述的代码有只有当 COFFEE\_SECTOR\_SIZE(逻辑区大小)与 FLASH\_PAGE\_SIZE(FLASH 物理页大小)相等的时候, 才不会有问题。若 COFFEE\_SECTOR\_SIZE 大于 FLASH\_PAGE\_SIZE, 但 FLASH\_ErasePage 只擦除一页, 那余下的就没擦除了; 倘若 COFFEE\_SECTOR\_SIZE 小于 FLASH\_PAGE\_SIZE, 那还多擦除了部分。在我们的例子, 将 COFFEE\_SECTOR\_SIZE 被设成 FLASH 物理页大小 FLASH\_PAGE\_SIZE, 所以没出现问题。

细心的你会发现, 即便如此也还有问题, 因为 FLASH 是页读写, 按块擦除, 而块通常由若干页组成, 而代码却是按页擦除。但你忽略了另一个问题, 平台相关性, 我们例子的 FLASH 是片上 FLASH, 即 NOR FLASH, 与内存统一编址, 可以随机读取, 按页擦除或者整块擦除, 所以一直没出问题。

## 2.3 memset

**memset** 函数功能是将一段内存块填充某个给定的值，通常用于对较大结构体或数组清零操作。memset(void\*s, int c, size\_t n)，即将从地址 s 开始往后的 n 个字节都设成 c[5]。在这里，将 protected\_mem 结构体（见 2.1）的成员变量都设成 0（数组的每个元素也为 0）。奇怪了，实现这个函数怎么会在 Install\_Software\IAR Systems\Embedded Workbench 5.4\arm\INC 呢？源码如下，完全看不懂：-（

```
1. #define _DLIB_STRING_SKIP_INLINE_MEMSET
2. #pragma inline
3. void *memset(void *_D, int _C, size_t _N)
4. {
5.     __aeabi_memset(_D, _N, _C);
6.     return _D;
7. }
```

至此 Coffee 文件系统格式化完毕：-)

更多Contiki学习笔记可通过博文《[Contiki学习笔记：目录](#)》索引访问。

### 参考资料：

[1] [Enabling Large-Scale Storage in Sensor Networks with the Coffee File System.pdf](#)

- [2] [File Systems - ContikiWiki](#)
- [3] 文章《[什么是低级格式化 硬盘低级格式化的作用和原理](#)》
- [4] 维基百科词条: [格式化](#)
- [5] 博文《[memset用法详解](#)》

[Contiki学习笔记: Coffee文件系统创建文件](#) (2011-12-16 15:28)

标签: [Coffee](#) [Contiki](#) [cfs\\_open](#) [创建文件](#) [学习笔记](#) 分类: [Contiki](#)

### 摘要:

在 Coffee, 没有单独的文件创建函数。在 name 对应文件不存在的情况下, 为 cfs\_open 指定 flags(取 000 或 010 或 011 或 100 或 110 或 111)可创建新文件, 本文深入源码并用多组示意图分析 Coffee 创建文件技术细节。包括 find\_file、page\_count、reserve、find\_contiguous\_pages、write\_header。

## 一、概述

在 Linux, 用以下两种方式创建一个新文件(两种方式等价):

1. `int creat(const char *pathname, mode_t mode);` //不足之处是它以只写的方式打开创建的文件
2. `int open(const char *pathname, O_WRONLY|O_CREAT|O_TRUNC, mode_t mode);` //mode 指定访问权限

但在 Coffee，没有单独的文件创建函数。虽有 `cfs_open` 函数，但 `flags` 只能是 `CFS_READ`、`CFS_WRITE`、`CFS_APPEND`，没有类似 `O_CREAT`，最开始以为 Coffee 少了创建文件相关函数（因为 Contiki 源码包的测试例子，有些问题），最近花了不少时间分析 Coffee 打开文件函数 `cfs_open`，才搞明白。

打开一个物理上不存在的文件，Coffee 会试图创建新的文件。之所以说“试图”，是因为创建并不一定会成功，得满足以下条件，才会创建成功：

- (1) `cfs_open` 函数的 `flags` 不能是 `(CFS_READ|CFS_APPEND)` 或者 `CFS_READ`，见 2.2
- (2) 物理 FLASH 有连续 pages 空闲页或者经过垃圾回收后有连续的 pages 空闲页

值得注意的是，即便文件创建成功也可能返回-1（当 `coffee_files[COFFEE_MAX_OPEN_FILES]` 数组没有可用项时，新创建的文件并没有缓存到数组 `coffee_files`）。

## 二、源码分析

Coffee 打开文件 `cfs_open` 函数分析见博文《Contiki 学习笔记：Coffee 文件系统打开文件 `cfs_open`》，以下选取与文件创建相关代码进行分析，如下：

```
1. fdp->file = find_file(name); //见 2.1
2. if (fdp->file == NULL) //物理上没有 name 对应的文件 file，则试图创建之
3. {
4.     if ((flags & (CFS_READ | CFS_WRITE)) == CFS_READ) //检查 flags 是否符合创建文件要求，见 2.2
5.     {
6.         return -1;
```

```
7.     }
8.
9.     fdp->file = reserve(name, page_count(COFFEE_DYN_SIZE), 1, 0); //见三
10.    if (fdp->file == NULL)
11.    {
12.        return - 1;
13.    }
14.
15.    fdp->file->end = 0; //新创建的空文件，文件末尾自然是 0
16.}
17.
18./*此处略去与文件创建无关的 4 行代码*/
19.
20.fdp->flags |= flags;
21.fdp->offset = flags & CFS_APPEND ? fdp->file->end: 0; //如果 flags 没有设置 CFS_APPEND，则文件偏移量设为 0
22.fdp->file->references++; //文件引用次数加 1
23.
24.return fd;
```

## 2.1 find\_file

执行 find\_file 函数，若 name 对应的文件 file 还驻留在内存(即若 file 还在 coffee\_files[COFFEE\_MAX\_OPEN\_FILES] 数组里)，并且对应的物理文件是有效的，则直接返回 file，否则扫描整个 FLASH，将 name 对应文件 file(在 FLASH 中但没缓存)缓存到内存(这一步得确保 coffee\_files 数组有可用的项，否则返回空 NULL，参见 load\_file 函数)。关于 find\_file 详细分析见《Contiki 学习笔记：Coffee 文件系统打开文件 cfs\_open》。显然，这里要处理的情况是后者，即物理上没有 name 对应的文件 file，则试图创建之。

## 2.2 cfs\_open 函数 flags 判断

首先检查 cfs\_open 参数 flags，flags 至少有一项是 CFS\_WRITE，才有可能创建新文件，源代码如下：

```
1. if ((flags & (CFS_READ | CFS_WRITE)) == CFS_READ) //若 flags 仅仅是 CFS_READ 则返回-1，即不创建新文件
2. {
3.     return -1;
4. }
```

那么，if 条件在什么时候会返回真呢，如下图所示，当 flags 中没有 CFS\_WRITE 且有 CFS\_READ 时 (CFS\_APPEND 可有可无)，即 flags 取值为 001、101，条件为真。也就是说，

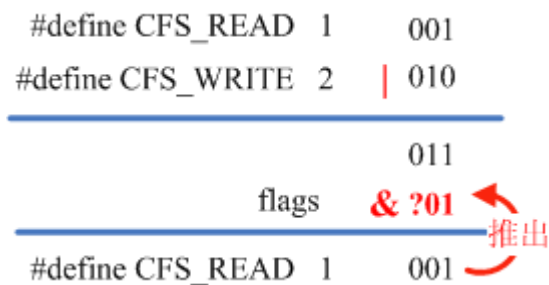


图 1 cfs\_open 函数 flags 判断示意图

当 flags 含有 CFS\_WRITE 或者没有 CFS\_READ 或者两者皆有之时才有可能创建新文件 (CFS\_APPEND 可有可无)，即 flags 取 000、010、011、100、110、111。(这样的设计让我很费解)

## 2.3 page\_count



page\_count 函数用于算出给定的 size 需要多少 Coffee 页，即 size 加上 file\_header 大小，并且按 Coffee 页 (Coffee 页大小与 FLASH 物理页大小不一定等同，见 cfs\_coffee\_arch.h 文件) 对齐，示意图如下：

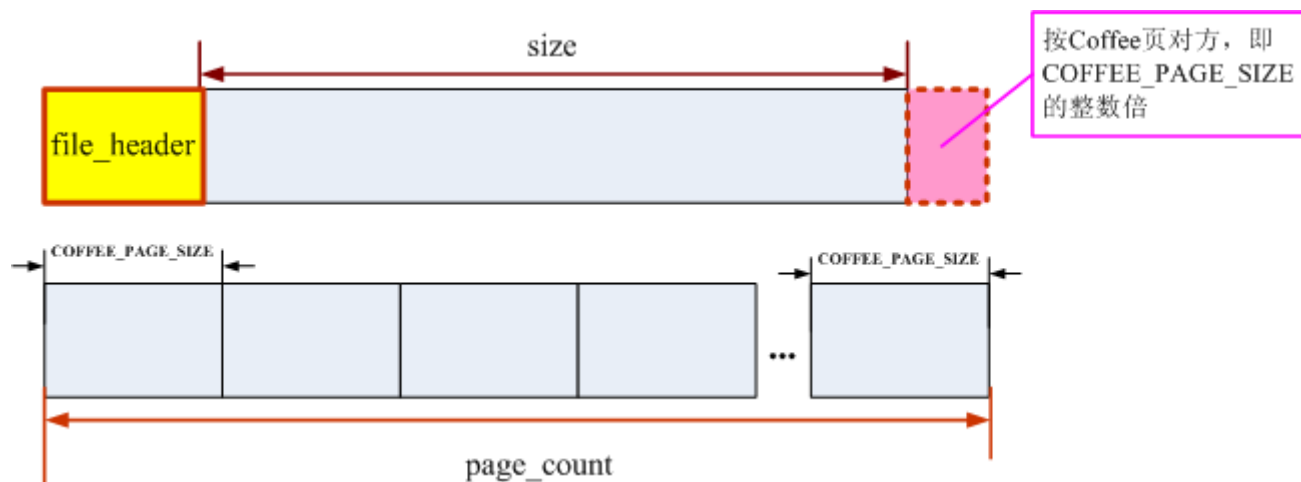


图 2 page\_count 函数示意图

在本例中，page\_count 返回 2，即 2 页 Coffee\_Page，源代码如下：

```
1. //page_count(COFFEE_DYN_SIZE)
2. #define COFFEE_DYN_SIZE (COFFEE_PAGE_SIZE*1) //在 cfs_coffee_arch.h 可以配置
3.
4. static coffee_page_t page_count(cfs_offset_t size)
5. {
6.     return (size + sizeof(struct file_header) + COFFEE_PAGE_SIZE - 1) / COFFEE_PAGE_SIZE;
7. }
```

### 三、创建文件主体函数 `reserve`

`reserve` 是创建新文件的主体函数，首先进行参数验证，接着查看物理 FLASH 是否有连续 `pages` 空闲页，若有，则返回该文件即将占有页的第一页。否则，调用 Coffee 垃圾回收，再次查看物理 FLASH 是否有连续 `pages` 空闲页，如果还没有就返回 `NULL`。

值得注意的是，即便文件创建成功也可能返回-1(当 `coffee_files[COFFEE_MAX_OPEN_FILES]` 数组没有可用项时，新创建的文件并没有缓存到数组 `coffee_files`)。源代码如下：

```
1. //fdp->file = reserve(name, page_count(COFFEE_DYN_SIZE), 1, 0);
2. static struct file *reserve(const char *name, coffee_page_t pages, int allow_duplicates, unsigned flags)
3. {
4.     struct file_header hdr;
5.     coffee_page_t page;
6.     struct file *file;
7.
8.     if (!allow_duplicates && find_file(name) != NULL) //参数验证，在本例中，!allow_duplicates 为 0，条件为假
9.     {
10.         return NULL;
11.     }
12.
13.     page = find_contiguous_pages(pages); //查看物理 FLASH 是否有连续 pages 空闲页，若有，则返回该文件即将占有页的
    第一页。见 3.1
14.     if (page == INVALID_PAGE) //没有可用的页，尝试着调用垃圾回收
15.     {
16.         if (*gc_wait)
```

```
17.     {
18.         return NULL;
19.     }
20.     collect_garbage(GC_GREEDY); //垃圾回收
21.     page = find_contiguous_pages(pages); //再次查看物理 FLASH 是否有连续 pages 空闲页, 见 3.1
22.     if (page == INVALID_PAGE) //如果垃圾回收后还没找到可用的页, 就返回 NULL
23.     {
24.         *gc_wait = 1;
25.         return NULL;
26.     }
27. }
28.
29. memset(&hdr, 0, sizeof(hdr)); //将 hdr 成员变量初始化为 0
30. memcpy(hdr.name, name, sizeof(hdr.name) - 1); //给 file_header 的成员变量 name 赋值
31. hdr.max_pages = pages; //实参 pages, 在本例等于 2
32. hdr.flags = HDR_FLAG_ALLOCATED | flags; //file_header 的 flags 的 A 位置 1, 表示文件正在使用
33. write_header(&hdr, page); //将 file_header 写入物理 FLASH, 见 3.2
34.
35. PRINTF("Coffee: Reserved %u pages starting from %u for file %s\n", pages, page, name);
36.
37. file = load_file(page, &hdr); //如果 coffee_files[COFFEE_MAX_OPEN_FILES]数组没有可用项, 就返回 NULL 了(但此
    时文件已创建完毕)
38. if (file != NULL)
39. {
40.     file->end = 0;
41. }
```

```

42.
43.     return file;
44. }

```

### 3.1 find\_contiguous\_pages

find\_contiguous\_pages 从 next\_free(全局变量, 指向下一个空闲页) 指向的页面开始查找, 若找到  $start + amount \leq next\_file()$ , 则返回 start, 否则返回 INVALID\_PAGE(说明没有足够空闲页甚至没有空闲页供使用)。

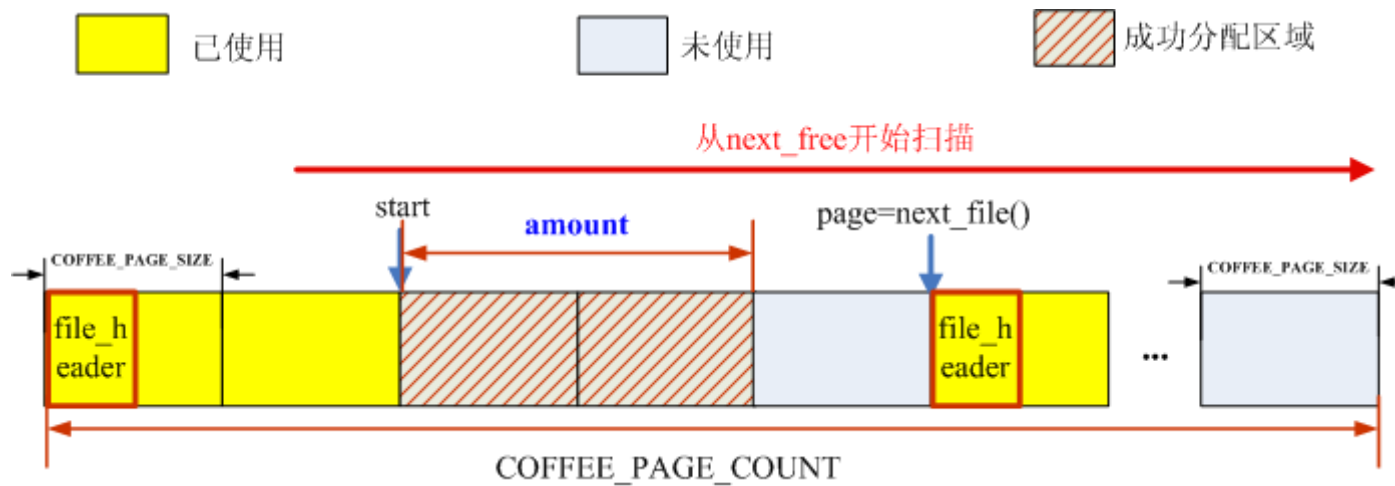


图3 find\_contiguous\_pages 函数示意图

find\_contiguous\_pages 函数源代码如下:

```

1. static coffee_page_t find_contiguous_pages(coffee_page_t amount)

```

```
2. {
3.     coffee_page_t page, start;
4.     struct file_header hdr;
5.
6.     start = INVALID_PAGE; //初始化 start, #define INVALID_PAGE ((coffee_page_t)-1)
7.
8.     for (page = *next_free; page < COFFEE_PAGE_COUNT;) //从 next_free 指向的页面开始查找
9.     {
10.        read_header(&hdr, page);
11.        if (HDR_FREE(hdr)) //判断 file_header 的 flags 中 A(Allocated)位, 若 A 位为 0, 则 HDR_FREE(hdr)返回真, 即页
            面是空闲的
12.        {
13.            if (start == INVALID_PAGE)
14.            {
15.                start = page;
16.                if (start + amount >= COFFEE_PAGE_COUNT) //如果剩下的页面数不足以分配, 则跳出循环, 直接返回
                    INVALID_PAGE
17.                {
18.                    break;
19.                }
20.            }
21.
22.            /**若找到 start+amount<=next_file(), 则返回 start***/
23.            page = next_file(page, &hdr); //返回下一个文件占用 Coffee 页的第一页(从 page 页开始)
24.            if (start + amount <= page)
25.            {
```

```
26.         if (start == *next_free)
27.         {
28.             *next_free = start + amount; //找到空闲页，分配成功。更新 next_free
29.         }
30.         return start;
31.     }
32. }
33. else //如果文件正在使用(即 A 位为 1)，跳到下一个文件
34. {
35.     start = INVALID_PAGE;
36.     page = next_file(page, &hdr);
37. }
38. }
39. return INVALID_PAGE;
40. }
```

### 3.2 write\_header

write\_header 将 file\_header 写入物理 FLASH，源代码如下：

```
1. static void write_header(struct file_header *hdr, coffee_page_t page)
2. {
3.     hdr->flags |= HDR_FLAG_VALID; //将 file_header 的 flags 中的 V 位置 1，即标记文件头是完整的
4.     COFFEE_WRITE(hdr, sizeof(*hdr), page *COFFEE_PAGE_SIZE);
5. }
6.
```

7. #define HDR\_FLAG\_VALID 0x1
8. #define COFFEE\_WRITE(buf, size, offset) stm32\_flash\_write(COFFEE\_START + offset, buf, size)

本文图 1~3 源文件如下:



[find contiguous pages函数示意图.rar](#)



[cfs\\_open函数flags判断示意图.rar](#)



[page count函数示意图.rar](#)

[Contiki学习笔记: Coffee文件系统打开文件cfs\\_open](#) (2011-12-18 13:08)

标签: [Coffee](#) [Contiki](#) [cfs\\_open](#) [学习笔记](#) [打开](#) 分类: [Contiki](#)

### 摘要:

本文深入源码分析 Coffee 文件系统打开文件 cfs\_open 函数技术细节, 包括 flags、get\_available\_fd、file\_end、find\_file、FILE\_FREE、read\_header、next\_file、load\_file。

## 一、cfs\_open

cfs\_open 用于打开一个文件，成功打开返回文件描述符 fd，否则返回-1。如果文件不存在，则试图创建新文件。cfs\_open 首先获得最小未使用的文件描述符 fd(若没有可用的 fd，则返回-1)，而后查看文件是否已缓存并且对应物理文件有效，若是则打开成功。若物理上没有该文件，则试图创建。**值得注意的是，即便文件创建成功也可能返回-1**(当 coffee\_files[COFFEE\_MAX\_OPEN\_FILES] 数组没有可用项时)。

cfs\_open 算法流程图如下：



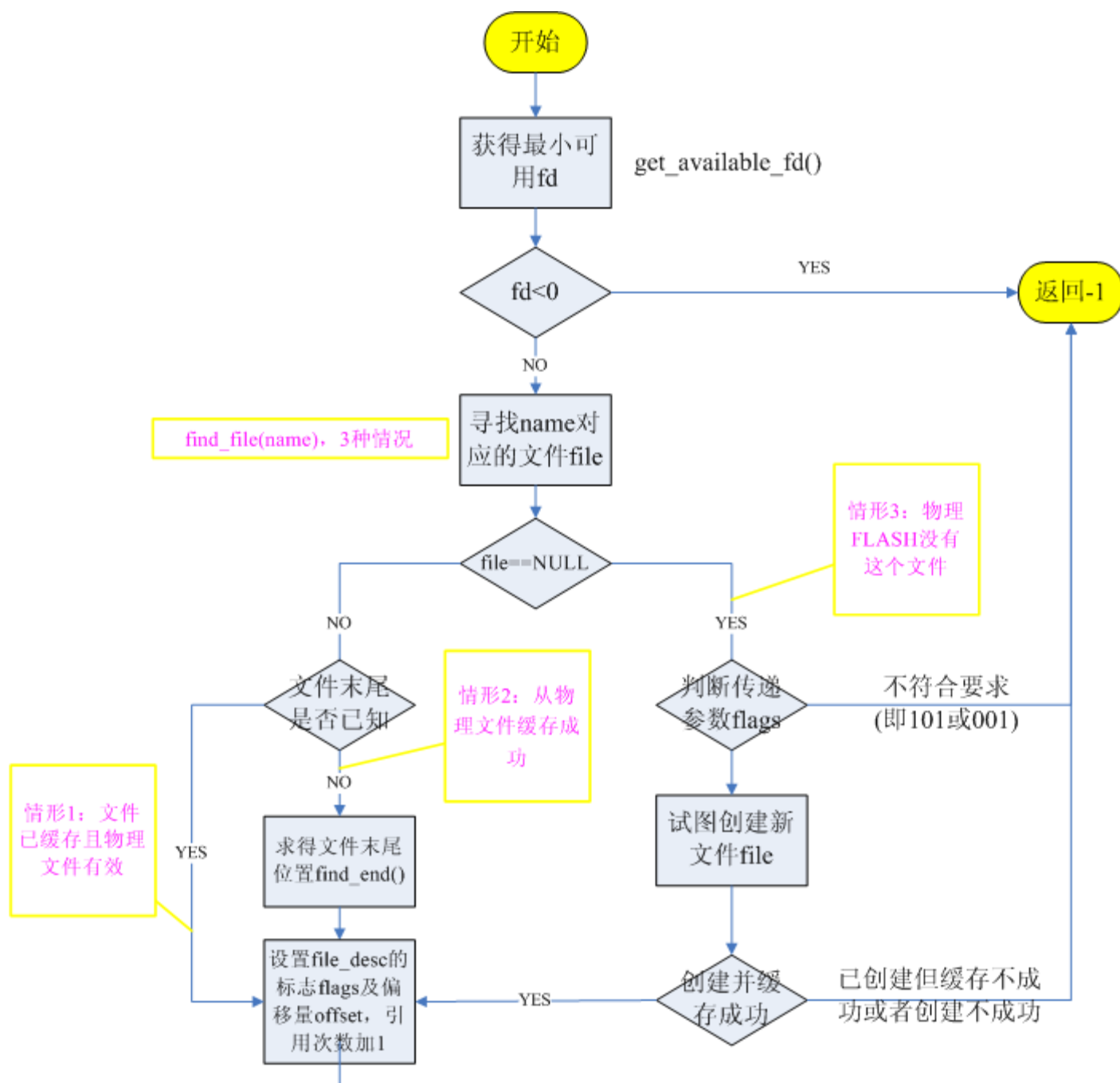


图1 cfs\_open 算法流程

cfs\_open 源代码如下:

```
1. int cfs_open(const char *name, int flags) //flags 见 1.1
2. {
3.     int fd;
4.     struct file_desc *fdp;
5.
6.     fd = get_available_fd(); //获得最小可用的 fd, 见 1.2
7.     if (fd < 0)
8.     {
9.         PRINTF("Coffee: Failed to allocate a new file descriptor!\n");
10.        return -1;
11.    }
12.
13.    fdp = &coffee_fd_set[fd];
14.    fdp->flags = 0; //此处的 0 表示 COFFEE_FD_FREE, 源码#define COFFEE_FD_FREE 0x0
15.
16.    fdp->file = find_file(name); //寻找 name 对应的文件(不存在、在物理 FLASH 但没缓存、缓存), 见二
17.
18.    /**没有 name 对应的文件, 则试图创建新文件, 见 1.3***/
19.    if (fdp->file == NULL)
20.    {
21.        if ((flags & (CFS_READ | CFS_WRITE)) == CFS_READ) //若 flags 是 001 或 101, 则返回-1
22.        {
```

```
23.         return - 1;
24.     }
25.     fdp->file = reserve(name, page_count(COFFEE_DYN_SIZE), 1, 0);
26.     if (fdp->file == NULL)
27.     {
28.         return - 1;
29.     }
30.     fdp->file->end = 0; //新创建的空文件，文件末尾自然是 0
31. }
32. /*file 不为 NULL (即找到文件并缓存)，寻找文件末尾位置*/
33. else if (fdp->file->end == UNKNOWN_OFFSET) //找到文件末尾的位置，见 1.4
34. {
35.     fdp->file->end = file_end(fdp->file->page);
36. }
37.
38. fdp->flags |= flags;
39. fdp->offset = flags & CFS_APPEND ? fdp->file->end: 0; //如果文件打开方式是追加 APPEND，则将文件偏移量 offset
    设到文件末尾，否则设成 0
40. fdp->file->references++; //文件引用次数加 1
41.
42. return fd;
43. }
```

### 1.1 flags

flags 表示文件是以什么 (只读、写、追加) 打开，源码如下 (在 contiki/core/cfs/cfs.h):

```
1. #ifndef CFS_READ
2.     #define CFS_READ 1
3. #endif
4.
5. #ifndef CFS_WRITE
6.     #define CFS_WRITE 2
7. #endif
8.
9. #ifndef CFS_APPEND
10.    #define CFS_APPEND 4
11. #endif
```

可以单独设 CFS\_READ、CFS\_WRITE、CFS\_APPEND 其中的一个,也可以用或(即“|”)连接设置多个 flags(如 CFS\_READ | CFS\_APPEND)。设置了 CFS\_APPEND 意味着 CFS\_WRITE 已被设置(注 1),即从文件末尾写。如果设置 CFS\_WRITE 而没有设置 CFS\_APPEND,则文件被截断(truncate)为 0,即文件大小为 0 了[2](这点跟 Linux 不同)。

注 1: 尽管[2]ContikiWiki,指出设置了 CFS\_APPEND 意味着 CFS\_WRITE 也被设置了,最近在分析写入文件 cfs\_write,发现情况并非如此。必须指定 CFS\_WRITE 才能进行写,详情见博文《Contiki 学习笔记: Coffee 文件系统写入文件 cfs\_write》,部分源码如下:

```
1. if(!(FD_VALID(fd) && FD_WRITABLE(fd)))
2. {
3.     return - 1;
4. }
5.
6. #define FD_WRITABLE(fd) (coffee_fd_set[(fd)].flags & CFS_WRITE)
```

## 1.2 get\_available\_fd

类似于 Linux，所有打开文件都通过文件描述符 fd 引用，函数 get\_available\_fd 返回最小未使用的描述符(即缓存 file\_desc 数组 coffee\_fd\_set[] 首个 flags 为 COFFEE\_FD\_FREE)，没有可用的 fd 则返回-1。源码如下：

```
1. static int get_available_fd(void)
2. {
3.     int i;
4.     for (i = 0; i < COFFEE_FD_SET_SIZE; i++)
5.     {
6.         if (coffee_fd_set[i].flags == COFFEE_FD_FREE)
7.         {
8.             return i;
9.         }
10.    }
11.    return - 1;
12. }
```

Coffee将文件描述符file\_desc组织成一个数组，作为protected\_mem\_t的一个成员变量，系统缓存file\_desc示意图如下，详情见博文《[Contiki学习笔记：Coffee文件组织及若干数据结构](#)》。

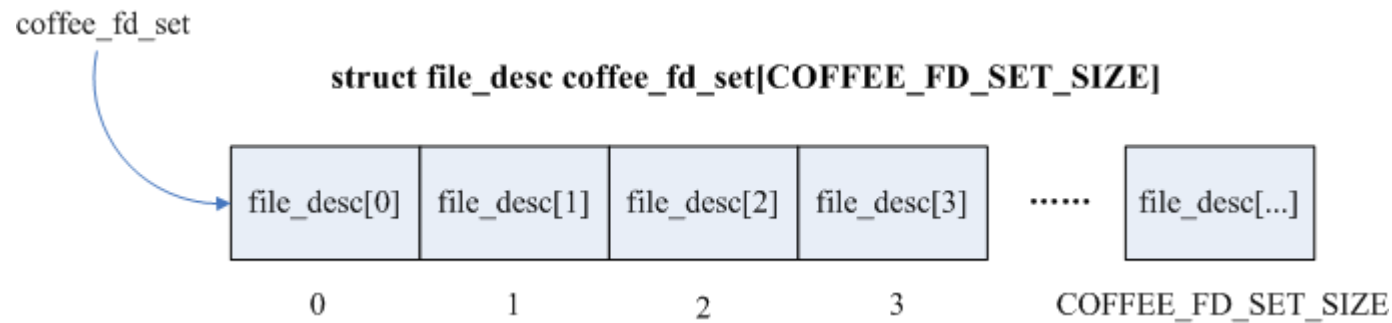


图 2 Coffee 缓存 file\_desc 示意图

### 1.3 创建文件

执行 `find_file` 函数, 若 `name` 对应的文件 `file` 还驻留在内存 (即是否还在 `coffee_files[COFFEE_MAX_OPEN_FILES]` 数组里), 并且对应的物理文件是有效的, 则直接返回 `file`, 否则扫描整个 FLASH, 将 `name` 对应文件 `file` (在 FLASH 中但没缓存) 缓存到内存 (这一步得确保 `coffee_files` 数组有可用的项, 否则返回空 `NULL`, 参见 `load_file` 函数)。如果 FLASH 没有 `name` 对应的文件, 也返回 `NULL`。显然, 这里要处理的情况是后者, 则试图创建之。

首先检查 `cfs_open` 参数 `flags`, 若 `flags` 是 001 (`CFS_READ`) 或 101 (`CFS_APPEND|CFS_READ`), 则返回 -1, 即 创建不成功。否则调用 `reserve` 为其分配新页面, 并对 `file_header` 作一些设置。其中 `page_count (COFFEE_DYN_SIZE)` 求得 `COFFEE_DYN_SIZE` 字节需要分配多少 Coffee 页 (因为还要包括 `file_header` 大小且需考虑页对齐)。详情见博文 [《Contiki 学习笔记: Coffee 文件系统创建文件》](#)。

### 1.4 找到文件末尾位置 `file_end`

`find_file(const char *name)`, 如果 `name` 对应的文件没有被缓存或者是缓存了但物理文件失效, 则调用 `load_file` 将对应的物理 FLASH 文件缓存到 RAM (物理上 有这个文件且 `coffee_files` 有可用的项), 此时 `file_end` 是 `UNKNOWN_OFFSET`。因为 Coffee 没有

在 `file_header` 存储文件末尾的位置(因为文件长度经常改变, 加之 FLASH 先擦后写特性), 所以打开一个文件时必须找到文件末尾的位置。

如图 2 所示, `cfs_offset_t file_end` 从文件的最后一页 `max_pages-1` 往前找, 直到找到空字节(即 `0x00`), 就找到 `file_end` 了, 否则返回 0。返回 0 有两种情况, 一种是

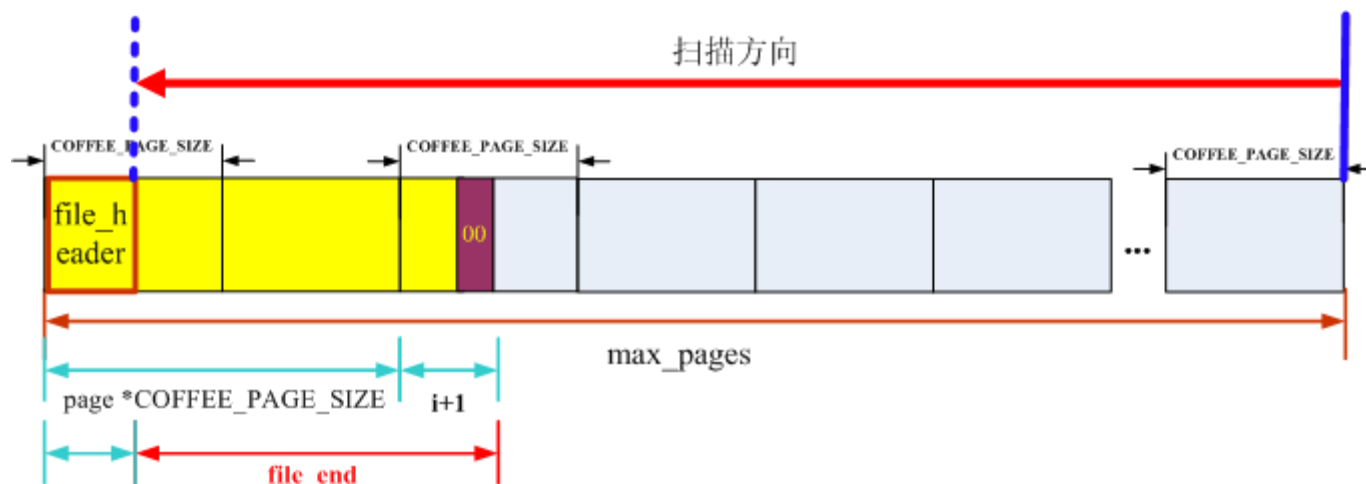


图 3 `file_end` 函数示意图

`file_end` 源代码如下:

```
1. //fdp->file->end = file_end(fdp->file->page);
2. static cfs_offset_t file_end(coffee_page_t start)
3. {
4.     struct file_header hdr;
```

```
5.   unsigned char buf[COFFEE_PAGE_SIZE];
6.   coffee_page_t page;
7.   int i;
8.
9.   read_header(&hdr, start); //读取文件头 file_header
10.
11.  /* An important implication of this is that if the last written bytes are zeroes, then these are skipped from
    the calculation.*/
12.  for (page = hdr.max_pages - 1; page >= 0; page--) //从文件最后一页(预留的页而不是实际占有)往前找
13.  {
14.      COFFEE_READ(buf, sizeof(buf), (start + page) *COFFEE_PAGE_SIZE); //将 Coffee 页读入 buf
15.      for (i = COFFEE_PAGE_SIZE - 1; i >= 0; i--)
16.      {
17.          if (buf[i] != 0)
18.          {
19.              if (page == 0 && i < sizeof(hdr)) //没必要检查 file_header
20.              {
21.                  return 0;
22.              }
23.
24.              return 1+i + (page *COFFEE_PAGE_SIZE) - sizeof(hdr);
25.          }
26.      }
27.  }
28.
29.  return 0; //All bytes are writable
```



30. }

## 二、find\_file

find\_file 函数用于找到文件名name对应的file指针。若name对应的文件file是还驻留在内存(即还在 coffee\_files[COFFEE\_MAX\_OPEN\_FILES]数组里), 并且对应的物理文件是有效的, 则直接返回file指针, 否则扫描整个 FLASH, 将name 对应文件file(在FLASH中但没缓存)缓存到内存(这一步得确保coffee\_files数组有可用的项, 否则返回空 NULL, 参见 load\_file函数)。file可以理解成是file\_header的缓存, 但不等同(可参见博文《[Contiki学习笔记: Coffee文件组织及若干数据结构](#)》)。源代码如下:

```
1. static struct file *find_file(const char *name)
2. {
3.     int i;
4.     struct file_header hdr;
5.     coffee_page_t page;
6.
7.     /*首先检查 name 对应的文件 file 是否还驻留在内存(是否缓存了 file_header), 即是否还在
   coffee_files[COFFEE_MAX_OPEN_FILES]数组里, 若有, 则直接返回 file*/
8.     for (i = 0; i < COFFEE_MAX_OPEN_FILES; i++) //遍历整个 coffee_files[COFFEE_MAX_OPEN_FILES]数组
9.     {
10.        /*若没有 max_pages 不为 0 的文件 file, 则扫描整个 FLASH(即跳出 for 循环, 执行后面内容)*/
11.        if (FILE_FREE(&coffee_files[i])) //file 的 max_pages 是否为 0, 见 2.1
12.        {
13.            continue;
14.        }
15.
```

```
16.      /*如果找到了 RAM 缓存的 file 数组 coffee_files[COFFEE_MAX_OPEN_FILES]中有空闲的项*/
17.      read_header(&hdr, coffee_files[i].page); //读取文件的元数据, 即 file_header, 详情见 2.2
18.      if (HDR_ACTIVE(hdr) && !HDR_LOG(hdr) && strcmp(name, hdr.name) == 0) //通过文件头 file_header 判断物理文
      件是否有效, 见 2.3
19.      {
20.          return &coffee_files[i]; //文件元数据 file_header 被缓存且物理文件有效, 则直接返回 file
21.      }
22.  }
23.
24.  /*否则顺序扫描整个 FLASH(没有缓存该文件 file)*/
25.  for (page = 0; page < COFFEE_PAGE_COUNT; page = next_file(page, &hdr)) //next_file 见 2.4
26.  {
27.      read_header(&hdr, page);
28.      if (HDR_ACTIVE(hdr) && !HDR_LOG(hdr) && strcmp(name, hdr.name) == 0) //见 2.3
29.      {
30.          return load_file(page, &hdr); //见 2.5
31.      }
32.  }
33.
34.  return NULL; //如果物理 FLASH 没有该 name 对应的文件, 也返回 NULL
35. }
```

典型的情况, 系统启动后, 第一次打开文件, Coffee文件系统需要顺序扫描整个FLASH, 因为Coffee格式化将 coffee\_files[COFFEE\_MAX\_OPEN\_FILES]数组的结构成员变量都初始化为 0(当然也包括max\_pags), 详情见博文 [《Contiki学习笔记: Coffee文件系统格式化cfs coffee format》](#)。

## 2.1 FILE\_FREE

max\_pages是file的成员变量，用来表示文件预设的最大页面数(所有的Coffee文件创建时都会被预先设置)。在Coffee格式化时(参见博文《[Contiki学习笔记: Coffee文件系统格式化cfs\\_coffee\\_format](#)》)，max\_pages被置为0。当删除一个文件cfs\_remove时，max\_pages也被置为0。

```
1. #define FILE_FREE(file) ((file)->max_pages == 0)
```

## 2.2 read\_header

read\_header 用于读取物理文件的元数据(即 file\_header)，源码如下：

```
1. static void read_header(struct file_header *hdr, coffee_page_t page)
2. {
3.     COFFEE_READ(hdr, sizeof(*hdr), page *COFFEE_PAGE_SIZE);
4.     #if DEBUG
5.         if (HDR_ACTIVE(*hdr) && !HDR_VALID(*hdr))
6.         {
7.             PRINTF("Invalid header at page %u!\n", (unsigned)page);
8.         }
9.     #endif
10. }
```

宏 COFFEE\_READ 直接映射到平台相关的 FLASH 读函数(如 stm32\_flash\_read)，从 offset 处读取 size 字节存放到 buf。其中 offset 是指从文件系统管理的 FLASH 处(即 COFFEE\_START，因为有些 NOR FLASH 将一部分空间用于存放代码)开始算的偏移量。

```
1. #define COFFEE_READ(buf, size, offset) stm32_flash_read(COFFEE_START+offset, buf, size)
```

```
2.
3. void stm32_flash_read(u32_t address, void *data, u32_t length)
4. {
5.     u8_t *pdata = (u8_t*)address;
6.     ENERGEST_ON(ENERGEST_TYPE_FLASH_READ);
7.     memcpy(data, pdata, length);
8.     ENERGEST_OFF(ENERGEST_TYPE_FLASH_READ);
9. }
```

### 2.3 file\_header 的 flags

文件头file\_header(文件的元数据)的flags, 为方便操作, Coffee将这些flags单独定义成宏, 源码如下, 各个标志位的含义见博文《[Contiki学习笔记: Coffee文件组织及若干数据结构](#)》。

```
1. /* File header flags. */
2. #define HDR_FLAG_VALID      0x1 /* Completely written header. */
3. #define HDR_FLAG_ALLOCATED  0x2 /* Allocated file. */
4. #define HDR_FLAG_OBSOLETE   0x4 /* File marked for GC. */
5. #define HDR_FLAG_MODIFIED   0x8 /* Modified file, log exists. */
6. #define HDR_FLAG_LOG        0x10 /* Log file. */
7. #define HDR_FLAG_ISOLATED 0x20 /* Isolated page. */
```

Coffee 定义了若干宏来判断文件的状态(依据 file\_header 中的 flags), 如下:

```
1. /* File header macros. */
2. #define CHECK_FLAG(hdr, flag) ((hdr).flags & (flag))
3.
```

```

4. #define HDR_VALID(hdr) CHECK_FLAG(hdr, HDR_FLAG_VALID)           //若 V 标志置 1，则返回真
5. #define HDR_ALLOCATED(hdr) CHECK_FLAG(hdr, HDR_FLAG_ALLOCATED)   //若 A 标志置 1，则返回真
6. #define HDR_OBSOLETE(hdr) CHECK_FLAG(hdr, HDR_FLAG_OBSOLETE)     //若 O 标志置 1，则返回真
7. #define HDR_MODIFIED(hdr) CHECK_FLAG(hdr, HDR_FLAG_MODIFIED)     //若 M 标志置 1，则返回真
8. #define HDR_LOG(hdr) CHECK_FLAG(hdr, HDR_FLAG_LOG)               //若 L 标志置 1，则返回真
9. #define HDR_ISOLATED(hdr) CHECK_FLAG(hdr, HDR_FLAG_ISOLATED)     //若 I 标志置 1，则返回真
10.
11. #define HDR_FREE(hdr) !HDR_ALLOCATED(hdr) //若 A 标志置 0，则返回真
12. #define HDR_ACTIVE(hdr) (HDR_ALLOCATED(hdr) && !HDR_OBSOLETE(hdr) && !HDR_ISOLATED(hdr)) //若 A 为 1，O 为 0，I 为 0，则返回真

```

从以上分析可以看出，`(HDR_ACTIVE(hdr) && !HDR_LOG(hdr) && strcmp(name, hdr.name)==0)` 含义就是文件正在使用(A)，并且不是孤立(I)和无效(O)页，微日志文件没有被修改(?)，且文件名相同，只有这样该文件的元数据才算被缓存，够复杂的吧:-) **事实上，就是判断物理文件有效性。**考虑这样的情况(RAM 确实缓存了文件元数据 file\_header，但物理文件已失效)，写入文件数据超过文件预留的大小，此时，Coffee 会新建一个文件，将原始文件和微日志文件拷贝到新文件，并将原来的文件标记为失效。

## 2.4 next\_file 函数

Coffee 用快速跳跃(quick-skip)算法，快速跳过空闲的区域，以快速寻找下一个文件。算法最坏情况是遇到多个长连续的孤立页(multiple long sequences of isolated pages)，但这种情况并不常见[3]。源码如下：

```

1. static coffee_page_t next_file(coffee_page_t page, struct file_header *hdr)
2. {
3.     if (HDR_FREE(*hdr))
4.     {
5.         return (page + COFFEE_PAGES_PER_SECTOR) & ~(COFFEE_PAGES_PER_SECTOR - 1);
6.     }

```

```
7.     else if (HDR_ISOLATED(*hdr))
8.     {
9.         return page + 1;
10.    }
11.    return page + hdr->max_pages;
12. }
```

## 2.5 load\_file

load\_file 将文件的元数据 file\_header 缓存到 RAM 的 file，如果能在 coffee\_files[COFFEE\_MAX\_OPEN\_FILES] 数组中找到可用项(即是否有文件 file 的 max\_pages 为 0 或者引用次数为 0 的项)，就缓存，否则返回空 NULL。源代码如下：

```
1. static struct file *load_file(coffee_page_t start, struct file_header *hdr)
2. {
3.     int i, unreferenced, free;
4.     struct file *file;
5.
6.     /*首先检查内存是否缓存了 file，即是否在 coffee_files[COFFEE_MAX_OPEN_FILES] 数组中*/
7.     for (i = 0, unreferenced = free = - 1; i < COFFEE_MAX_OPEN_FILES; i++) //遍历整个
        coffee_files[COFFEE_MAX_OPEN_FILES] 数组
8.     {
9.         /*试图从 coffee_files[COFFEE_MAX_OPEN_FILES] 数组找到第一个 max_pages 为 0 的 file (表示该项空闲，可使用)*/
10.        if (FILE_FREE(&coffee_files[i]))
11.        {
12.            free = i;
13.            break;

```

```
14.     }
15.     /*退而求其次，查看 coffee_files[COFFEE_MAX_OPEN_FILES]数组是否有引用次数为 0 的项*/
16.     else if (FILE_UNREFERENCED(&coffee_files[i]))
17.     {
18.         unreferenced = i;
19.     }
20. }
21.
22.     /*如果 coffee_files[COFFEE_MAX_OPEN_FILES]数组中的 file，即没有 max_pages 为 0，也没有引用次数为 0 的，那么就
    返回空*/
23.     if (free == - 1)
24.     {
25.         if (unreferenced != - 1)
26.         {
27.             i = unreferenced;
28.         }
29.         else
30.         {
31.             return NULL;
32.         }
33.     }
34.
35.     /*将物理上文件的 file_header(存放在文件的开始处)缓存到内存的 file*/
36.     file = &coffee_files[i];
37.     file->page = start;
38.     file->end = UNKNOWN_OFFSET; //因为文件大小经常变化，所以在 file_header 不存储文件末尾位置
```

```
39.  file->max_pages = hdr->max_pages;
40.  file->flags = 0;
41.  if (HDR_MODIFIED(*hdr))
42.  {
43.      file->flags |= COFFEE_FILE_MODIFIED; //如果文件被修改(表示日志存在), #define COFFEE_FILE_MODIFIED 0x1
44.  }
45.  file->record_count = - 1; //此时还不知道日志数量
46.
47.  return file;
48. }
```

FILE\_UNREFERENCED 宏定义如下, 即判断文件引用次数是否为 0, 源代码如下:

```
1. #define FILE_UNREFERENCED(file) ((file)->references == 0)
```

HDR\_MODIFIED 宏, 源代码如下:

```
1. #define HDR_MODIFIED(hdr) CHECK_FLAG(hdr, HDR_FLAG_MODIFIED)
2.
3. #define CHECK_FLAG(hdr, flag) ((hdr).flags & (flag))
```

莫非内存 file 的 flags 与 file\_header 的 flags 各位含义不同???源码如下:

```
1. #define COFFEE_FILE_MODIFIED 0x1
2. #define HDR_FLAG_MODIFIED 0x8
```



更多Contiki学习笔记可通过博文《[Contiki学习笔记：目录](#)》索引访问。

上文图 1~2 源文件如下：



[Coffee缓存fd示意图.rar](#)



[file\\_end函数示意图.rar](#)



[cfs\\_open算法流程图.rar](#)

参考资料：

- [1] [Enabling Large-Scale Storage in Sensor Networks with the Coffee File System.pdf](#)
- [2] [File Systems - ContikiWiki](#)
- [3] Contiki 源代码

[Contiki学习笔记: Coffee文件系统读取文件cfs\\_read](#) (2011-12-20 18:42)

标签: [Coffee](#) [Contiki](#) [cfs\\_read](#) [学习笔记](#) [读取](#) 分类: [Contiki](#)

## 摘要:

本文分析了 Coffee 文件系统读取文件 `cfs_read` 技术细节,包括 `FD_VALID`、`FD_READABLE`、`FILE_MODIFIED`、`absolute_offset`、`COFFEE_READ`、`COFFEE_MICRO`、`log_param`、`read_header`、`read_log_page`、`adjust_log_config`、`modify_log_buffer`、`get_record_index`。`cfs_read`, 当没有微日志文件时, `cfs_read` 直接从原始文件读取, 否则从微日志文件读取。

## 一、cfs\_read

### 1.1 概述

`cfs_read` 从打开文件 `fd` 读取数据, 存放在 `buf`, 读取的字节数是 `size`, 返回实际读取的字节数。从预定义宏指令可以看出, `cfs_read` 总体上包括两种情况: 配置了微日志与没有配置微日志。但都必须进行参数验证。`cfs_read` 先进行参数验证(`fd` 有效性、读取权限、`size`), 而后再读取(无微日志文件的读和有微日志文件的读), 源码如下:

```
1. int cfs_read(int fd, void *buf, unsigned size)
2. {
3.     struct file_desc *fdp;
4.     struct file *file;
5.
6.     #if COFFEE_MICRO_LOGS
7.         struct file_header hdr;
8.         struct log_param lp;
```

```
9.         unsigned bytes_left;
10.         int r;
11.     #endif
12.
13.     /*****fd 有效性检查及权限判断, 见 1.2*****/
14.     if (!(FD_VALID(fd) && FD_READABLE(fd)))
15.     {
16.         return - 1;
17.     }
18.
19.     /*****
20.     fdp = &coffee_fd_set[fd];
21.     file = fdp->file;
22.
23.     *****/size 参数调整, 见 1.3*****/
24.     if (fdp->offset + size > file->end)
25.     {
26.         size = file->end - fdp->offset;
27.     }
28.
29.
30.     /*****没有微日志文件读, 见二*****/
31.     if (!FILE_MODIFIED(file))
32.     {
33.         COFFEE_READ(buf, size, absolute_offset(file->page, fdp->offset));
34.
```

```
35.     fdp->offset += size;
36.     return size;
37. }
38.
39. /*****有微日志文件读， 见三*****/
40. #if COFFEE_MICRO_LOGS
41.     read_header(&hdr, file->page);
42.     for (bytes_left = size; bytes_left > 0; bytes_left -= r)
43.     {
44.         r = - 1;
45.         lp.offset = fdp->offset;
46.         lp.buf = buf;
47.         lp.size = bytes_left;
48.
49.         r = read_log_page(&hdr, file->record_count, &lp);
50.
51.         if (r < 0)
52.         {
53.             COFFEE_READ(buf, lp.size, absolute_offset(file->page, fdp->offset));
54.             r = lp.size;
55.         }
56.         fdp->offset += r;
57.         buf = (char*)buf + r;
58.     }
59. #endif
60.
```

```
61.     return size;  
62. }
```

## 1.2 FD\_VALID 宏和 FD\_READABLE 宏

### (1)FD\_VALID

FD\_VALID 宏用于判断 cfs\_read 函数传进来的 fd 是否有效, 不能小于 0(如, -1 是 get\_available\_fd 函数分配不到 fd 的返回值), 也不能大于 FD 的上限(即 COFFEE\_FD\_SET\_SIZE), 其对应的 file\_desc 的标志 flags 不能为空闲(COFFEE\_FD\_FREE), 源码 如下:

```
1. #define FD_VALID(fd) ((fd)>=0 && (fd)<COFFEE_FD_SET_SIZE && coffee_fd_set[(fd)].flags!=COFFEE_FD_FREE)
```

### (2)FD\_READABLE

FD\_READABLE 宏用于打开的文件有读权限, 源码如下:

```
1. #define FD_READABLE(fd) (coffee_fd_set[(fd)].flags & CFS_READ)
```

## 1.3 size 参数调整

如果文件欲读取的字节数(size)超过文件末尾, 则将读取字节数设为最大能读取的字节数(即最多只能读到文件末尾), 源码如下:

```
1. if (fdp->offset + size > file->end)  
2. {  
3.     size = file->end - fdp->offset;  
4. }
```

## 二、没有微日志的读

没有微日志情况下的读，略去参数验证及与本节无关代码如下：

```
1. struct file_desc *fdp;
2. struct file *file;
3.
4. fdp = &coffee_fd_set[fd];
5. file = fdp->file;
6.
7. if (!FILE_MODIFIED(file)) //判断该文件是否只是原始文件(意味着没有修改过，没有微日志文件)，详情见 2.1
8. {
9.     COFFEE_READ(buf, size, absolute_offset(file->page, fdp->offset)); //absolute_offset 见 2.2, COFFEE_READ 见 2.3
10.    fdp->offset += size; //更新偏移量
11.    return size; //返回实际读写字节数
12. }
```

### 2.1 FILE\_MODIFIED

file的flags只有两种取值：0 和COFFEE\_FILE\_MODIFIED，将物理文件缓存时(load\_file函数，详情见博文[《Contiki学习笔记：Coffee文件系统打开文件cfs\\_open》](#) 2.5)，如果物理文件元数据file\_header的flags的M位为 1 的话(即物理文件被修改，日志存在)，则file->flags设为COFFEE\_FILE\_MODIFIED，否则设为 0。部分代码如下：

```
1. //load_file 函数部分代码
2. file->flags = 0;
```

```
3.  
4. if (HDR_MODIFIED(*hdr))  
5. {  
6.     file->flags |= COFFEE_FILE_MODIFIED; //如果文件被修改(表示日志存在), #define COFFEE_FILE_MODIFIED 0x1  
7. }
```

再看下面的代码就很清楚了, 如果 `file->flags` 为 `COFFEE_FILE_MODIFIED`, 则返回真。

```
1. #define FILE_MODIFIED(file) ((file)->flags & COFFEE_FILE_MODIFIED)  
2.  
3. #define COFFEE_FILE_MODIFIED 0x1
```

## 2.2 absolute\_offset

**absolute\_offset** 函数返回文件系统的绝对偏移量(但不是物理偏移量), 得出的是从 `COFFEE_START` 处的偏移量, 而不是从 `FLASH` 开始外的偏移量, 因为 `COFFEE_START` 往往不等于 `FLASH_START` (因为会拿出 `FLASH` 一部分空间用于存放代码, 不受文件系统管理)。  
**absolute\_offset** 源码如下:

```
1. //absolute_offset(file->page, fdp->offset)  
2. static cfs_offset_t absolute_offset(coffee_page_t page, cfs_offset_t offset)  
3. {  
4.     return page * COFFEE_PAGE_SIZE + sizeof(struct file_header) + offset;  
5. }
```

## 2.3 COFFEE\_READ 宏

**COFFEE\_READ** 宏直接定位到硬件相关的读函数, 移植 `Coffee` 文件的时候需要映射过去(在 `cfs-coffee-arch.h`), 源码如下:

1. `#define COFFEE_READ(buf, size, offset) stm32_flash_read(COFFEE_START+offset, buf, size)`

COFFEE\_READ 与 stm32\_flash\_read 映射关系如下图，在原有的 offset 加上 COFFEE\_START，就得到了从 FLASH\_START 处的偏移量，即实际物理 FLASH 位置。

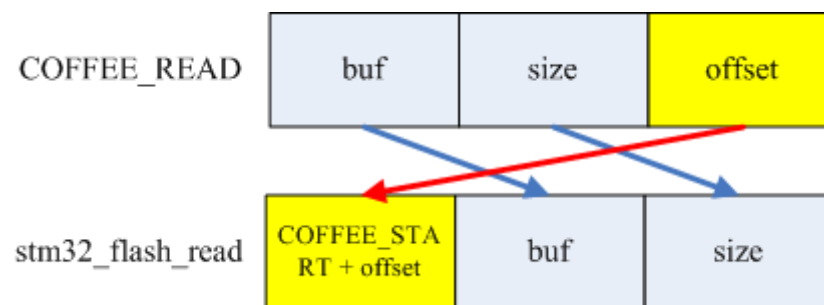


图 1 COFFEE\_READ 与 stm32\_flash\_read 映射关系

### 三、有微日志文件的读取

有微日志情况下的读，略去参数验证及与本节无关代码如下：

```
1. //int cfs_read(int fd, void *buf, unsigned size)
2. struct file_desc *fdp;
3. struct file *file;
4.
5. #if COFFEE_MICRO_LOGS //见 3.1
```



```
6.     struct file_header hdr;
7.     struct log_param lp; //见 3.2
8.     unsigned bytes_left;
9.     int r;
10. #endif
11.
12. fdp = &coffee_fd_set[fd];
13. file = fdp->file;
14.
15. #if COFFEE_MICRO_LOGS
16.     read_header(&hdr, file->page); //读取物理文件的元数据 file_header, 见 3.3
17.     /*如果微日志文件有数据, 直接从微日志文件读取数据, 否则从原始文件读取*/
18.     for (bytes_left = size; bytes_left > 0; bytes_left -= r) //读取日志记录, 当 size 跨越多个日志记录时, 则需要
        逐个记录读取
19.     {
20.         r = - 1;
21.         /*初始化结构体 log_param 类型的 lp*/
22.         lp.offset = fdp->offset;
23.         lp.buf = buf;
24.         lp.size = bytes_left;
25.
26.         r = read_log_page(&hdr, file->record_count, &lp); //见四
27.
28.         /*如果在微日志文件找不到数据, 就直接从原始文件读取*/
29.         if (r < 0)
30.         {
```

```
31.         COFFEE_READ(buf, lp.size, absolute_offset(file->page, fdp->offset)); //详情见 1.3
32.         r = lp.size;
33.     }
34.     fdp->offset += r;
35.     buf = (char*)buf + r;
36. }
37. #endif
38.
39. return size;
```

### 3.1 COFFEE\_MICRO\_LOGS

微日志是 Coffee 文件系统的一大亮点，当文件需要修改时，创建微日志结构并链接到原始文件，而不是创建新的文件。系统默认是配置微日志，源码如下：

```
1. #ifndef COFFEE_MICRO_LOGS
2.     #define COFFEE_MICRO_LOGS 1
3. #endif
```

不过也可以不配置微日志，在 cfs\_coffee\_arch.h 文件将 COFFEE\_MICRO\_LOGS 定义为 0，如下：

```
1. #define COFFEE_MICRO_LOGS 0
```

### 3.2 log\_param

在 Google 搜没找到 buggy compiler 相关资料，不晓得是什么东西。通过源码分析，当欲读取大小跨越两个日志记录(甚至更多)时，log\_param 用于辅助分开读取，即先读前一个日志记录包含的内容，再读取后一个日志记录包含的内容。源码如下：

```
1. /* This is needed because of a buggy compiler. */
2. struct log_param
3. {
4.     cfs_offset_t offset;
5.     const char *buf;
6.     uint16_t size;
7. };
```

### 3.3 read\_header

read\_header 将物理文件的元数据 file\_header 读出来放在 hdr 结构体，源代码如下：

```
1. static void read_header(struct file_header *hdr, coffee_page_t page)
2. {
3.     COFFEE_READ(hdr, sizeof(*hdr), page *COFFEE_PAGE_SIZE);
4.
5.     #if DEBUG
6.         if (HDR_ACTIVE(*hdr) && !HDR_VALID(*hdr))
7.         {
8.             PRINTF("Invalid header at page %u!\n", (unsigned)page);
9.         }
10.    #endif
11. }
```

## 四、read\_log\_page

read\_log\_page 将日志记录读到 lp->buffer 中, 返回实际读取的大小 lp->size。首先检查一些参(log\_record\_size、log\_records、search\_records), 必要时进行一些调整, 而后求得欲读取位置对应的索引表项, 进而求得偏移量, 最后调用 COFFEE\_READ 读取数据。源代码如下:

```
1. //r = read_log_page(&hdr, file->record_count, &lp);
2. #if COFFEE_MICRO_LOGS
3.     static int read_log_page(struct file_header *hdr, int16_t record_count, struct log_param *lp)
4.     {
5.         uint16_t region;
6.         int16_t match_index;
7.         uint16_t log_record_size;
8.         uint16_t log_records;
9.         cfs_offset_t base;
10.        uint16_t search_records;
11.
12.        adjust_log_config(hdr, &log_record_size, &log_records); //若 log_record_size 和 log_records 为 0, 则设成
        默认值, 详情见 4.1
13.        region = modify_log_buffer(log_record_size, &lp->offset, &lp->size); //设置 offset, 必要时调整 size, 返
        回 region, 见 4.2
14.
15.        search_records = record_count < 0 ? log_records : record_count; //需要搜索的记录数, record_count 是 file
        的成员变量
16.
17.        match_index = get_record_index(hdr->log_page, search_records, region); //见 4.3
18.        if (match_index < 0)
19.        {
```

```
20.         return - 1;
21.     }
22.
23.     /**求出文件欲读取的 Coffee 偏移量(即从 COFFEE_START 开始处的偏移量)，见 4.5***/
24.     base = absolute_offset(hdr->log_page, log_records * sizeof(region)); //索引表大小
25.     base += (cfs_offset_t)match_index * log_record_size; //跳过 match_index 个记录
26.     base += lp->offset;
27.
28.     COFFEE_READ(lp->buf, lp->size, base);
29.
30.     return lp->size;
31. }
32. #endif
```

#### 4.1 adjust\_log\_config

在博文《[Contiki学习笔记: Coffee文件组织及若干数据结构](#)》分析结构体file\_header成员变量log\_record\_size, 说如果该项为0, 则会被设成默认值(即COFFEE\_PAGE\_SIZE), 就是在这里设置的。log\_records表示日志可以容纳的记录数量(log records denotes the number of records that the log can hold), 如果该项为0, 则设成(COFFEE\_LOG\_SIZE)/(\*log\_record\_size)。

```
1. //adjust_log_config(hdr, &log_record_size, &log_records);
2. #if COFFEE_MICRO_LOGS
3.     static void adjust_log_config(struct file_header *hdr, uint16_t *log_record_size, uint16_t *log_records)
4.     {
5.         *log_record_size = hdr->log_record_size == 0 ? COFFEE_PAGE_SIZE : hdr->log_record_size;
6.         *log_records = hdr->log_records == 0 ? COFFEE_LOG_SIZE / *log_record_size : hdr->log_records;
```

7. }
8. #endif

## 4.2 modify\_log\_buffer

modify\_log\_buffer 将偏移量(全局的)调整为日志记录内部的偏移量, 如果欲读取的大小跨越两个日志记录(甚至更多), 则需调整 size 为从 offset 到第一日志记录末尾, 最后返回 region。modify\_log\_buffer 函数示意图如下:

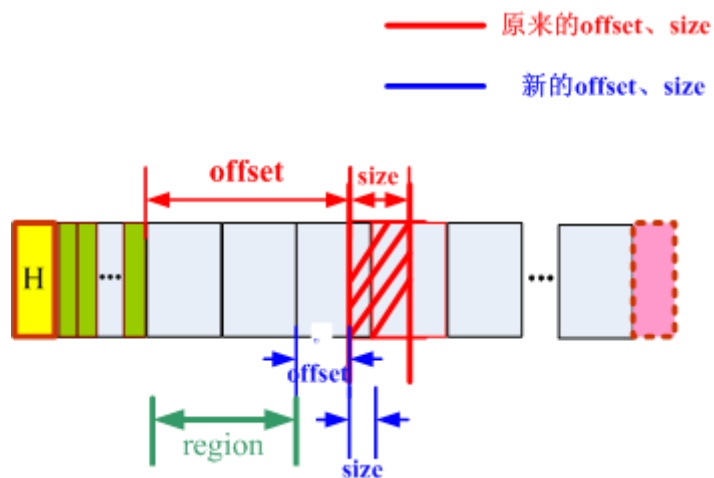


图 2 modify\_log\_buffer 函数示意图

modify\_log\_buffer 源代码如下:

1. //region = modify\_log\_buffer(log\_record\_size, &lp->offset, &lp->size), 其中 lp.offset 被初始化为 fdp->offset
2. #if COFFEE\_MICRO\_LOGS

```
3. static uint16_t modify_log_buffer(uint16_t log_record_size, cfs_offset_t *offset, uint16_t *size)
4. {
5.     uint16_t region;
6.
7.     region = *offset / log_record_size; //算出从第几个 region 开始
8.     *offset %= log_record_size;
9.
10.    if (*size > log_record_size - *offset) //当 size 跨越两个(甚至更多)日志记录时, 需调整 size
11.    {
12.        *size = log_record_size - *offset;
13.    }
14.
15.    return region;
16. }
17. #endif
```

### 4.3 log\_records & record\_count & search\_records

log\_records 是 file\_header 的成员变量, 表示日志可以容纳的记录数量(log records denotes the number of records that the log can hold)。如果为 0, 会自动调节成 COFFEE\_LOG\_SIZE/\*log\_record\_size, 而 log\_record\_size 若为 0, 则会 自动调成 COFFEE\_PAGE\_SIZE, 这些是在 adjust\_log\_config 函数完成的(见 4.1)。

record\_count 是 file 的成员变量, 表示文件实际的日志记录数量, 加载文件 load\_file 将 record-count 设为-1(源码注释-We don't know the amount of records yet)。

search\_records 是 read\_log\_page 函数定义的一个成员变量, 表示需要搜索的记录数, 若已有 record\_count 值, 则用 record\_count 初始化 search\_records, 否则用 log\_records, 源码如下:

1. `//static int read_log_page(struct file_header *hdr, int16_t record_count, struct log_param *lp)`
2. `search_records = record_count < 0 ? log_records : record_count;`

#### 4.4 get\_record\_index

get\_record\_index 求得 region 日志记录对应的索引表项，源代码如下：

1. `//match_index = get_record_index(hdr->log_page, search_records, region);`
2. `#if COFFEE_MICRO_LOGS`
3. `static int get_record_index(coffee_page_t log_page, uint16_t search_records, uint16_t region)`
4. `{`
5.  `cfs_offset_t base;`
6.  `uint16_t processed;`
7.  `uint16_t batch_size;`
8.  `int16_t match_index, i;`
9.
10.  `base = absolute_offset(log_page, sizeof(uint16_t) * search_records); //返回索引表最后一个字节处，见 4.4.1`
11.  `batch_size = search_records > COFFEE_LOG_TABLE_LIMIT ? COFFEE_LOG_TABLE_LIMIT : search_records; //调整`  
`search_records, 不能超过 COFFEE_LOG_TABLE_LIMIT`
12.  `processed = 0; //正在读取的记录`
13.  `match_index = - 1;`
14.
15. `{`
16.  `uint16_t indices[batch_size]; //存放索引表`
17.
18.  `while (processed < search_records && match_index < 0)`



```
19.     {
20.         if (batch_size + processed > search_records) //调整 batch_size 大小, 不能超过 search_records
21.         {
22.             batch_size = search_records - processed;
23.         }
24.
25.         base -= batch_size * sizeof(indices[0]); //base 从索引表末尾移到索引表开始处, 见 4.4.1
26.         COFFEE_READ(&indices, sizeof(indices[0]) * batch_size, base); //读取索引表放入 indices[batch_size],
    见 4.4.1
27.
28.         for (i = batch_size - 1; i >= 0; i--) //索引表从后向前扫描
29.         {
30.             /**索引表项正好指向 region 记录(即找到索引表项), 返回 match_index***/
31.             if (indices[i] - 1 == region)
32.             {
33.                 match_index = search_records - processed - (batch_size - i);
34.                 break;
35.             }
36.         }
37.
38.         processed += batch_size;
39.     }
40. }
41.
42. return match_index; //没找到 region 目录对应的索引项, 则返回-1
43. }
```

44. `#endif`

#### 4.4.1 读取索引表

通过 `absolute_offset` 求得索引表末尾位置 `base` (如下图红色箭头所示), 再通过 `base -= batch_size * sizeof(indices[0])` 求得索引表开始位置 (如下图绿色箭头所示), 最后通过 `COFFEE_READ` 将索引表读入 `indices[]` 数组。整个过程示意图如下:

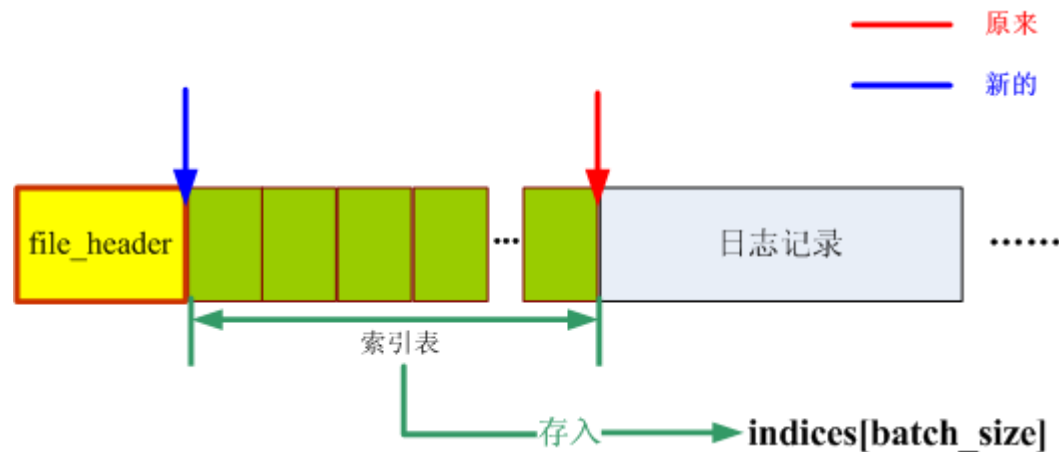


图 3 读取索引表示意图

#### 4.5 read\_log\_page 中求偏移量

得到文件欲读取的索引表项后, 接下来, 求得偏移量 (从 `COFFEE_START` 开始处的偏移量), 结合源码和下图理解, 很快就会懂的, 如下:

```
1. base = absolute_offset(hdr->log_page, log_records * sizeof(region)); //索引表大小
```

2. `base += (cfs_offset_t)match_index * log_record_size; //跳过 match_index 个记录`
3. `base += lp->offset;`

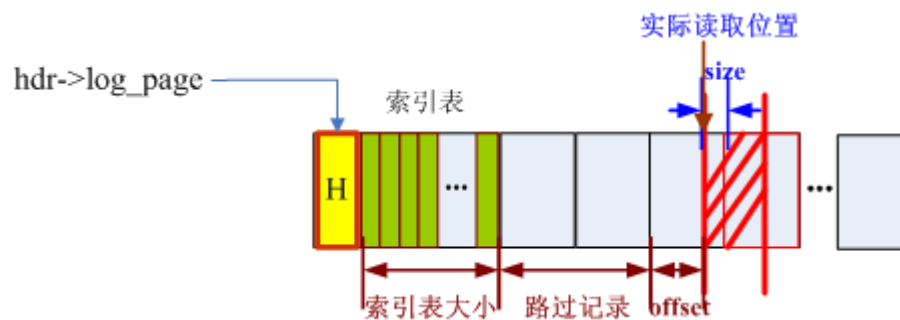


图 4 read\_log\_page 求偏移量示意图

更多Contiki学习笔记可通过博文《[Contiki学习笔记：目录](#)》索引访问。

上述图 1-4 源文件如下：



[get\\_record\\_index函数示意图.rar](#)



[modify\\_log\\_buffer函数示意图.rar](#)



[read\\_log\\_page求偏移量示意图.rar](#)



[COFFEE\\_READ与stm32 flash read映射关系.rar](#)

[Contiki学习笔记: Coffee文件系统写入文件cfs\\_write](#) (2011-12-24 14:09)

标签: [Coffee](#) [Contiki](#) [cfs\\_write](#) [写入](#) [学习笔记](#) 分类: [Contiki](#)

### 摘要:

本文深入源码分析 Coffee 文件系统写入文件 cfs\_write 技术细节, 包括 FD\_WRITABLE、COFFEE\_IO\_SEMANTICS、merge\_log、write\_log\_page、find\_next\_record、create\_log、COFFEE\_APPEND\_ONLY、COFFEE\_WRITE。

### 一、cfs\_write

源代码如下:

```
1. int cfs_write(int fd, const void *buf, unsigned size)
2. {
3.     struct file_desc *fdp;
```

```
4.     struct file *file;
5.
6.     #if COFFEE_MICRO_LOGS
7.         int i;
8.         struct log_param lp;
9.         cfs_offset_t bytes_left;
10.        const char dummy[1] =
11.        {
12.            0xff
13.        };
14.    #endif
15.
16.    if(!(FD_VALID(fd) && FD_WRITABLE(fd))) //fd 有效性判断及检查写权限, 详情见 1.1
17.    {
18.        return - 1;
19.    }
20.
21.    fdp = &coffee_fd_set[fd];
22.    file = fdp->file;
23.
24.
25.    /**若超过文件末尾写, 则扩展文件(当没有设置 CFS_COFFEE_IO_FIRM_SIZE 时), 见二***/
26.    #if COFFEE_IO_SEMANTICS
27.        if(!(fdp->io_flags & CFS_COFFEE_IO_FIRM_SIZE))
28.        {
29.            #endif
```

```
30.
31.     while(size + fdp->offset + sizeof(struct file_header) > (file->max_pages *COFFEE_PAGE_SIZE))
32.     {
33.         if(merge_log(file->page, 1) < 0)
34.         {
35.             return - 1;
36.         } file = fdp->file;
37.         PRINTF("Extended the file at page %u\n", (unsigned)file->page);
38.     }
39.
40.     #if COFFEE_IO_SEMANTICS
41.     }
42. #endif
43.
44. /*****写入文件概述，见 1.2*****/
45. /*****实际写入文件*****/
46. #if COFFEE_MICRO_LOGS
47.     #if COFFEE_IO_SEMANTICS
48.         if(!(fdp->io_flags&CFS_COFFEE_IO_FLASH_AWARE) && (FILE_MODIFIED(file) || fdp->offset<file->end))
49.         #else
50.             if(FILE_MODIFIED(file) || fdp->offset < file->end)
51.             {
52.                 #endif
53.                 for(bytes_left = size; bytes_left > 0;)
54.                 {
55.                     lp.offset = fdp->offset;
```

```
56.         lp.buf = buf;
57.         lp.size = bytes_left;
58.         i = write_log_page(file, &lp);
59.         if(i < 0)
60.         {
61.             if(size == bytes_left)
62.             {
63.                 return - 1;
64.             }
65.             break;
66.         }
67.         else if(i == 0)
68.         {
69.             file = fdp->file;
70.         }
71.         else
72.         {
73.             bytes_left -= i;
74.             fdp->offset += i;
75.             buf = (char*)buf + i;
76.
77.             if(fdp->offset > file->end)
78.             {
79.                 file->end = fdp->offset;
80.             }
81.         }
```

```
82.         }
83.         if(fdp->offset > file->end)
84.         {
85.             COFFEE_WRITE(dummy, 1, absolute_offset(file->page, fdp->offset));
86.         }
87.     }
88.     else
89.     {
90. #endif /* COFFEE_MICRO_LOGS */
91.
92.         #if COFFEE_APPEND_ONLY
93.             if(fdp->offset < file->end)
94.             {
95.                 return - 1;
96.             }
97. #endif
98.
99.             COFFEE_WRITE(buf, size, absolute_offset(file->page, fdp->offset));
100.             fdp->offset += size;
101.
102.         #if COFFEE_MICRO_LOGS
103.         }
104. #endif
105.
106.         if(fdp->offset > file->end)
107.         {
```



```
108.         file->end = fdp->offset;
109.     }
110.
111.     return size;
112. }
```

## 1.1 宏 FD\_VALID 和 FD\_WRITABLE

### (1) FD\_VALID

FD\_VALID 宏用于判断 cfs\_read 函数传进来的 fd 是否有效, 不能小于 0 (如, -1 是 get\_available\_fd 函数分配不到 fd 的返回值), 也不能大于 FD 的上限 (即 COFFEE\_FD\_SET\_SIZE), 其对应的 file\_desc 的标志 flags 不能为空闲 (COFFEE\_FD\_FREE), 源码如下:

```
1. #define FD_VALID(fd) ((fd) >= 0 && (fd) < COFFEE_FD_SET_SIZE && coffee_fd_set[(fd)].flags != COFFEE_FD_FREE)
```

### (2) FD\_WRITABLE

FD\_WRITABLE 判断该文件是否以 CFS\_WRITE 打开, 可见**如果想从文件末尾写, 则需同时指定 CFS\_WRITE 和 CFS\_APPEND**, 源码如下:

```
1. #define FD_WRITABLE(fd) (coffee_fd_set[(fd)].flags & CFS_WRITE)
```

注: 尽管 Coffee 官方论文[1]声称指定了 CFS\_APPEND 意味着指向 CFS\_WRITE, 从源码分析, 显然必须得指定 CFS\_WRITE 才能写入。我觉得把 FD\_WRITABLE 改下, 会更符合编程习惯, 如下:

```
1. #define FD_WRITABLE(fd) (coffee_fd_set[(fd)].flags & CFS_WRITE & CFS_APPEND)
```

## 1.2 写入文件概述

扩展文件后(必要时, 见二), 就实际进入写阶段了, 出现了很多预定义的宏, 先理清下这些宏的层次关系, 而后再详细分析各种情况, 简化后源代码如下:

```
1.  /*****写入文件概述, 见 1.2*****/
2.  #if COFFEE_MICRO_LOGS
3.      #if COFFEE_IO_SEMANTICS
4.          if(!(fdp->io_flags & CFS_COFFEE_IO_FLASH_AWARE) && (FILE_MODIFIED(file) || fdp->offset < file->end))
5.              {
6.          #else
7.              if(FILE_MODIFIED(file) || fdp->offset < file->end)
8.                  {
9.          #endif
10.             /*****情型 1, 见三*****/
11.             for(bytes_left = size; bytes_left > 0;)
12.                 {
13.                 }
14.             }
15.             else /*****情型 2, 见四*****/
16.                 {
17. #endif
18.
19. #if COFFEE_MICRO_LOGS
20.             }
21. #endif
```

```
22.  
23.  /***情型 3, 见五***/  
24.  if(fdp->offset > file->end)  
25.  {  
26.      file->end = fdp->offset;  
27.  }
```

如果借助代码缩进(我用 SourceFormat 格式化代码, 不是很智能, 还得手动调)还没看清层次关系, 那继续看下图:

```

/*****写入文件概述，见1.2*****/
#if COFFEE_MICRO_LOGS
  #if COFFEE_IO_SEMANTICS
    if(!(fdp->io_flags & CFS_COFFEE_IO_FLASH_AWARE) && (FILE_MODIFIED(file) ||
    #else
    if(FILE_MODIFIED(file) || fdp->offset < file->end)
    #endif
    /*****情型1，见三*****/
    for(bytes_left = size; bytes_left > 0;)
    {
    }
    else /*****情型2，见四*****/
    #endif
  #if COFFEE_MICRO_LOGS
  #endif

  /****情型3，见五****/
  if(fdp->offset > file->end)
  {
    file->end = fdp->offset;
  }

```

The diagram illustrates the macro configuration logic for writing to a file. It shows a series of nested conditional compilation directives (#if, #else, #endif) and a loop. Annotations in Chinese and arrows indicate the flow of execution and specific cases (情型1, 情型2, 情型3) mentioned in the code comments. A vertical green line on the left marks the start of the main logic block.

图1 宏配置示意图

搞清了宏层次关系，就不难理解写入文件都有哪些情况：

### (1) 情形 1

配置了 COFFEE\_MICRO\_LOGS 且文件要么被修改了(即微日志文件存在)或者文件偏移量小于 end(即原始文件还有空间可以写入), 如果 Coffee 还配置了 COFFEE\_IO\_SEMANTICS, 还要求 file\_desc 的 io\_flags 中 CFS\_COFFEE\_IO\_FLASH\_AWARE 没有设置。

### (2) 情形 2

配置了 COFFEE\_MICRO\_LOGS, 文件要么没有被修改(即微日志文件不存在)且文件偏移量大于等于 end, 如果 Coffee 还配置了 COFFEE\_IO\_SEMANTICS, 还要求 file\_desc 的 io\_flags 中 CFS\_COFFEE\_IO\_FLASH\_AWARE 被设置。

若文件被扩展了(见二), 此时 file->end 等于偏移量, file 的 flags 中 M 位为 0(即没有微日志文件), 这种情况恰好满足这个条件(CFS\_COFFEE\_IO\_FLASH\_AWARE 有设置的话)。

### (3) 情形 3

没有配置 COFFEE\_MICRO\_LOGS, 就直接跳到情况三了。事实上, 情型 1 和情型 2 都需要执行这段代码(也有可能中途就退出了)。

## 1.3 一个 BUG?

假设系统配置了 COFFEE\_IO\_SEMANTICS 且 io\_flags 也设置了 CFS\_COFFEE\_IO\_FIRM\_SIZE, 此时 if(! (fdp->io\_flags & CFS\_COFFEE\_IO\_FIRM\_SIZE)) 为假, 直到跳到 #if COFFEE\_MICRO\_LOGS。倘若系统没有配置 COFFEE\_MICRO\_LOGS, 再次跳过, 执行以下语句, 不论 offset 与 end 关系如何, 直接返回 size。也就是说, 在这种最简单的模型下, 传给 cfs\_write 的 size, 原封不动返回。而这种情况是有可能存在的, 这难道不是一个 BUG?

```
1. if(fdp->offset > file->end)
2. {
3.     file->end = fdp->offset;
```

```
4. }  
5.  
6. return size;
```

## 二、COFFEE\_IO\_SEMANTICS 与 CFS\_COFFEE\_IO\_FIRM\_SIZE

如果没有设置 CFS\_COFFEE\_IO\_FIRM\_SIZE，当现有的空间不足以数据写入时，则需扩展文件(merge\_log 函数)，略去参数验证及与本节无关代码如下：

```
1. //设置了 COFFEE_IO_SEMANTICS  
2. struct file_desc *fdp;  
3. struct file *file;  
4.  
5. fdp = &coffee_fd_set[fd];  
6. file = fdp->file;  
7.  
8. #if COFFEE_IO_SEMANTICS //见 2.1  
9.     if(!(fdp->io_flags&CFS_COFFEE_IO_FIRM_SIZE))//若 io_flags 没设置 CFS_COFFEE_IO_FIRM_SIZE 则返回真，见 2.1  
10.    {  
11.#endif  
12.  
13.    while(size + fdp->offset + sizeof(struct file_header) > (file->max_pages *COFFEE_PAGE_SIZE)) //若待写入的数  
    据超过文件末尾，则合并日志  
14.    {  
15.        if(merge_log(file->page, 1) < 0) //见 2.2
```

```
16.     {
17.         return - 1;
18.     }
19.     file = fdp->file;
20.     PRINTF("Extended the file at page %u\n", (unsigned)file->page);
21. }
22.
23. #if COFFEE_IO_SEMANTICS
24. }
25. #endif
```

## 2.1 COFFEE\_IO\_SEMANTICS

如 果定义了 COFFEE\_IO\_SEMANTICS, 则在 file\_desc 结构体会多一个成员变量 io\_flags, 配置了 COFFEE\_IO\_SEMANTICS 可以优化某些存储设备的文件访问(optimize file access on certain storage types), 系统默认没有配置 COFFEE\_IO\_SEMANTICS。系统定义了 io\_flags 两个值, 即 CFS\_COFFEE\_IO\_FLASH\_AWARE 和 CFS\_COFFEE\_IO\_FIRM\_SIZE。设置了 CFS\_COFFEE\_IO\_FIRM\_SIZE, 如果写入文件超过预留的大小, Coffee 不会继续扩展文件。当文件有固定大小限制时, 设置 CFS\_COFFEE\_IO\_FIRM\_SIZE 可以保护过度写。

在这里, 如果 io\_flags 设置了 CFS\_COFFEE\_IO\_FIRM\_SIZE, 就没必要扩展文件(即使是超过了)。如果连 COFFEE\_IO\_SEMANTICS 都没定义, 也就是说 file 压根就没有 io\_flags, 那就更省事了:-)

## 2.2 merge\_log

merge\_log 用于合并日志, 即当文件剩余空间不足以写入 size 字节时, 需要扩展, 具体做法是: 将原始文件和微日志文件(如果有的话)拷贝到新文件。源代码如下:

```
1. //merge_log(file->page, 1)
```

```
2. static int merge_log(coffee_page_t file_page, int extend)
3. {
4.     struct file_header hdr, hdr2;
5.     int fd, n;
6.     cfs_offset_t offset;
7.     coffee_page_t max_pages;
8.     struct file *new_file;
9.     int i;
10.
11.     read_header(&hdr, file_page);
12.
13.     fd = cfs_open(hdr.name, CFS_READ); //以只读方式打开文件
14.     if(fd < 0)
15.     {
16.         return - 1;
17.     }
18.
19.     /**创建新的文件***/
20.     max_pages = hdr.max_pages << extend;
21.     new_file = reserve(hdr.name, max_pages, 1, 0); //创建新文件的主体函数
22.     if(new_file == NULL)
23.     {
24.         cfs_close(fd);
25.         return - 1;
26.     }
27.
```



```
28.     offset = 0;
29.
30.     /**将原文件的数据内容拷贝到新文件 new_file***/
31.     do
32.     {
33.         char buf[hdr.log_record_size == 0 ? COFFEE_PAGE_SIZE : hdr.log_record_size]; //数组表达式不是常量，编译
出错
34.
35.         n = cfs_read(fd, buf, sizeof(buf));
36.         if(n < 0)
37.         {
38.             remove_by_page(new_file->page, !REMOVE_LOG, !CLOSE_FDS, ALLOW_GC);
39.             cfs_close(fd);
40.             return - 1;
41.         }
42.         else if(n > 0)
43.         {
44.             COFFEE_WRITE(buf, n, absolute_offset(new_file->page, offset));
45.             offset += n;
46.         }
47.     }while(n != 0);
48.
49.     /**利用原来那个 file_desc***/
50.     for(i = 0; i < COFFEE_FD_SET_SIZE; i++)
51.     {
52.         if(coffee_fd_set[i].flags != COFFEE_FD_FREE && coffee_fd_set[i].file->page == file_page)
```

```
53.     {
54.         coffee_fd_set[i].file = new_file;
55.         new_file->references++;
56.     }
57. }
58.
59. /**删除原文件**/
60. if(remove_by_page(file_page, REMOVE_LOG, !CLOSE_FDS, !ALLOW_GC) < 0) //删除原始文件和微日志文件、不关闭 FD、
    不进行垃圾回收
61. {
62.     remove_by_page(new_file->page, !REMOVE_LOG, !CLOSE_FDS, !ALLOW_GC);
63.     cfs_close(fd);
64.     return - 1;
65. }
66.
67. /**将原来 file_header 的 log_record_size 和 log_records 拷贝到新的 file_header**/
68. read_header(&hdr2, new_file->page);
69. hdr2.log_record_size = hdr.log_record_size;
70. hdr2.log_records = hdr.log_records;
71. write_header(&hdr2, new_file->page);
72.
73. /**设置新文件 new_file 的 flags、end**/
74. new_file->flags &= ~COFFEE_FILE_MODIFIED;
75. new_file->end = offset;
76.
77. cfs_close(fd);
```

```
78.  
79.     return 0;  
80. }
```

reserve 是创建新文件的主体函数，首先进行参数验证，接着查看物理FLASH是否有连续pages空闲页，若有，则返回该文件即将占有页的第一页。否则，调用 Coffee垃圾回收，再次查看物理FLASH是否有连续pages空闲页，如果还没有就返回NULL。创建成功后，加载文件(load\_file)，如果缓存失败也是返回NULL。详情见博文《[Contiki学习笔记：Coffee文件系统创建文件](#)》。

### 三、写入文件情形 1

配置了 COFFEE\_MICRO\_LOGS 且文件要么被修改了(即微日志文件存在)或者文件偏移量小于 end(即原始文件还有空间可以写入)，如果 Coffee 还配置了 COFFEE\_IO\_SEMANTICS，还要求 file\_desc 的 io\_flags 中 CFS\_COFFEE\_IO\_FLASH\_AWARE 没有设置。略去参数验证及与本节无关代码如下：

```
1. struct file_desc *fdp;  
2. struct file *file;  
3.  
4. #if COFFEE_MICRO_LOGS  
5.     int i;  
6.     struct log_param lp;  
7.     cfs_offset_t bytes_left;  
8.     const char dummy[1] =  
9.     {  
10.         0xff  
11.     };
```

```
12. #endif
13.
14. fdp = &coffee_fd_set[fd];
15. file = fdp->file;
16.
17. {
18.     for(bytes_left = size; bytes_left > 0;)
19.     {
20.         /**初始化 lp 结构体***/
21.         lp.offset = fdp->offset;
22.         lp.buf = buf;
23.         lp.size = bytes_left;
24.
25.         i = write_log_page(file, &lp); //见 3.1
26.         if(i < 0)
27.         {
28.             if(size == bytes_left)
29.             {
30.                 return - 1;
31.             }
32.             break;
33.         }
34.         else if(i == 0)
35.         {
36.             file = fdp->file;
37.         }
```

```
38.     else
39.     {
40.         bytes_left -= i;
41.         fdp->offset += i;
42.         buf = (char*)buf + i;
43.
44.         if(fdp->offset > file->end)
45.         {
46.             file->end = fdp->offset;
47.         }
48.     }
49. }
50. if(fdp->offset > file->end)
51. {
52.     COFFEE_WRITE(dummy, 1, absolute_offset(file->page, fdp->offset)); //向文件 offset 处写入字节"0xFF"
53. }
54. }
55.
56. /*****/
57. if(fdp->offset > file->end)
58. {
59.     file->end = fdp->offset;
60. }
61.
62. return size;
```

### 3.1 write\_log\_page

write\_log\_page 将内容写入日志，源代码如下：

```
1. #if COFFEE_MICRO_LOGS
2.     static int write_log_page(struct file *file, struct log_param *lp)
3.     {
4.         struct file_header hdr;
5.         uint16_t region;
6.         coffee_page_t log_page;
7.         int16_t log_record;
8.         uint16_t log_record_size;
9.         uint16_t log_records;
10.        cfs_offset_t offset;
11.        struct log_param lp_out;
12.
13.        read_header(&hdr, file->page);
14.
15.        adjust_log_config(&hdr, &log_record_size, &log_records);
16.        region = modify_log_buffer(log_record_size, &lp->offset, &lp->size);
17.
18.        log_page = 0;
19.        if(HDR_MODIFIED(hdr))
20.            //判断微日志文件是否存在
21.            {
22.                /* A log structure has already been created. */
```

```
23.     log_page = hdr.log_page;
24.     log_record = find_next_record(file, log_page, log_records); //见 3.2
25.     if(log_record >= log_records)
26.     {
27.         /* The log is full; merge the log. */
28.         PRINTF("Coffee: Merging the file %s with its log\n", hdr.name);
29.         return merge_log(file->page, 0);
30.     }
31. }
32. else
33. {
34.     /**创建微日志文件, 见 3.3***/
35.     log_page = create_log(file, &hdr);
36.     if(log_page == INVALID_PAGE)
37.     {
38.         return - 1;
39.     }
40.     PRINTF("Coffee: Created a log structure for file %s at page %u\n", hdr.name, (unsigned)log_page);
41.     hdr.log_page = log_page;
42.     log_record = 0;
43. }
44.
45. {
46.     char copy_buf[log_record_size];
47.
48.     lp_out.offset = offset = region * log_record_size;
```

```
49.     lp_out.buf = copy_buf;
50.     lp_out.size = log_record_size;
51.
52.     if((lp->offset>0 || lp->size != log_record_size) && read_log_page(&hdr, log_record, &lp_out) < 0)
53.     {
54.         COFFEE_READ(copy_buf, sizeof(copy_buf), absolute_offset(file->page, offset));
55.     }
56.
57.     memcpy(&copy_buf[lp->offset], lp->buf, lp->size);
58.
59.     /*
60.      * Write the region number in the region index table.
61.      * The region number is incremented to avoid values of zero.
62.      */
63.     offset = absolute_offset(log_page, 0);
64.     ++region;
65.     COFFEE_WRITE(&region, sizeof(region), offset + log_record * sizeof(region));
66.
67.     offset += log_records * sizeof(region);
68.     COFFEE_WRITE(copy_buf, sizeof(copy_buf), offset + log_record * log_record_size);
69.     file->record_count = + 1;
70. }
71.
72. return lp->size;
73. }
74. #endif
```



### 3.2 find\_next\_record

find\_next\_record 源代码如下:

```
1. //log_record = find_next_record(file, log_page, log_records);
2. #if COFFEE_MICRO_LOGS
3.     static int find_next_record(struct file *file, coffee_page_t log_page, int log_records)
4.     {
5.         int log_record, preferred_batch_size;
6.
7.         if(file->record_count >= 0)
8.         {
9.             return file->record_count;
10.        }
11.
12.        preferred_batch_size = log_records > COFFEE_LOG_TABLE_LIMIT ? COFFEE_LOG_TABLE_LIMIT: log_records;
13.        {
14.            /* The next log record is unknown at this point; search for it. */
15.            uint16_t indices[preferred_batch_size];
16.            uint16_t processed;
17.            uint16_t batch_size;
18.
19.            log_record = log_records;
20.            for(processed = 0; processed < log_records; processed += batch_size)
21.            {
```

```
22.         batch_size = log_records - processed >= preferred_batch_size ? preferred_batch_size: log_records
23.         - processed;
24.         COFFEE_READ(&indices, batch_size * sizeof(indices[0]), absolute_offset(log_page, processed
25.         * sizeof(indices[0])));
26.         for(log_record = 0; log_record < batch_size; log_record++)
27.         {
28.             if(indices[log_record] == 0)
29.             {
30.                 log_record += processed;
31.                 break;
32.             }
33.         }
34.     }
35.
36.     return log_record;
37. }
38. #endif
```

### 3.3 创建微日志文件 create\_log

create\_log 首先调整日志记录大小和日志记录数量，reserve 创建并加载微日志文件，若成功，对 file\_header 及 file 相关成员变量进行一些设置，返回微日志文件的第一页，否则返回 INVALID\_PAGE。创建成功后的文件总体示意图如下：

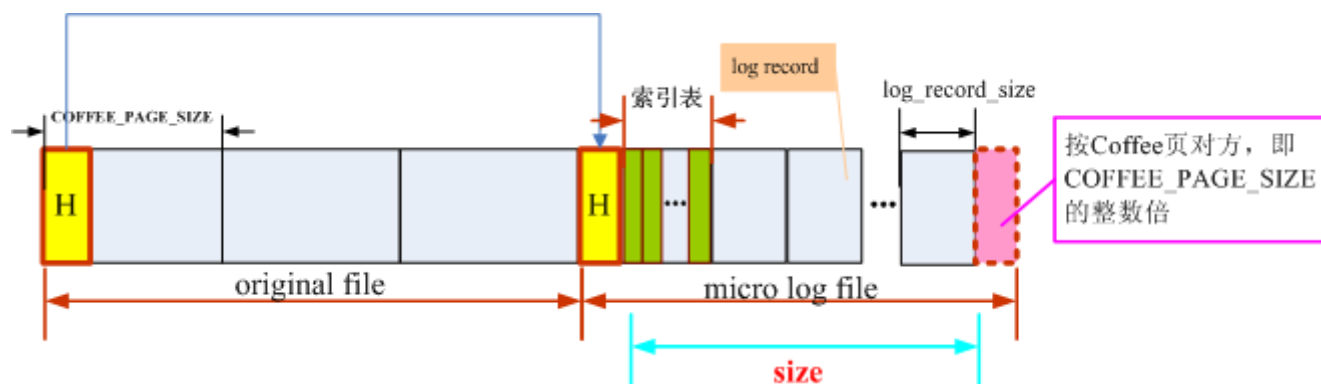


图 3 创建微日志文件示意图

创建微日志文件 create\_log 源代码如下：

```

1. //log_page = create_log(file, &hdr);
2. #if COFFEE_MICRO_LOGS
3.     static coffee_page_t create_log(struct file *file, struct file_header *hdr)
4.     {
5.         uint16_t log_record_size, log_records;
6.         cfs_offset_t size;
7.         struct file *log_file;
8.
9.         adjust_log_config(hdr, &log_record_size, &log_records); //调整微日志配置
10.
11.         /*****创建微日志文件*****/

```

```
12.     size = log_records *(sizeof(uint16_t) + log_record_size); //Log index size+log data size
13.     log_file = reserve(hdr->name, page_count(size), 1, HDR_FLAG_LOG);
14.     if (log_file == NULL)
15.     {
16.         return INVALID_PAGE;
17.     }
18.
19.     hdr->flags |= HDR_FLAG_MODIFIED; //将 file_header 的 flags 中 M 位置 1, 表示文件已修改, 微日志文件存在
20.     hdr->log_page = log_file->page; //将 file_header 的 log_page 指向微日志文件的第一页
21.     write_header(hdr, file->page); //写入 file_header
22.
23.     file->flags |= COFFEE_FILE_MODIFIED; //file_header 的 flag 中 M 位为 1(即物理文件被修改, 日志存在), 则
    file->flags 设为 COFFEE_FILE_MODIFIED
24.     return log_file->page;
25. }
26. #endif
```

adjust\_log\_config 调整日志记录大小和日志记录数量, 即如果 log\_record\_size 及 log\_records 为 0, 则设成默认值, 详情参见博文《Contiki 学习笔记: Coffee 文件系统读取文件 cfs\_read》。

reserve 是创建微日志文件的主体函数, 首先进行参数验证, 接着查看物理FLASH是否有连续pages空闲页, 若有, 则返回该文件即将占有页的第一页。否则, 调用 Coffee垃圾回收, 再次查看物理FLASH是否有连续pages空闲页, 如果还没有就返回NULL。并加载文件load\_file, 若成功, 返回 file指针, 否则返回NULL。详情参见博文《[Contiki学习笔记: Coffee文件系统创建文件](#)》三。

#### 四、写入文件情形 2

配置了 COFFEE\_MICRO\_LOGS，文件要么没有被修改(即微日志文件不存在)且文件偏移量大于等于 end，如果 Coffee 还配置了 COFFEE\_IO\_SEMANTICS，还要求 file\_desc 的 io\_flags 中 CFS\_COFFEE\_IO\_FLASH\_AWARE 被设置。

若文件被扩展了(见二)，此时 file->end 等于偏移量，file 的 flags 中 M 位为 0(即没有微日志文件)，这种情况恰好满足这个条件(CFS\_COFFEE\_IO\_FLASH\_AWARE 有设置的话)。

当系统配置了 COFFEE\_APPEND\_ONLY，即只允许文件末尾写入，确保偏移量在文件末尾之后，就直接写入。略去参数验证及与本节无关代码如下：

```
1. struct file_desc *fdp;
2. struct file *file;

1. #if COFFEE_MICRO_LOGS
2.     int i;
3.     struct log_param lp;
4.     cfs_offset_t bytes_left;
5.     const char dummy[1] =
6.     {
7.         0xff
8.     };
9. #endif

1.
2. fdp = &coffee_fd_set[fd];
3. file = fdp->file;
4.
5. #if COFFEE_APPEND_ONLY //见 4.1
```

```
6.     if(fdp->offset < file->end)
7.     {
8.         return - 1;
9.     }
10. #endif
11.
12. COFFEE_WRITE(buf, size, absolute_offset(file->page, fdp->offset)); //此时 offset >= end, 见 4.2
13. fdp->offset += size;
14.

/*****/

1. if(fdp->offset > file->end)
2. {
3.     file->end = fdp->offset;
4. }
5. return size;
```

#### 4.1 COFFEE\_APPEND\_ONLY

COFFEE\_APPEND\_ONLY 用于指定文件写入只能从文件末尾写，系统默认是没有设置的，若设置了，可以节省代码空间 (save some code space)。值得注意的是，COFFEE\_APPEND\_ONLY 和 COFFEE\_MICRO\_LOGS 不能同时为 1，否则编译错误，源码如下：

```
1. #if COFFEE_MICRO_LOGS && COFFEE_APPEND_ONLY
2.     #error "Cannot have COFFEE_APPEND_ONLY set when COFFEE_MICRO_LOGS is set."
3. #endif
```

## 4.2 COFFEE\_WRITE

COFFEE\_WRITE 宏直接定位到硬件相关的读函数，移植 Coffee 文件的时候需要映射过去(在 cfs-coffee-arch.h)，源码如下：

```
1. #define COFFEE_WRITE(buf, size, offset) stm32_flash_write(COFFEE_START + offset, buf, size)
```

COFFEE\_WRITE 与 stm32\_flash\_write 映射关系如下图，在原有的 offset 加上 COFFEE\_START，就得到了从 FLASH\_START 处的偏移量，即实际物理 FLASH 位置。

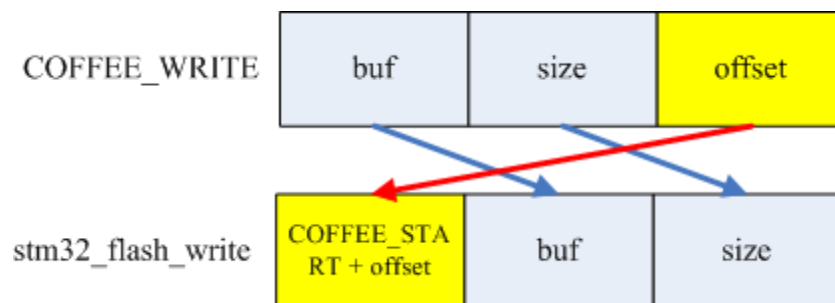


图 COFFEE\_WRITE 与 stm32\_flash\_write 映射关系

我觉得好奇怪，既然已经超过了文件末尾，那应该是从 `file->end` 写入，否则会产生文件空洞。

## 五、写入文件情形 3

没有配置 `COFFEE_MICRO_LOGS`，就直接跳到情形 3 了，事实上，情形 1 和情形 2 都需要执行这段代码，

略去参数验证及与本节无关代码如下：

```
1. struct file_desc *fdp;
2. struct file *file;
3.
4. fdp = &coffee_fd_set[fd];
5. file = fdp->file;
6.
7. if(fdp->offset > file->end)
8. {
9.     file->end = fdp->offset;
10.}
11. return size;
```

更多Contiki学习笔记可通过博文《[Contiki学习笔记：目录](#)》索引访问。

本文图片 1~2 源文件如下：



[COFFEE\\_WRITE与stm32 flash\\_write映射关系.rar](#)





[创建微日志示意图.rar](#)

### 参考资料:

[1] Tsiftes Nicolas, Dunkels Adam, He Zhitao. Enabling large-scale storage in sensor networks with the coffee file system[J]. International Conference on Information Processing in Sensor Networks. 2009, 349-360

[2] [File Systems - ContikiWiki](#)

[3] Contiki 源代码

[Contiki学习笔记: Coffee文件系统关闭文件cfs\\_close](#) (2011-12-14 16:39)

标签: [Coffee](#) [Contiki](#) [cfs\\_close](#) [关闭](#) [学习笔记](#) 分类: [Contiki](#)

### 摘要:

本文深入源码讲述了 Coffee 文件系统关闭文件 cfs\_close 函数。

当文件不使用时，需要关闭，这样可以释放文件占有的内部资源(如 fd)，腾出缓存可他用(possibly commit any cached data to permanent storage)[1]，源代码如下：

```
1. void cfs_close(int fd)
2. {
3.     if (FD_VALID(fd))
4.     {
5.         coffee_fd_set[fd].flags = COFFEE_FD_FREE;
6.         coffee_fd_set[fd].file->references--;
7.         coffee_fd_set[fd].file = NULL;
8.     }
9. }
10.
11. #define FD_VALID(fd) ((fd) >= 0 && (fd) < COFFEE_FD_SET_SIZE && coffee_fd_set[(fd)].flags != COFFEE_FD_FREE)
```

首先进行 fd 参数验证，确保 fd 有效。如果 fd 不在 [0, COFFEE\_FD\_SET\_SIZE) 范围内，说明传递的 fd 参数无效，如果该 fd 对应的 file\_desc 的 flags 是 COFFEE\_FD\_FREE，则说明该 fd 本来就是空闲的，无需再释放。如果 fd 有效，则将其对应的 file\_desc 的 flags 设为 COFFEE\_FD\_FREE，以便下次 cfs\_open 函数可以找到可用的 fd，而后将与该 file\_desc 关联文件 file 的引用次数减 1，最后将 file\_desc 指向的 file 设为空。

用宏 FILE\_UNREFERENCED 可以判断文件引用次数是否为 0，这个宏在 load\_file 函数被调用，FILE\_UNREFERENCED 宏定义如下：

```
1. #define FILE_UNREFERENCED(file) ((file)->references == 0)
```

另，我觉得这个函数设计不够合理。cfs\_close 应该要有返回值(fd 失效返回-1，执行成功返回 0)，增强程序的健壮性。

更多Contiki学习笔记可通过博文《[Contiki学习笔记：目录](#)》索引访问。

参考资料：

[1] [File Systems - ContikiWiki](#)

[Contiki学习笔记：Coffee文件系统删除文件cfs\\_remove](#) (2011-12-23 16:16)

标签： [Coffee](#) [Contiki](#) [cfs\\_remove](#) [删除](#) [学习笔记](#) 分类： [Contiki](#)

**摘要：**

本文深入源码分析了 Coffee 文件删除文件 cfs\_remove 技术细节，包括 remove\_by\_page、COFFEE\_EXTENDED\_WEAR\_LEVELLING。

## 一、cfs\_remove

Coffee 文件系统删除文件 cfs\_remove，删除成功返回 0，否则返回-1。cfs\_remove 首先找到 name 对应的文件 file 指针 (find\_file 函数)，而后调用 remove\_by\_page 删除文件，源码如下：

```
1. int cfs_remove(const char *name)
2. {
3.     struct file *file;
```

```
4.
5.     file = find_file(name);
6.     if(file == NULL)
7.     {
8.         return - 1;
9.     }
10.
11.     return remove_by_page(file->page, REMOVE_LOG, CLOSE_FDS, ALLOW_GC);
12. }
```

find\_file 函数用于找到文件名name对应的file指针。若name对应的文件file还驻留在内存(即还在 coffee\_files[COFFEE\_MAX\_OPEN\_FILES]数组里), 并且对应的物理文件是有效的, 则直接返回file指针, 否则扫描整个 FLASH, 将name 对应文件file(在FLASH中但没缓存)缓存到内存(这一步得确保coffee\_files数组有可用的项, 否则返回空 NULL, 参见 load\_file函数)。详情见博文《[Contiki学习笔记: Coffee文件系统打开文件cfs\\_open](#)》二。这个有点纳闷了, 删除文件, 事先得把file\_header缓存。

## 二、remove\_by\_page

remove\_by\_page, 删除微日志文件(如果有的话), 将文件头 file\_header 的 flags 中的 0 位(isolated, 孤立)置 1, 更新 file\_header, 将该 文件关联的 file\_desc 的 flags 设为 COFFEE\_FD\_FREE 和初始化 file 结构体, 必要时 (COFFEE\_EXTENDED\_WEAR\_LEVELLING 为 0 且 gc\_allowed 为 1)进行垃圾回收。

值得一提的是, 通过将文件头 file\_header 的 flags 中的 0 位置 1, 虽表示文件已删除, 但此时, 文件占用的空间还不能被使用(还没擦写), 只有垃圾回收处理后才能使用, 而垃圾回收只有当存储空间不足时才执行。源码如下:

```
1. //return remove_by_page(file->page, REMOVE_LOG, CLOSE_FDS, ALLOW_GC);
```

```
2. static int remove_by_page(coffee_page_t page, int remove_log, int close_fds, int gc_allowed) //2.1
3. {
4.     struct file_header hdr;
5.     int i;
6.
7.     read_header(&hdr, page); //read_header 用于读取物理文件的元数据(即 file_header)
8.     if (!HDR_ACTIVE(hdr)) //若 A 为 1, 0 为 0, I 为 0, 则 HDR_ACTIVE 返回真
9.     {
10.         return - 1;
11.     }
12.
13.     /***删除微日志文件***/
14.     if (remove_log && HDR_MODIFIED(hdr))
15.     {
16.         if (remove_by_page(hdr.log_page, !REMOVE_LOG, !CLOSE_FDS, !ALLOW_GC) < 0)
17.         {
18.             return - 1;
19.         }
20.     }
21.
22.     hdr.flags |= HDR_FLAG_OBSOLETE; //标志为失效页
23.     write_header(&hdr, page); //更新物理的 file_header
24.     *gc_wait = 0;
25.
26.     /***将待删除文件相关联的 file_desc 中 flags 设为 COFFEE_FD_FREE(一个文件可能被多次打开, 如多线程)***/
27.     if (close_fds)
```

```
28.  {
29.     for (i = 0; i < COFFEE_FD_SET_SIZE; i++)
30.     {
31.         if (coffee_fd_set[i].file != NULL && coffee_fd_set[i].file->page == page)
32.         {
33.             coffee_fd_set[i].flags = COFFEE_FD_FREE;
34.         }
35.     }
36. }
37.
38. /**将待删除文件相关联的 file 初始化, 变为可用***/
39. for (i = 0; i < COFFEE_MAX_OPEN_FILES; i++)
40. {
41.     if (coffee_files[i].page == page)
42.     {
43.         coffee_files[i].page = INVALID_PAGE;
44.         coffee_files[i].references = 0;
45.         coffee_files[i].max_pages = 0;
46.     }
47. }
48.
49. /**没有配置 COFFEE_EXTENDED_WEAR_LEVELLING 且 gc_allowed 为 1, 则进行垃圾回收, 见 2.2***/
50. #if !COFFEE_EXTENDED_WEAR_LEVELLING
51.     if (gc_allowed)
52.     {
53.         collect_garbage(GC_RELUCTANT);
```

```
54.     }
55. #endif
56.
57.     return 0; //邮件成功返回 0
58. }
```

## 2.1 remove\_by\_page 参数

删除文件 cfs\_remove 传给 remove\_by\_page 参数分别是 file->page、REMOVE\_LOG、CLOSE\_FDS、ALLOW\_GC，如下：

```
1. //return remove_by_page(file->page, REMOVE_LOG, CLOSE_FDS, ALLOW_GC);
2.
3. #define REMOVE_LOG 1 //删除微日志文件
4. #define CLOSE_FDS 1 //关闭 FD，事实上是将 file_desc 的 flags 设为 COFFEE_FD_FREE
5. #define ALLOW_GC 1 //允许垃圾回收，得确保 COFFEE_EXTENDED_WEAR_LEVELLING 为 0，才会进行垃圾回收
```

## 2.2 COFFEE\_EXTENDED\_WEAR\_LEVELLING

Coffee 文件系统默认是配置了 COFFEE\_EXTENDED\_WEAR\_LEVELLING，源码如下。在本例，文件删除并没有立即进行垃圾回收，而是待到没有空间可用的时候再回收(可理解成批处理)，显然这样做有一个明显的缺点，垃圾回收会占用比较长的时间。

```
1. #ifndef COFFEE_EXTENDED_WEAR_LEVELLING
2.     #define COFFEE_EXTENDED_WEAR_LEVELLING 1
3. #endif
```

系统提供了两种垃圾回收机制：GC\_GREEDY 和 GC\_RELUCTANT，前者垃圾回收过程中，擦除尽可能多的区(即贪心回收)，后者擦除一个区后就停止，删除文件采用的是后一种。两种回收机制源代码如下：

1. `/* "Greedy" garbage collection erases as many sectors as possible. */`
2. `#define GC_GREEDY 0`
3. `/* "Reluctant" garbage collection stops after erasing one sector. */`
4. `#define GC_RELUCTANT 1`

### 参考资料:

[1] Tsiftes Nicolas, Dunkels Adam, He Zhitao. Enabling large-scale storage in sensor networks with the coffee file system[J]. International Conference on Information Processing in Sensor Networks. 2009, 349-360

[2] [File Systems - ContikiWiki](#)

[3] Contiki 源代码

[Contiki学习笔记: Coffee文件系统垃圾回收collect\\_garbage](#) (2012-01-07 15:21)

标签: [Coffee](#) [Contiki](#) [collect\\_garbage](#) [垃圾回收](#) [学习笔记](#) 分类: [Contiki](#)

### 摘要:

本文深入源码分析了 Coffee 文件系统垃圾回收 `collect_garbage` 技术细节, 包括回收模式 `mode`、`sector_status`、`COFFEE_SECTOR_COUNT`、`get_sector_status`、`isolate_pages` 等。



## 一、collect\_garbage

### 1.1 垃圾回收概述

FLASH 先擦后写的特性，也就不可能每次把失效的页及时擦除(效率低下。有些页可能还没失效，则需要转移这些页的数据，才能擦除)，只能在适当时候才擦除，这就是 垃圾回收。Coffee 文件系统默认情况下，文件删除并没有立即进行垃圾回收，而是待到没有空间可用的时候再回收(可理解成批处理)，显然这样做有一个明显的缺点，垃圾回收会占用比较长的时间。collect\_garbage 源代码如下：

```
1. static void collect_garbage(int mode) //mode 见 1.2
2. {
3.     uint16_t sector;
4.     struct sector_status stats; //sector_status 结构体用于统计垃圾回收页面情况，见 1.3
5.     coffee_page_t first_page, isolation_count;
6.
7.     PRINTF("Coffee: Running the file system garbage collector in %s mode\n", mode == GC_RELUCTANT ? "reluctant" :
            "greedy");
8.
9.     /*The garbage collector erases as many sectors as possible. A sector is erasable if there are only free or obsolete
       pages in it. */
10.
11.    for(sector = 0; sector < COFFEE_SECTOR_COUNT; sector++) //COFFEE_SECTOR_COUNT 见 1.4
12.    {
13.        isolation_count = get_sector_status(sector, &stats); //见二
14.
```

```
15.     PRINTF("Coffee: Sector %u has %u active, %u obsolete, and %u free pages. \n", sector, (unsigned)stats.active,
    (unsigned)stats.obsolete, (unsigned)stats.free);
16.
17.     if(stats.active > 0)
18.     {
19.         continue;
20.     }
21.
22.     if((mode == GC_RELUCTANT && stats.free == 0) || (mode == GC_GREEDY && stats.obsolete > 0))
23.     {
24.         first_page = sector * COFFEE_PAGES_PER_SECTOR;
25.         if(first_page < *next_free)
26.         {
27.             *next_free = first_page;
28.         }
29.
30.         if(isolation_count > 0) //找到 isolation_count 个连续的孤立页
31.         {
32.             isolate_pages(first_page + COFFEE_PAGES_PER_SECTOR, isolation_count); //见 1.5
33.         }
34.
35.         COFFEE_ERASE(sector);
36.         PRINTF("Coffee: Erased sector %d!\n", sector);
37.
38.         if(mode == GC_RELUCTANT && isolation_count > 0) //如果是 GC_RELUCTANT 回收策略，一旦有逻辑分区被擦除
    就停止
```

```
39.         {  
40.             break;  
41.         }  
42.     }  
43. }  
44. }
```

## 1.2 回收模式 mode

系统提供了两种垃圾回收机制：GC\_GREEDY 和 GC\_RELUCTANT，前者垃圾回收过程中，擦除尽可能多的区(即贪心回收)，后者擦除一个区后就停止。当创建文件或者创建日志，找不到可用空间时(reserve 函数)，就会用贪心回收 GC\_GREEDY。而删除文件采用的是后一种(前提是 COFFEE\_EXTENDED\_WEAR\_LEVELLING 为 0 且 gc\_allowed 为 1)。两种回收机制源代码如下：

1. `#define GC_GREEDY 0`
2. `#define GC_RELUCTANT 1`

## 1.3 sector\_status

sector\_status 结构体用于垃圾回收的统计页面情况，如下：

```
1. //struct sector_status stats;  
2. struct sector_status  
3. {  
4.     coffee_page_t active;  
5.     coffee_page_t obsolete;  
6.     coffee_page_t free;  
7. };
```

## 1.4 COFFEE\_SECTOR\_COUNT

Coffee 文件系统是按逻辑区擦除的，之所以有这个是为了应付大的存储设备(比如 SD 卡)，在这种情况下，将其设置大一点可加快顺序扫描速度。COFFEE\_SECTOR\_COUNT 宏定义如下：

```
1. #define COFFEE_SECTOR_COUNT (unsigned) (COFFEE_SIZE/COFFEE_SECTOR_SIZE)
```

## 1.5 isolate\_pages

isolate\_pages 源码如下：

```
1. //isolate_pages(first_page + COFFEE_PAGES_PER_SECTOR, isolation_count);
2. static void isolate_pages(coffee_page_t start, coffee_page_t skip_pages)
3. {
4.     struct file_header hdr;
5.     coffee_page_t page;
6.
7.     /*Split an obsolete file starting in the previous sector and mark the following pages as isolated.*/
8.
9.     /**将 file_header 的 flags 中的 A 位、I 位置 1，其他位为 0***/
10.    memset(&hdr, 0, sizeof(hdr));
11.    hdr.flags = HDR_FLAG_ALLOCATED | HDR_FLAG_ISOLATED;
12.
13.    /*Isolation starts from the next sector.*/
14.    for(page = 0; page < skip_pages; page++)
15.    {
16.        write_header(&hdr, start + page);
```

```
17.     }
18.     PRINTF("Coffee: Isolated %u pages starting in sector %d\n", (unsigned) skip_pages,
19.     (int) start/COFFEE_PAGES_PER_SECTOR);
19. }
```

## 二、get\_sector\_status

get\_sector\_status 源代码如下:

```
1. //isolation_count = get_sector_status(sector, &stats);
2. static coffee_page_t get_sector_status(uint16_t sector, struct sector_status *stats)
3. {
4.     static coffee_page_t skip_pages;
5.     static char last_pages_are_active;
6.     struct file_header hdr;
7.     coffee_page_t active, obsolete, free;
8.     coffee_page_t sector_start, sector_end;
9.     coffee_page_t page;
10.
11.     memset(stats, 0, sizeof(*stats));
12.     active = obsolete = free = 0;
13.
14.     /*get_sector_status() is an iterative function using local static state. It therefore requires the the caller
15.     loops starts from sector 0 in order to reset the internal state.*/
15.     if(sector == 0)
```

```
16.  {
17.      skip_pages = 0;
18.      last_pages_are_active = 0;
19.  }
20.
21.  sector_start = sector * COFFEE_PAGES_PER_SECTOR;
22.  sector_end = sector_start + COFFEE_PAGES_PER_SECTOR;
23.
24.  /*Account for pages belonging to a file starting in a previous segment that extends into this segment. If the
    whole segment is covered, we do not need to continue counting pages in this iteration.*/
25.  if(last_pages_are_active)
26.  {
27.      if(skip_pages >= COFFEE_PAGES_PER_SECTOR)
28.      {
29.          stats->active = COFFEE_PAGES_PER_SECTOR;
30.          skip_pages -= COFFEE_PAGES_PER_SECTOR;
31.          return 0;
32.      }
33.      active = skip_pages;
34.  }
35.  else
36.  {
37.      if(skip_pages >= COFFEE_PAGES_PER_SECTOR)
38.      {
39.          stats->obsolete = COFFEE_PAGES_PER_SECTOR;
40.          skip_pages -= COFFEE_PAGES_PER_SECTOR;
```

```
41.         return skip_pages >= COFFEE_PAGES_PER_SECTOR ? 0 : skip_pages;
42.     }
43.     obsolete = skip_pages;
44. }
45.
46. /*Determine the amount of pages of each type that have not been accounted for yet in the current sector.*/
47. for(page = sector_start + skip_pages; page < sector_end;)
48. {
49.     read_header(&hdr, page);
50.     last_pages_are_active = 0;
51.     if(HDR_ACTIVE(hdr))
52.     {
53.         last_pages_are_active = 1;
54.         page += hdr.max_pages;
55.         active += hdr.max_pages;
56.     }
57.     else if(HDR_ISOLATED(hdr))
58.     {
59.         page++;
60.         obsolete++;
61.     }
62.     else if(HDR_OBSOLETE(hdr))
63.     {
64.         page += hdr.max_pages;
65.         obsolete += hdr.max_pages;
66.     }
```

```
67.     else
68.     {
69.         free = sector_end - page;
70.         break;
71.     }
72. }
73.
74. /*Determine the amount of pages in the following sectors that should be remembered for the next iteration. This
    is necessary because no page except the first of a file contains information about what type of page it is. A side
    effect of remembering this amount is that there is no need to read in the headers of each of these pages from the
    storage.*/
75. skip_pages = active + obsolete + free - COFFEE_PAGES_PER_SECTOR;
76. if(skip_pages > 0)
77. {
78.     if(last_pages_are_active)
79.     {
80.         active = COFFEE_PAGES_PER_SECTOR - obsolete;
81.     }
82.     else
83.     {
84.         obsolete = COFFEE_PAGES_PER_SECTOR - active;
85.     }
86. }
87.
88. stats->active = active;
89. stats->obsolete = obsolete;
```



```
90.     stats->free = free;
91.
92.     /*To avoid unnecessary page isolation, we notify the callee that "skip_pages" pages should be isolated only
    if the current file extent ends in the next sector. If the file extent ends in a more distant sector, however, the
    garbage collection can free the next sector immediately without requiring page isolation.*/
93.     return (last_pages_are_active || (skip_pages >= COFFEE_PAGES_PER_SECTOR)) ? 0: skip_pages;
94. }
```

## contiki 代码学习之一：浅探 protothread 进程控制模型【1】

以 contiki 2.4 中最简单的 example/hello\_world.c 为例。程序的代码如下：

```
1. /* Declare the process */
2. PROCESS(hello_world_process, "Hello world");
3. /* Make the process start when the module is loaded */
4. AUTOSTART_PROCESSES(&hello_world);
5.
6. /* Define the process code */
7. PROCESS_THREAD(hello_world_process, ev, data) {
8.     PROCESS_BEGIN();                /* Must always come first */
9.     printf("Hello, world!\n");      /* Initialization code goes here */
10.    while(1) {                      /* Loop for ever */
11.        PROCESS_WAIT_EVENT();        /* Wait for something to happen */
12.    }
13.    PROCESS_END();                  /* Must always come last */
14. }
```

### 复制代码

A process in Contiki consists of a single protothread.

protothread 是 contiki 的进程控制模型，是 contiki 关于进程的精华知识，详情可参考 [contiki protothreads](#)。

其它关于 contiki 的资料也可参考 [The Contiki Operating System](#)，里面介绍了关于 Contiki 的 process、protothread、timer，以及 communication stack 的一些内容，归纳和总结了里面的一些库函数，这些其实在 contiki 的源码中都可以找到。

contiki 程序有着非常规范的程序步骤,PROCESS 宏是用来声明一个进程;AUTOSTART\_PROCESS 宏是使这个进程自启动,PROCESS\_THREAD 里面定义的是程序的主体,并且主体内部要以 PROCESS\_BEGIN() 开头,以 PROCESS\_END() 来结束。

接下来一个一个宏展开来看清它们的真面目:

### 1. PROCESS

PROCESS 宏定义在 core/sys 的 process.h 文件内, 如下:

```
1. #define PROCESS(name, strname) \
2.     PROCESS_THREAD(name, ev, data); \
3.     struct process name = { NULL, strname, \
4.                             process_thread_##name }
```

*复制代码*

PROCESS\_THREAD 的宏定义如下, 它用来定义一个 process 的 body:

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

*复制代码*

最终又回归到 PT\_THREAD, 所以说 A process in Contiki consists of a single protothread.

看看 PT\_THREAD 宏中参数的定义:

```
1. struct pt {
2.     lc_t lc;
3. };//typedef unsigned short lc_t;
4.
```

```
5. typedef unsigned char process_event_t;  
6.  
7. typedef void *           process_data_t;
```

### 复制代码

第一个参数 lc 是英文全称是 local continuation (本地延续? , 这个不好翻译), 它可以说就是 protothread 的控制参数, 因为 protothread 的精华在 C 的 switch 控制语句, 这个 lc 就是 switch 里面的参数;

第二个参数就是 timer 给这个进程传递的事件了, 其实就是一个 unsigned char 类型的参数, 具体的参数值在 process .h 中定义;

第三个参数也是给进程传递的一个参数 (举个例子, 假如要退出一个进程的话, 那么必须把这个进程所对应的 timer 也要从 timer 队列 timerlist 清除掉, 那么进程在死掉之前就要给 etimer\_process 传递这个参数, 参数的值就是进程本身, 这样才能是 etimer\_process 在 timer 队列里找到相应的 etimer 进而清除)。

PT\_THREAD 是用来定义一个 protothread, 每个 protothread 都要用到这个宏, 其宏定义如下:

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

### 复制代码

所以 PROCESS(hello\_world\_process, "Hello world process"); 的展开代码如下:

```
1. static char process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data);  
2. struct process hello_world_process = { ((void *)0), "Hello world process", process_thread_hello_world_process};
```

### 复制代码

第一个语句是声明了一个 process\_thread\_hello\_world\_process 的函数, 返回值是 char 型, 其实就是声明了 hello\_world 这个 process 的 body 函数, 接下来的 PROCESS\_THREAD 宏其实就是这个函数的定义。

第二个语句是声明了一个 process 类型的数据, 是 hello\_world\_process。下面看看 struct process 是怎么定义的。

```
1. struct process {
2.     struct process *next;
3.     const char *name;
4.     PT_THREAD((* thread) (struct pt *, process_event_t, process_data_t));
5.     struct pt pt;
6.     unsigned char state, needspoll;
7. };
```

### 复制代码

1. 因为要维持一个进程的链表, struct process \*next 是指向它的下一个进程, 声明一个 process 时初始化为 0;
2. const char \*name 指的是这个进程的名字;
3. 第三个展开来就是 char (\* thread) (struct pt \*, process\_event\_t, process\_data\_t), thread 是一个函数指针, 也就是说 process 包含了一个函数, 每个 process 都包含了一个函数, process 的执行其实就是这个函数体的执行。
4. 再来看看 pt 的定义:

```
1. struct pt {
2.     lc_t lc;    //typedef unsigned short lc_t;
3. };
```

### 复制代码

struct pt pt 在这有何用, 这正是 protothread 的精髓所在, 看 contiki 内在 ls-switch.h 文件里的一段说明:

This implementation of local continuations uses the C switch() statement to resume execution of a function somewhere inside the function's body. The implementation is based on the fact that switch() statements are able to jump directly into the bodies of control structures such as if() or while() statements.

上面已经提到过 lc 表示 local continuation, 它是为 switch() 语句服务的, 保留 process 的 pt 值, 在下一次 process 执行的时候便

可通过 switch 语句 直接跳到要执行的语句，从而可以恢复 process 的执行，实现对 process 的控制。

5. 还有 unsigned char state, needspoll, state 表示 process 的状态，具体定义在 process.h 文件中；一般 process 初始化的时候 needspoll 的值为 0，假如 needspoll 的值为 1 的话，可以优先执行，具体先不深入讨论。

## 2. AUTOSTART\_PROCESSES 先看看 AUTOSTART\_PROCESS 的宏定义：

1. #define AUTOSTART\_PROCESSES(...) \
2. struct process \* const autostart\_processes[] = {\_\_VA\_ARGS\_\_, NULL}

### 复制代码

autostart\_process 是一个指针数组，每个指针指向的都是一个 process。  
所以展开的代码就是：

1. struct process \* const autostart\_processes[] = {&hello\_world\_process, ((void \*)0)};

### 复制代码

在 main 函数中的进程自启动函数 autostart\_start() 函数中参数进程就是这个指针数组，从这里取 process 一依次执行。

## 3. PROCESS\_THREAD

process 的 body 就是定义在 PROCESS\_THREAD 中，可见 PROCESS\_THREAD 的重要性，其实展开来它就是一个函数，上面已经提到了，展开如下：

1. static char process\_thread\_hello\_world\_process(struct pt \*process\_pt, \
2. process\_event\_t ev, \
3. process\_data\_t data)
4. {
5. }

复制代码

process 的 body 就定义在 { } 中。

#### 4. PROCESS\_BEGIN 和 PROCESS\_END

process 的 body 必须要以这两个宏来作为开始和结尾，看看这连个宏定义：

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

复制代码

所以展开来就是：

```
1. {
2.     char PT_YIELD_FLAG = 1;
3.     switch((process_pt)->lc)
4.     {
5.         case 0:
```

复制代码

再看看 PROCESS\_END 的宏定义：

```
1. #define PROCESS_END()          PT_END(process_pt)
2.
3. #define PT_END(pt) LC_END((pt)->lc); PT_YIELD_FLAG = 0;    \
4.                     PT_INIT(pt); return PT_ENDED; }
5.
6. #define LC_END(s) }
```

```
7.  
8. #define PT_INIT(pt)      LC_INIT((pt)->lc)  
9.  
10. #define LC_INIT(s) s = 0;  
11.  
12. #define PT_ENDED        3
```

复制代码

展开来就是:

```
1. };  
2. PT_YIELD_FLAG = 0;  
3. (process_pt)->lc = 0;;  
4. return 3;  
5. }
```

复制代码

## 5. 总体展开的代码如下

```
1. struct process hello_world_process = { ((void *)0), "Hello world process", process_thread_hello_world_process};  
2. struct process * const autostart_processes[] = {&hello_world_process, ((void *)0)};  
3.  
4. static char process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data)  
5. {  
6.     {  
7.         char PT_YIELD_FLAG = 1;
```



```
8.      switch((process_pt)->lc)
9.      {
10.         case 0:
11.             ;
12.             printf( "Hello, world!\n" );    /* Initialization code goes here */
13.             while(1)
14.             {
15.                 /* Wait for an event. */
16.                 //PROCESS_WAIT_EVENT();
17.                 do
18.                 {
19.                     PT_YIELD_FLAG = 0;
20.                     (process_pt)->lc = __LINE__; case __LINE__:    ;
21.                     if(PT_YIELD_FLAG == 0)
22.                     {
23.                         return 1;
24.                     }
25.                 } while(0);
26.                 /* Got the timer's event~ */
27.             }
28.         }
29.
30.     };
31.     PT_YIELD_FLAG = 0;
32.     (process_pt)->lc = 0;;
33.     return 3;
```

```
34.     }  
35. }
```

### 复制代码

这段代码有的细节实在是让人困惑，原来代码段{ }后可以加分号，所以 switch 后也可加分号，而且一条语句之后也可以用多个分号。

这是一段用几个宏包装后的代码，其实是非常简单的几个变量声明和函数定义，首先声明了一个进程函数 process\_thread\_hello\_world\_process，然后再声明了一个进程 hello\_world\_process，并且这个进程的 函数指针指向的就是前一个进程函数，接着再声明一个自启动进程列表供 main 函数调用，最后再定义进程函数 process\_thread\_hello\_world\_process，当进程执行的时候，执行的就是 process\_thread\_hello\_world\_process 函数。

由于字数有限，暂且到这里，进程的具体执行过程请看下一部分。

## contiki 代码学习之一：浅探 protothread 进程控制模型【2】

要知道 hello\_world 进程是怎么被执行的，得看看 main 函数，找到 platform 文件夹下的对应的平台，这里选的是 mb851 下的 main 函数。要执行 hello\_world.c 的 process，主要是用到 main 函数的这个 autostart\_start 函数：

```
1. autostart_start(autostart_processes);
```

*复制代码*

接下来看看这个函数的定义：

```
1. void
2. autostart_start(struct process * const processes[])
3. {
4.     int i;
5.
6.     for(i = 0; processes[i] != NULL; ++i) {
7.         process_start(processes[i], NULL);
8.         PRINTF("autostart_start: starting process '%s' \n", processes[i]->name);
9.     }
10. }
```

*复制代码*

struct process \* const process[] 维护的是一个进程队列，它是一个 extern 变量，如果编译的是 hello\_world，那么这个外部变量当然引用的就是 hello\_world.c 文件中的 process[] 变量了。然后轮流执行队列里面的 process，此例中只有一个 hello\_world 进程，当然 还可以同时定义多个进程。要启动一个进程，必须要用到 process\_start 函数，此函数的定义如下：

```
1. void
2. process_start(struct process *p, const char *arg)
3. {
4.     struct process *q;
5.
6.     /* First make sure that we don't try to start a process that is
7.        already running. */
8.     for(q = process_list; q != p && q != NULL; q = q->next);
9.
10.    /* If we found the process on the process list, we bail out. */
11.    if(q == p) {
12.        return;
13.    }
14.    /* Put on the procs list.*/
15.    p->next = process_list;
16.    process_list = p;
17.    p->state = PROCESS_STATE_RUNNING;
18.    PT_INIT(&p->pt);
19.
20.    PRINTF("process: starting '%s' \n", p->name);
21.
22.    /* Post a synchronous initialization event to the process. */
23.    process_post_synch(p, PROCESS_EVENT_INIT, (process_data_t)arg);
24. }
```

复制代码

process\_start 所干的工作就是：1. 先把将要执行的进程加入到进程队列 process\_list 的首部，如果这个进程已经在 process\_list 中，就 return；  
2. 接下来就把 state 设置为 PROCESS\_STATE\_RUNNING 并且初始化 pt 的 lc 为 0（这一步初始化很重要，关系到 protothread 进程模型的理解）。  
3. 最后通过函数 process\_post\_synch() 给进程传递一个 PROCESS\_EVENT\_INIT 事件，让其开始执行，看看这个函数的定义：

```
1. void
2. process_post_synch(struct process *p, process_event_t ev, process_data_t data)
3. {
4.     struct process *caller = process_current;
5.
6.     call_process(p, ev, data);
7.     process_current = caller;
8. }
```

### 复制代码

为什么 process\_post\_synch() 中要把 process\_current 保存起来呢，process\_current 指向的是一个正在运行 的 process，当调用 call\_process 执行这个 hello\_world 这个进程时，process\_current 就会指向当前的 process 也就是 hello\_world 这个进程，而 hello\_world 这个进程它可能会退出或者正在被挂起等待一个事件，这时 process\_current = caller 语句正是要恢复先前的那个正在运行的 process。

接下来展开 call\_process()，开始真正的执行这个 process 了：

```
1. static void
2. call_process(struct process *p, process_event_t ev, process_data_t data)
```

```
3. {
4.     int ret;
5.
6. #if DEBUG
7.     if(p->state == PROCESS_STATE_CALLED) {
8.         printf("process: process '%s' called again with event %d\n", p->name, ev);
9.     }
10. #endif /* DEBUG */
11.
12.     if((p->state & PROCESS_STATE_RUNNING) &&
13.         p->thread != NULL) {
14.         PRINTF("process: calling process '%s' with event %d\n", p->name, ev);
15.         process_current = p;
16.         p->state = PROCESS_STATE_CALLED;
17.         ret = p->thread(&p->pt, ev, data);
18.         if(ret == PT_EXITED ||
19.            ret == PT_ENDED ||
20.            ev == PROCESS_EVENT_EXIT) {
21.             exit_process(p, p);
22.         } else {
23.             p->state = PROCESS_STATE_RUNNING;
24.         }
25.     }
26. }
```

复制代码

这个函数中才是真正的执行了 `hello_world_process` 的内容。

假如进程 `process` 的状态是 `PROCESS_STATE_RUNNING` 以及进程中的 `thread` 函数不为空的话，就执行这个进程：

1. 首先把 `process_current` 指向 `p`;
2. 接着把 `process` 的 `state` 改为 `PROCESS_STATE_CALLED`;
3. 执行 `hello_world` 这个进程的 `body` 也就是函数 `p->thread`，并将返回值保存在 `ret` 中，如果返回值表示退出或者遇到了 `PROCESS_EVENT_EXIT` 的时事件后，便执行 `exit_process()` 函数，`process` 退出。不然程序就应该在挂起等待事件的状态，那么就继续把 `p` 的状态维持为 `PROCESS_STATE_RUNNING`。

最后稍微看一下 `exit_process()` 函数：

```
1. static void
2. exit_process(struct process *p, struct process *fromprocess)
3. {
4.     register struct process *q;
5.     struct process *old_current = process_current;
6.
7.     PRINTF("process: exit_process '%s' \n", p->name);
8.
9.     if(process_is_running(p)) {
10.         /* Process was running */
11.         p->state = PROCESS_STATE_NONE;
12.
13.         /*
14.          * Post a synchronous event to all processes to inform them that
15.          * this process is about to exit. This will allow services to
```

```
16.      * deallocate state associated with this process.
17.      */
18.      for(q = process_list; q != NULL; q = q->next) {
19.          if(p != q) {
20.
21. call_process(q, PROCESS_EVENT_EXITED, (process_data_t)p);
22.          }
23.      }
24.
25.      if(p->thread != NULL && p != fromprocess) {
26.          /* Post the exit event to the process that is about to exit. */
27.          process_current = p;
28.          p->thread(&p->pt, PROCESS_EVENT_EXIT, NULL);
29.      }
30.  }
31.
32.  if(p == process_list) {
33.      process_list = process_list->next;
34.  } else {
35.      for(q = process_list; q != NULL; q = q->next) {
36.          if(q->next == p) {
37.
38. q->next = p->next;
39.
40. break;
41.          }
```



```
42.     }  
43. }  
44.  
45. process_current = old_current;  
46. }
```

### 复制代码

exit\_process 函数有两个参数，前一个是要退出的 process，后一个是当前的 process。

在这个进程要退出之前，必须要给所有的进程都要发送一个 PROCESS\_EVENT\_EXITED 事件，告诉所有与之相关的进程它要走了，如果收到这个事件的进程与要死的进程有关的话，自然会做出相应的处理（一个典型的例子是，如果一个程序要退出的话，就会给 etimer\_process 进程发送一个 PROCESS\_EVENT\_EXITED 事件，那么收到这个事件之后，etimer\_process 就会查找 timerlist 看看哪个 timer 是与这个进程相关的，就把它从 timerlist 中清除）。

如果要退出的 process 就是当前的 process，那么就只需要把它从进程列表中清除即可；如果要退出的 process 不是当前的 process，那么当前的 process 就要给要退出的 process 传递一个 PROCESS\_EVENT\_EXIT 事件让其进程的 body 退出，然后再将其从进程列表中清除。

### 关于对 protothread 进程模型的理解：

hello.c 展开宏后的代码请见帖子 [contiki 代码学习之一：浅探 prothread 进程控制模型【1】](#)的最后一段代码。我们从 process\_thread\_hello\_world\_process() 函数的执行过程来窥探一下 protothread：

1. 当第一次执行这个函数时，便会执行 PT\_YIELD\_FLAG = 1。因为 (process\_pt)->lc 被初始化为 0，所以就直接输出 hello world。
2. 当进入到 while 循环时：

1. while(1)

```
2.      {
3.          /* Wait for an event. */
4.          //PROCESS_WAIT_EVENT();
5.          do
6.          {
7.              PT_YIELD_FLAG = 0;
8.              (process_pt)->lc = __LINE__; case __LINE__: ;
9.              if(PT_YIELD_FLAG == 0)
10.             {
11.                 return 1;
12.             }
13.         } while(0);
14.         /* Got the timer's event~ */
15.     }
```

### 复制代码

首先会执行 `PT_YIELD_FLAG = 0` 语句，要注意这个宏 `__LINE__`，这个宏在程序编译的时候会被预处理成这条语句所在的行号，所以程序接着就会 `return 1`，返回这个函数。看似函数不留痕迹的退出了，其实它保留了一个重要的值 `(process_pt)->lc`，相当于记录的阻塞的位置，那么这样有什么好处呢？

3. 当这个 `process` 收到一个事件再次被执行的时候，首先会进行初始化，让 `PT_YIELD_FLAG = 1`，接着进入 `switch` 语句，因为 `(process_pt)->lc` 已经被保存，所以这次 `process` 直接跳到了上次的执行结束的地方重新开始执行了，因为 `PT_YIELD_FLAG` 已经是 1 了，所以就不会 `return` 了。就会继续执行上次执行的代码。所以 `(process_pt)->lc` 相当于保持了现场，下次函数再运行的时候就可以通过 `switch` 语句迅速找到执行代码语句的入口，恢复现场，继续执行。执行完毕后，接着 `while` 循环，`return` 后，再等待事件，一直循环。