Contiki 学习笔记(重写版)

Jelline Blog： http://jelline.blog.chinaunix.net
新个人独立博客：http://sparkandshine.net

如果您引用本文的内容，记得添加引用：

[1]苏铅坤. 无线传感器网络文件系统与重编程技术研究[D]. 电子科技大学. 2013.

图表目录

Contiki 学习笔记(重写版)[①]

Contiki 是由瑞典计算机科学研究所开发专用的网络节点操作系统，自 2003 年发布 1.0 版本以来，得到飞速发展，成为一个完整的操作系统，包括文件系统 Coffee、网络协议栈 uIP 和 Rime、网络仿真器 COOJA，并于 2012 年发布全新版本 2.6。Contiki 由标准 C 语言开发，具有很强的移植性，已被移植到多种平台，包括 8051、MSP430、AVR、ARM，并得到广泛应用。除此之外，Contiki 将 Protothreads 轻量级线程模型和事件机制完美整合到一起，Protththreads 机制使得系统占用内存极小，事件机制保证了系统低功耗，非常适合资源受限、功耗敏感的传感器网络。

本工作开展之初，Contiki 最高版本为 2.5，除了几篇官方发表的论文及少许的介绍性资料外，没有详细的参考资料。为了深入理解无线传感器网络中的文件系统和重编程技术，不得不深入分析源码重现 Contiki 的技术细节。本文关于 Contiki 所有讨论是基于 2.5 版本。

2.1 运行原理

嵌入式系统可以看作是一个运行着死循环主函数系统，Contiki 内核是基于事件驱动的，系统运行可以视为不断处理事件的过程。Contiki 整个运行是通过事件触发完成，一个事件绑定相应的进程。当事件被触发，系统把执行权交给事件所绑定的进程。一个典型基于 Contiki 的系统运行示意图如下：

[①] 分析 Contiki 操作系统源码时，已将分析笔记分享在个人博客 (http://jelline.blog.chinaunix.net)，本章大部分是对这些内容的重新整理。

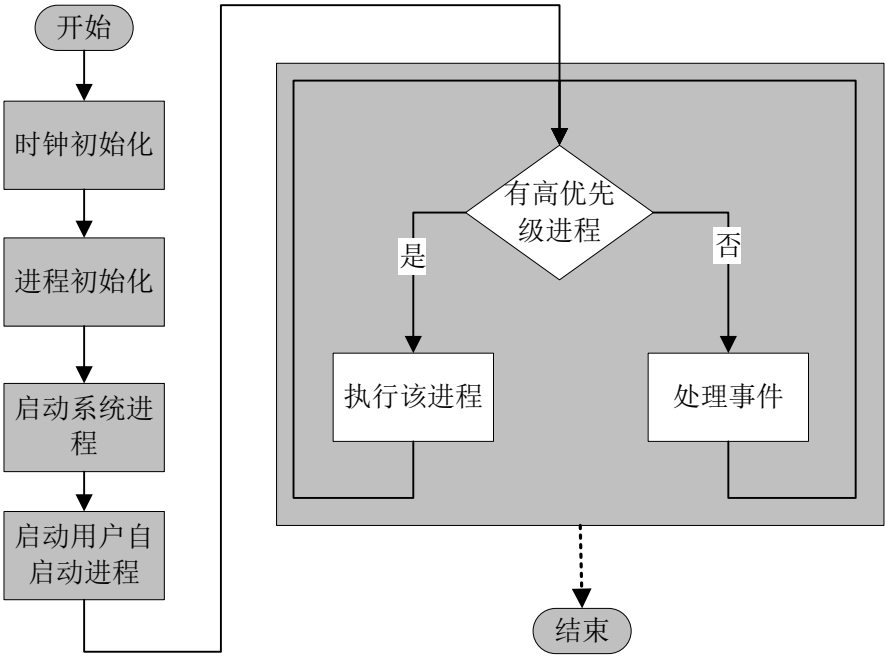

图 2- 1 Contiki 运行原理示意图

事实上，上述的框图几乎是主函数的流程图。通常情况下，应用程序作为一个进程放在自启动的指针数组中，系统启动后，先进行一系列的硬件初始化(包括串口、时钟)，接着初始化进程，启动系统进程(如管理 etimer 的系统进程 etimer_process)和用户指定的自启动进程，然后进入处理事件的死循环(如上图右边框框所示，实际上是 process_run 函数的功能)。通过遍历执行完所有高优先级的进程，而后转去处理事件队列的一个事件，处理该事件(通常对应于执行一个进程)之后，需先满足高优先级进程才能转去处理下一个事件。将 process_run 代码展开加到 main 函数，保留关键代码，如下：

```
int main()
{
   clock_init();                                  //时钟初始化
   process_init();                                //进程初始化
   process_start(&etimer_process, NULL);          //启动系统进程
   autostart_start(autostart_processes);          //启动用户自启动进程

   while(1)
   {
      /***函数 process_run 的功能***/
      if(poll_requested)
      {
         do_poll();                                //执行完所有高优先级的进程
```

```
        }
        do_event();                                    //仅处理事件队列的一个事件
    }
    return 0;
}
```

2.2 Contiki 内核

进程无疑是一个系统最重要的概述，Contiki 的进程机制是基于 Protothreads 线程模型，为确保高优先级任务尽快得到响应，Contiki 采用两级进程调度。

2.2.1 Protothreads

Contiki 使用 Protothreads[错误!未定义书签。]轻量级线程模型，在 Protothreads 基础上进行封装。为了适应内存受限的嵌入式系统，瑞典计算机科学研究所设计 Protothreads，实际上是一种轻量级无线结构的线程库。传统的桌面操作系统甚至服务器操作系统，每个进程都拥有自己的栈，进行进程切换时，将进程相关的信息(包括局部变量、断点、寄存器值)存储在栈中。然而，对于嵌入式系统，尤其是内存受限的传感器节点几乎不现实，基于这点考虑，Protothreads 巧妙地让所有进程共用一个栈，传统的进程与 Protothreads 对比示意图如下：

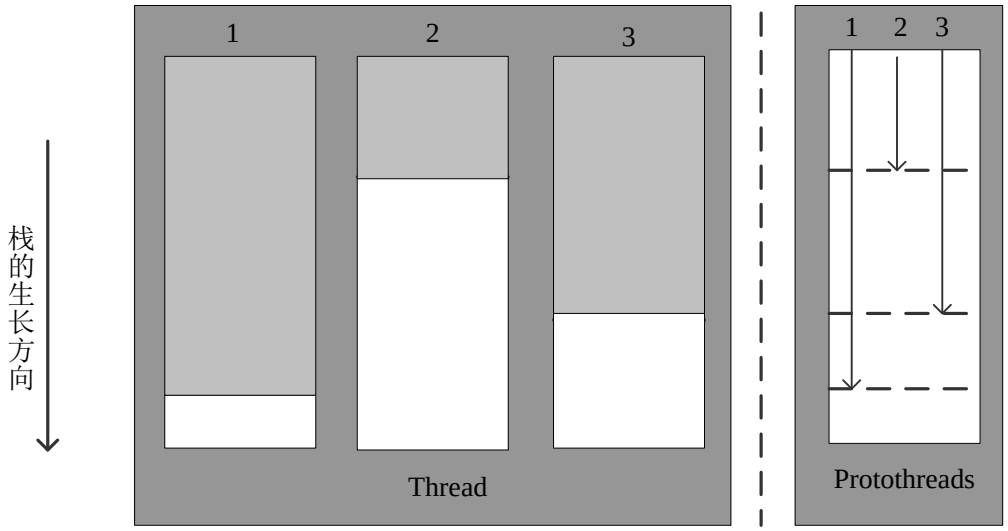

图 2- 2 Thread 与 Protothreads 栈对比示意图

从图可以看出，原本需要 3 个栈的 Thread 机制，在 Protothreads 只需要一个栈，当进程数量很多的时候，由栈空间省下来的内存是相当可观的。保存程序断点在传统的 Thread 机制很简单，只需要要保存在私有的栈，然而 Protothreads 不能将断

点保存在公有栈中。Protothreads 很巧妙地解决了这个问题，即用一个两字节静态变量存储被中断的行，因为静态变量不从栈上分配空间，所以即使有任务切换也不会影响到该变量，从而达到保存断点的。下一次该进程获得执行权的时候，进入函数体后就通过 switch 语句跳转到上一次被中断的地方。

(1)保存断点

保存断点是通过保存行数来完成的，在被中断的地方插入编译器关键字__LINE__，编译器便自动记录所中断的行数。展开那些具有中断功能的宏，可以发现最后保存行数是宏 LC_SET，取宏 PROCESS_WAIT_EVENT()为例，将其展开得到如下代码：

```
#define PROCESS_WAIT_EVENT() PROCESS_YIELD()
#define PROCESS_YIELD() PT_YIELD(process_pt)
#define PT_YIELD(pt) \
do{ \
   PT_YIELD_FLAG = 0; \
   LC_SET((pt)->lc); \
   if(PT_YIELD_FLAG == 0) \
   {
         return PT_YIELDED; \
   } \
}while(0)

#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__
```

值得一提的是，宏 LC_SET 展开还包含语句 case __LINE__，用于下次恢复断点，即下次通过 switch 语言便可跳转到 case 的下一语句。

(2)恢复断点

被中断程序再次获得执行权时，便从该进程的函数执行体进入，按照 Contiki 的编程替换，函数体第一条语句便是 PROCESS_BEGIN 宏，该宏包含一条 switch 语句，用于跳转到上一次被中断的行，从而恢复执行，宏 PROCESS_BEGIN 展开的源代码如下：

```
#define PROCESS_BEGIN() PT_BEGIN(process_pt)
#define PT_BEGIN(pt) { char PT_YIELD_FLAG = 1; LC_RESUME((pt)->lc)
#define LC_RESUME(s) switch(s) { case 0:       //switch 语言跳转到被中断的行
```

2.2.2 进程控制块

正如 Linux 一样，Contiki 也用一个结构体来描述整个进程的细节，所不同的

是，Contiki 进程控制块要简单得多。使用链表将系统所有进程组织起来，如下图所示(将 PT_THREAD 宏展开)：

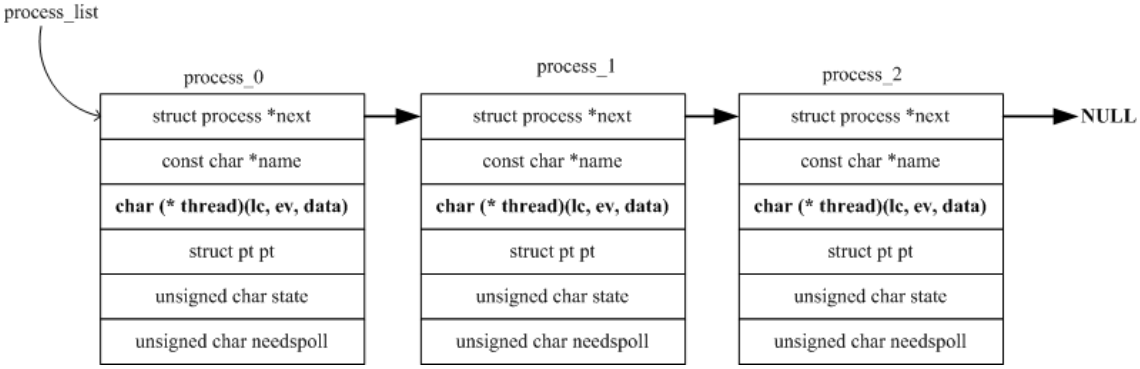

图 2- 3 Contiki 进程链表 process_list

Contiki 系统定义一个全局变量 process_list 作为进程链表的头，还定义了一个全局变量 process_current 用于指向当前进程。成员变量 next 指向下一个进程，最后一进程的 next 指向空。name 是进程的名称，可以将系统配置(定义变量 PROCESS_CONF_NO_PROCESS_NAMES 为 0)成没有进程名称，此时 name 为空字符串。变量 state 表示进程的状态，共 3 种，即 PROCESS_STATE_RUNNING、PROCESS_STATE_CALLED、PROCESS_STATE_NONE。变量 needspoll 标识进程优先级，只有两个值 0 和 1，needspoll 为 1 意味着进程具有更高的优先级。

(1)成员变量 thread

进程的执行体，即进程执行实际上是运行该函数。在实际的进程结构体代码中，该变量由宏 PT_THREAD 封装，展开即为一个函数指针，关键源代码如下：

```
PT_THREAD((*thread)(struct pt *, process_event_t, process_data_t));
#define PT_THREAD(name_args) char name_args
/***宏展开***/
char (*thread)(struct pt *, process_event_t, process_data_t);
```

(2)成员变量 pt

正如上文所述一样，Contiki 进程是基于 Protothreads，所以进程控制块需要有个变量记录被中断的行数。结构体 pt 只有一个成员变量 lc(无符号短整型)，可以将 pt 简单理解成保存行数的，相关源代码如下：

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

2.2.3 进程调度

Contiki 只有两种优先级，用进程控制块中变量 needspoll 标识，默认情况是 0，即普通优先级。想要将某进程设为更高优先级，可以在创建之初指定其 needspoll 为 1，或者运行过程中通过设置该变量动态提升其优先级。实际的调度中，会先运行有高优先级的进程，而后再去处理一个事件，随后又运行所有高优先级的进程。通过遍历整个进程链表，将 needspoll 为 1 的进程投入运行，关键代码如下：

```
/***do_poll()关键代码，由 process_run 调用***/
for(p = process_list; p != NULL; p = p->next)     //遍历进程链表
{
   if(p->needspoll)
   {
      p->state = PROCESS_STATE_RUNNING;                  //设置进程状态
      p->needspoll = 0;
      call_process(p, PROCESS_EVENT_POLL, NULL);    //将进程投入运行
   }
}
```

以上是进程的总体调度，具体到单个进程，成员变量 state 标识着进程的状态，共有三个状态 PROCESS_STATE_RUNNING、PROCESS_STATE_CALLED、PROCESS_STATE_NONE。Contiki 进程状态转换如下图：

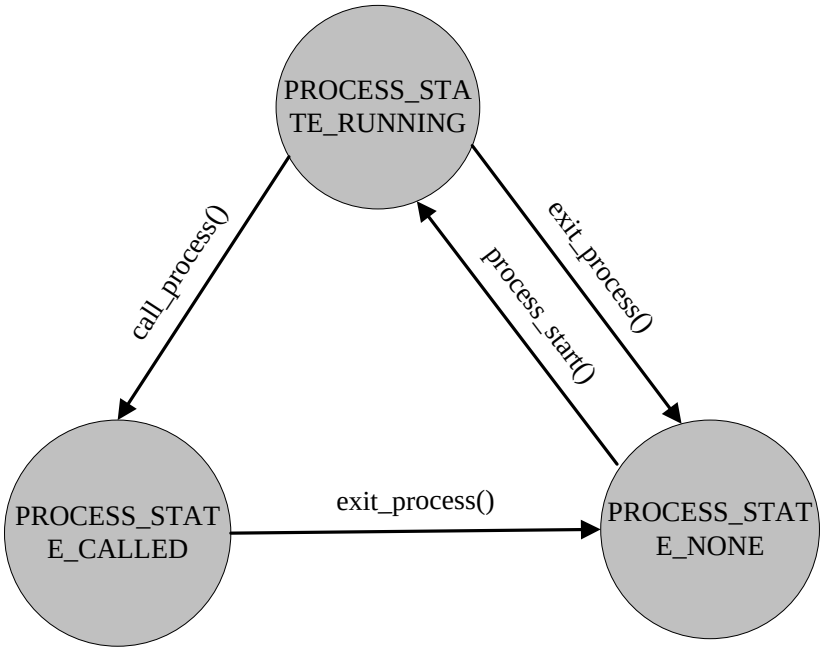

图 2- 4 Contiki 进程状态转换图

创建进程(还未投入运行)以及进程退出(但此时还没从进程链表删除)，进程状态都为 PROCESS_STATE_NONE。通过进程启动函数 process_start 将新创建的进

程投入运行队列(但未必有执行权)，真正获得执行权的进程状态为PROCESS_STATE_CALLED，处在运行队列的进程(包括正在运行和等待运行)可以调用 exit_process 退出。

(1)进程初始化

系统启动后需要先将进程初始化，通常在主函数调用，进程初始化主要完成事件队列和进程链表初始化。将进程链表头指向为空，当前进程也设为空。process_init 源代码如下：

```
void process_init(void)
{
  /***初始化事件队列***/
  lastevent = PROCESS_EVENT_MAX;
  nevents = fevent = 0;
  process_maxevents = 0;
  /***初始化进程链表***/
  process_current = process_list = NULL;
}
```

(2)创建进程

创建进程实际上是定义一个进程控制块和定义进程执行体的函数。宏 PROCESS 的功能包括定义一个结构体，声明进程执行体函数，关键源代码如下(假设进程名称为 Hello world)：

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

/***PROCESS 宏展开***/
PROCESS_THREAD(name, ev, data); \
struct process name = { NULL, strname, process_thread_##name }

/***PROCESS_THREAD 宏展开***/
static PT_THREAD(process_thread_##name(struct pt *process_pt, process_event_t ev,
process_data_t data))
#define PT_THREAD(name_args) char name_args

/***将参数代入，PROCESS 宏最后展开结果***/
static char process_thread_hello_world_process(struct pt *process_pt, process_event_t
ev, process_data_t data);
struct process hello_world_process =  \
{NULL, "Hello world", process_thread_hello_world_process };
```

可见，PROCESS 宏实际上声明一个函数并定义一个进程控制块，新创建的进程 next 指针指向空，进程名称为“Hello world”，进程执行体函数指针为 process_thread_hello_world_process，保存行数的 pt 为 0，状态为 0(即

PROCESS_STATE_NONE)，优先级标记位 needspoll 也为 0(即普通优先级)。

PROCESS 定义了结构体并声明了函数，还需要实现该函数，通过宏 PROCESS_THREAD 实现。值得注意的是，尽管 PROCESS 宏展开包含了宏 PROCESS_THREAD，用于声明函数，而这里是定义函数，区别在于前者宏展开后面加了个分号。定义函数框架代码如下：

```
PROCESS_THREAD(hello_world_process, ev, data)
//static char process_thread_hello_world_process(struct pt *process_pt, process_event_t ev, process_data_t data)
{
  PROCESS_BEGIN(); //函数开头必须有
  /***代码放在这***/
  PROCESS_END(); //函数末尾必须有
}
```

欲实现的代码必须放在宏 PROCESS_BEGIN 与 PROCESS_END 之间，这是因为这两个宏用于辅助保存断点信息(即行数)，宏 PROCESS_BEGIN 包含 switch(process_pt->lc)语句，这样被中断的进程再次获利执行便可通过 switch 语句跳转到相应的 case，即被中断的行。

(3)启动进程

函数 process_start 用于启动一个进程，首先进行参数验证，即判断该进程是否已经在进程链表中，而后将进程加到链表，给该进程发一个初始化事件 PROCESS_EVENT_INIT。函数 process_start 流程图如下：

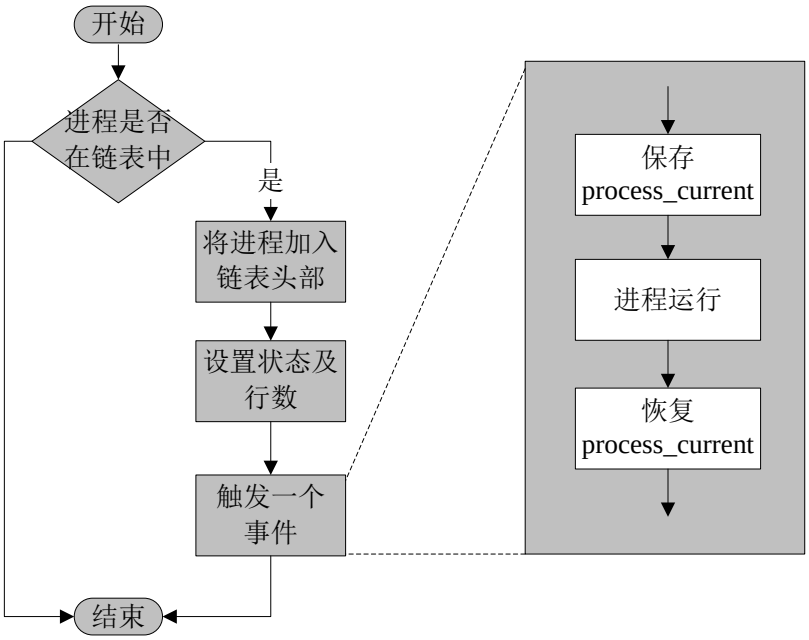

图 2- 5 函数 process_start 流程图

process_start 将进程状态设为 PROCESS_STATE_RUNNING，并调用 PT_INIT 宏将保存断点的变量设为 0(即行数为 0)。调用 process_post_synch 给进程触发一个同步事件，事件为 PROCESS_EVENT_INIT。考虑到进程运行过程中可能被中断(比如中断)，在进程运行前将当前进程指针保存起来，执行完再恢复。进程运行是由 call_process 函数实现。

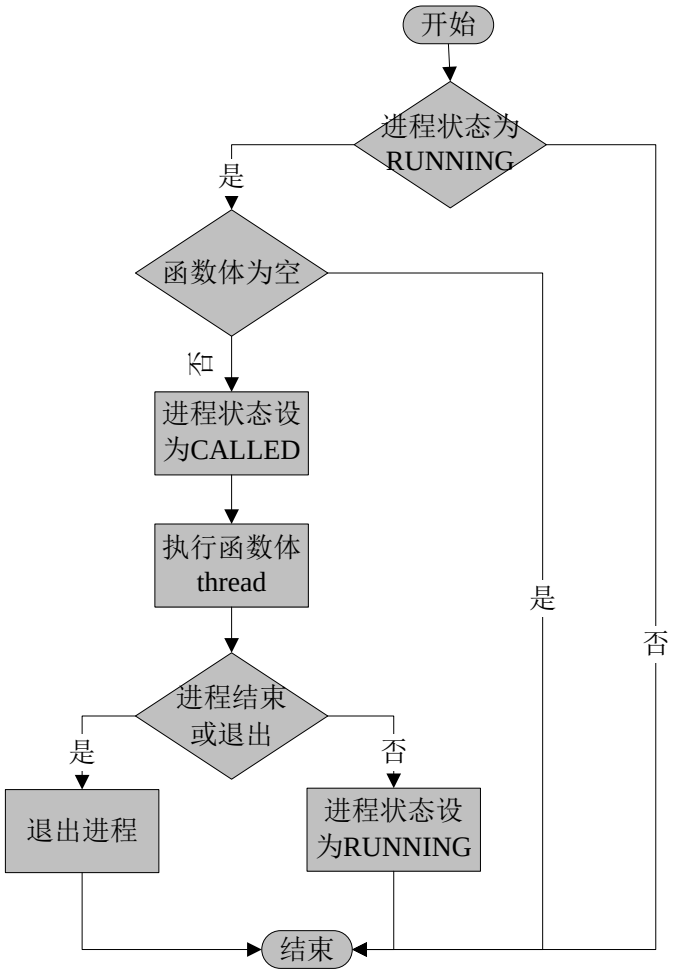

图 2- 6 call_process 流程图

call_process 首先进行参数验证，即进程处于运行状态(退出尚未删除的进程状态为 PROCESS_STATE_NONE)并且进程的函数体不为空，接着将进程状态设为 PROCESS_STATE_CALLED，表示该进程拥有执行权。接下来，运行进程函数体，根据返回值判断进程是否结束(主动的)或者退出(被动的)，若是调用 exit_process 将进程退出，否则将进程状态设为 PROCESS_STATE_RUNNING，继续放在进程链表。

(4) 进程退出

进程运行完或者收到退出的事件都会导致进程退出。根据 Contiki 编程规划，进程函数体最后一条语句是 PROCESS_END()，该宏包含语句 return PT_ENDED，表示进程运行完毕。系统处理事件时(事件绑定进程，事实上执行进程函数体)，倘若该进程恰好收到退出事件，thread 便返回 PT_EXITED，进程被动退出。还有就是给该进程传递退出事件 PROCESS_EVENT_EXIT 也会导致进程退出。进程退出函数 exit_process 流程图如下：

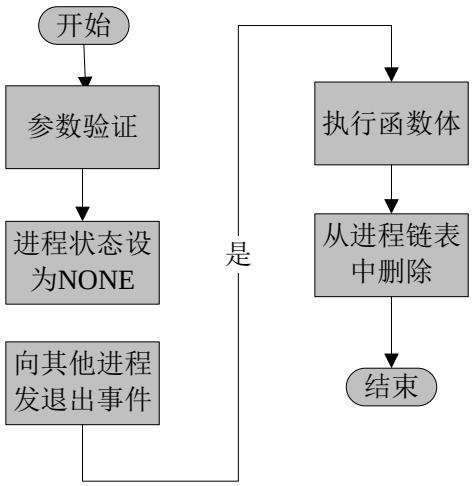

图 2- 7 exit_process 流程图

进程退出函数 exit_process 首先对传进来的进程 p 进行参数验证，确保该进程在进程链表中并且进程状态为 PROCESS_STATE_CALLED/RUNNING(即不能是 NONE)，接着将进程状态设为 NONE。随后，向进程链表的所有其他进程触发退出事件 PROCESS_EVENT_EXITED，此时其他进程依次执行处理该事件，其中很重要一部分是取消与该进程的关联。进程执行函数体 thread 进行善后工作，最后将该进程从进程链表删除。

2.2.4 事件调度

事件驱动机制广泛应用于嵌入式系统，类似于中断机制，当有事件到来时(比如按键、数据到达)，系统响应并处理该事件。相对于轮询机制，事件机制优势很明显，低功耗(系统处于休眠状态，当有事件到达时才被唤醒)和 MCU 利用率高。

Contiki 将事件机制融入 Protothreads 机制，每个事件绑定一个进程(广播事件例外)，进程间的消息传递也是通过事件来传递的。用无符号字符型来标识事件，事件结构体 event_data 定义如下：

```
struct event_data
{
   process_event_t ev;
   process_data_t data;
   struct process *p;
};

typedef unsigned char process_event_t;
typedef void *process_data_t;
```

用无符号字符型标识一个事件，Contiki 定义了 10 个事件(0x80~0x8A)，其他的供用户使用。每个事件绑定一个进程，如果 p 为 NULL，表示该事件绑定所有进程(即广播事件 PROCESS_BROADCAST)。除此之外，事件可以携带数据 data，可以利用这点进行进程间的通信(向另一进程传递带数据的事件)。

Contiki 用一个全局的静态数组存放事件，这意味着事件数目在系统运行之前就要指定(用户可以通过 PROCESS_CONF_NUMEVENTS 自选配置大小)，通过数组下标可以快速访问事件。系统还定义另两个全局静态变量 nevents 和 fevent，分别用于记录未处理事件总数及下一个待处理的位置。事件逻辑组成环形队列，存储在数组里，如下图：

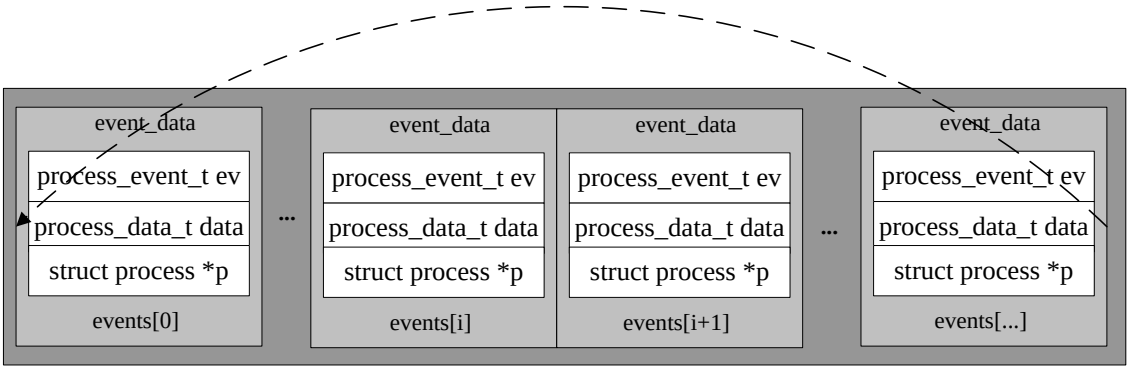

图 2- 8 Contiki 事件队列示意图

可见对于 Contiki 系统而言，事件并没有优先级之分，而是先到先服务的策略，全局变量 fevent 记录了下一次待处理事件的下标。

(1)事件产生

Conitki 有两种方式产生事件，即同步和异步。同步事件通过 process_post_synch 函数产生，事件触发后直接处理(调用 call_process 函数)。而异步事件产生是由 process_post 产生，并没有及时处理，而是放入事件队列等待处理，process_post 流程图如下：

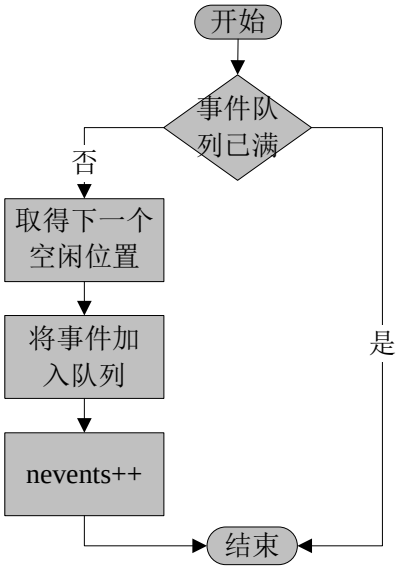

图 2- 9 process_post 函数流程图

process_post 首先判断事件队列是否已满，若满返回错误，否则取得下一个空闲位置(因为是环形队列，需做余操作)，而后设置该事件并将未处理事件总数加 1。

(2)事件调度

事件没有优先级，采用先到先服务策略，每一次系统轮询(process_run 函数)只处理一个事件，do_event 函数用于处理事件，其流程图如下：

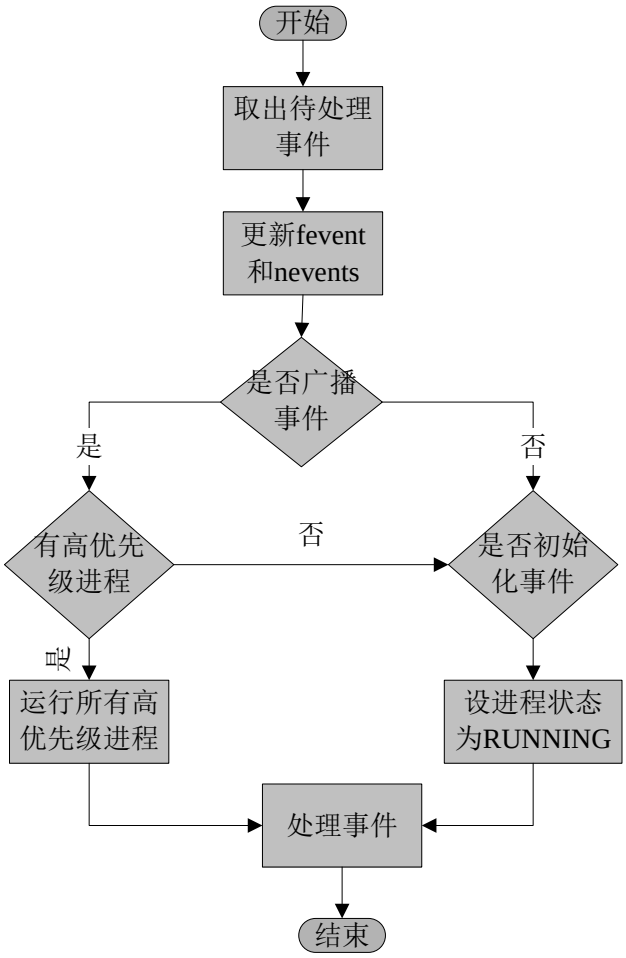

图 2- 10 do_event 函数流程图

do_event 首先取出该事件(即将事件的值复制到一个新变量)，更新总的未处理事件总数及下一个待处理事件的数组下标(环形队列，需要取余操作)。接着判断事件是否为广播事件 PROCESS_BROADCAST，若是，考虑到处理广播事件可能需要更多的时间，为保证系统实时性，先运行高优先级的进程，而后再去处理事件(调用 call_process 函数)。如果事件是初始化事件 PROCESS_EVENT_INIT (创建进程的时候会触发此事件)，需要将进程状态设为 PROCESS_STATE_RUNNING。

(3)事件处理

实际的事件处理是在进程的函数体 thread，正如上文所说的那样，call_process 会调用 tread 函数，执行该进程。关键代码如下：

```
ret = p->thread(&p->pt, ev, data);
```

2.2.5 定时器

Contiki 内核是基于事件驱动和 Protothreads 机制，事件既可以是外部事件(比如按键，数据到达)，也可以是内部事件(如时钟中断)。定时器的重要性不言而喻，Contiki 提供了 5 种定时器模型，即 timer(描述一段时间，以系统时钟嘀嗒数为单位)、stimer(描述一段时间，以秒为单位)、ctime(定时器到期，调用某函数，用于 Rime 协议栈)、etime(定时器到期，触发一个事件)、rtimer(实时定时器，在一个精确的时间调用函数)。

鉴于 etimer 在 Contiki 使用的广泛性，管理这些 etimer 由系统进程 etimer_process 管理，本小节详细简单 etimer 相关技术细节。

(1) etimer 组织结构

etimer 作为一类特殊事件存在，也是跟进程绑定。除此之外，还需变量描述定时器属性，etimer 结构体定义如下：

```
struct etimer
{
  struct timer timer;       //包含起始时刻和间隔两成员变量
  struct etimer *next;      //指向下一个 etimer
  struct process *p;
};
```

成员变量 timer 用于描述定时器属性，包含起始时刻及间隔，将起始时刻与间隔相加与当前时钟对比，便可知道是否到期。变量 p 指向所绑定的进程(p 为 NULL 则表示该定时器与所有进程绑定)。成员变量 next，指向下一个 etimer，系统所有 etimer 被链接成一个链表，如下图所示：

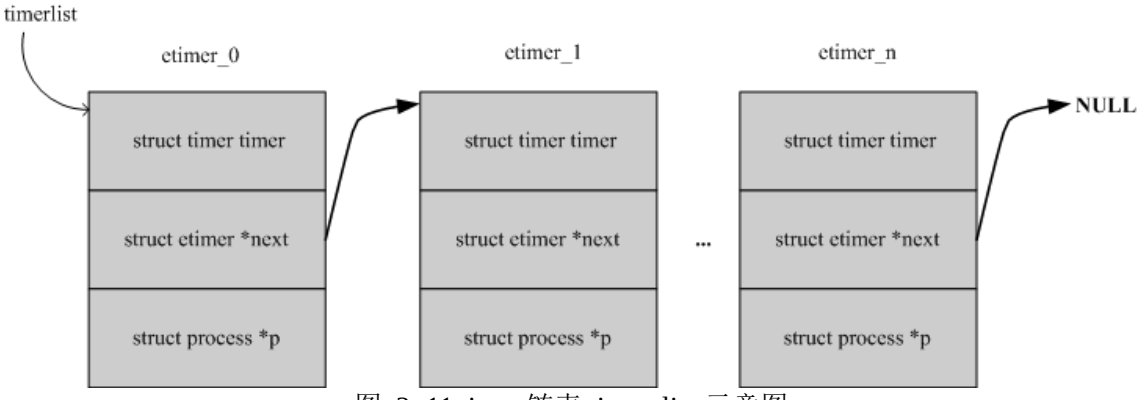

图 2- 11 timer 链表 timer_list 示意图

(2)添加 etimer

定义一个 etimer 结构体，调用 etimer_set 函数将 etimer 添加到 timerlist，函数

etimer_set 流程图如下：

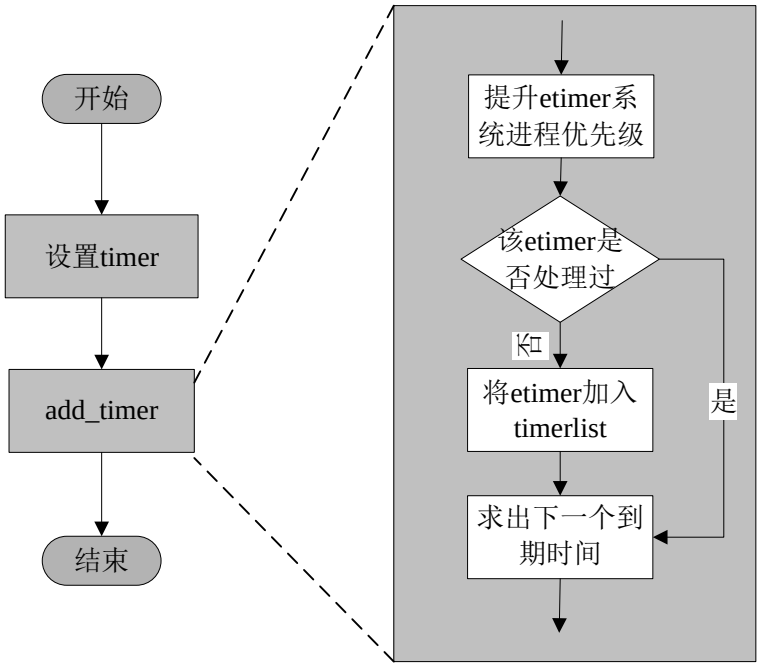

图 2- 12 etimer_set 流程图

etimer_set 首先设置 etimer 成员变量 timer 的值(由 timer_set 函数完成)，即用当前时间初始化 start，并设置间隔 interval，接着调用 add_timer 函数，该函数首先将管理 etimer 系统进程 etimer_process 优先级提升，以便定时器时间到了可以得到更快的响应。接着确保欲加入的 etimer 不在 timerlist 中(通过遍历 timerlist 实现)，若该 etimer 已经在 etimer 链表，则无须将 etimer 加入链表，仅更新时间。否则将该 etimer 插入到 timerlist 链表头位置，并更新时间(update_time)。这里更新时间的意思是求出 etimer 链表中，还需要多长 next_expiration(全局静态变量)时间，就会有 etimer 到期。

(3) etimer 管理

Contiki 用一个系统进程 etimer_process 管理所有 etimer 定时器。进程退出时，会向所有进程发送事件 PROCESS_EVENT_EXITED，当然也包括 etimer 系统进程 etimer_process。当 etimer_process 拥有执行权的时候，便查看是否有相应的 etimer 绑定到该进程，若有就删除这些 etimer。除此之外，etimer_process 还会处理到期的 etimer，etimer_process 的 thread 函数流程图如下：

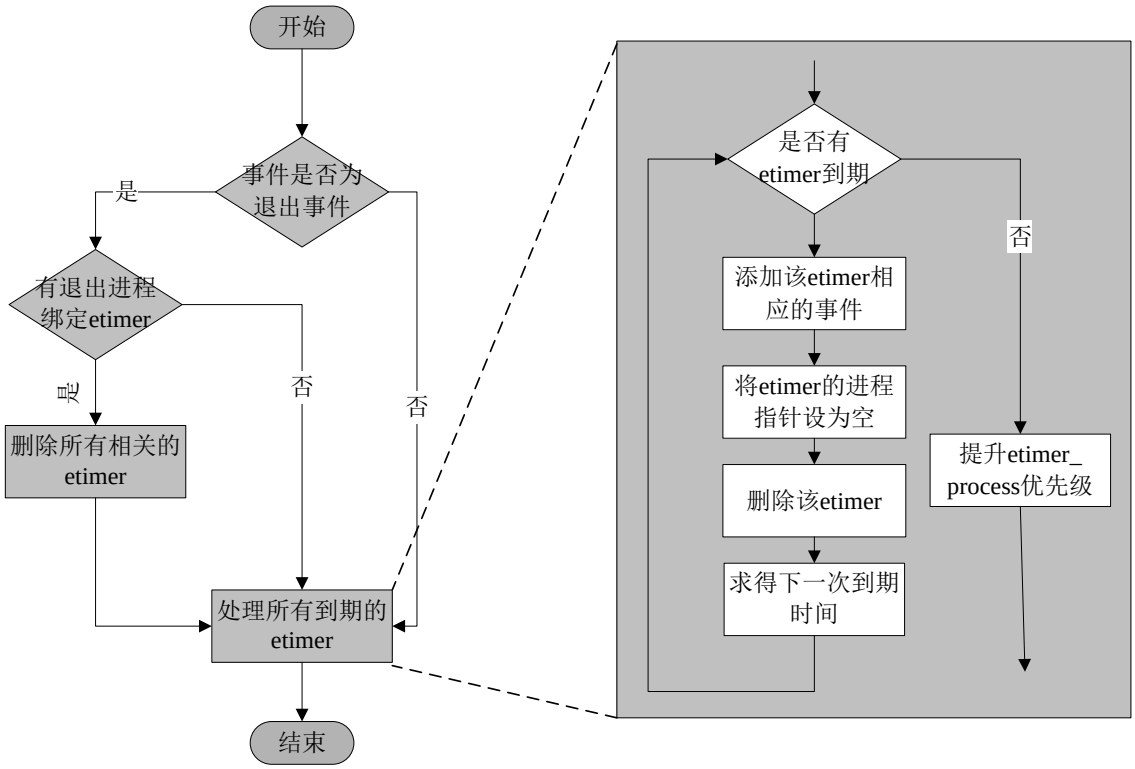

图 2- 13 etimer_process 的函数 thread 流程图

etimer_process 获得执行权时，若传递的是退出事件，遍历整个 timerlist，将与该进程(通过参数 data 传递)相关的 etimer 从 timerlist 删除，而后转去所有到期的 etimer。通过遍历整个 etimer 查看到期的 etimer，若有到期，发绑定的进程触发事件 PROCESS_EVENT_TIMER，并将 etimer 的进程指针设为空(事件已加入事件队列，处理完毕)，接着删除该 etimer，求出下一次 etimer 到期时间，继续检查是否还有 etimer 到期。提升 etimer_process 优先级，若接下来都没有 etimer 到期了，就退出。总之，遍历 timerlist，只要 etimer 到期，处理之后重头遍历整个链表，直到 timerlist 没有到期的 etimer 就退出。

2.3 文件系统 Coffee

本节内容见第三章。

2.4 动态加载

本节内容见第四章。

2.5 Rime 协议栈

传统的分层通信架构(communication architectures)很难满足资源受限的传感器网络，于是研究者转向跨层优化(比如将顶层数据聚合功能放在底层实现)，但这导致系统变得更脆弱以及难以控制(fragile and unmanageable systems)。因此，传统分层通信结构再次得到重视，同时研究发现，传统分层效率几乎可以与跨层优化相媲美[1]。基于此，Rime 也采用分层结构。

2.5.1 协议栈结构

Rime 是针对传感器网络轻量级、层次型协议栈，也是低功耗、无线网络协议栈，旨在简化传感器网络协议及代码重用，属于 Contiki 的一部分(Contiki 还支持 uIPv4、uIPv6、LwIP)。Rime 协议栈结构框图如下：

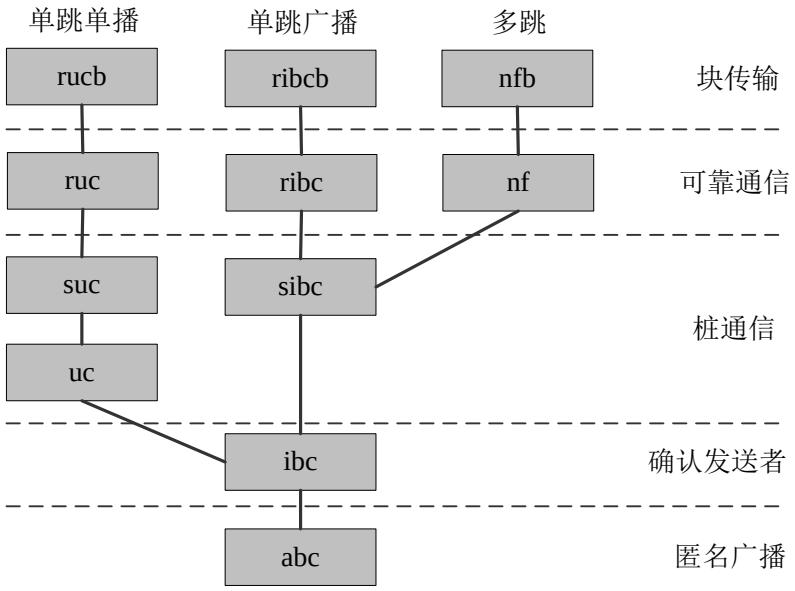

图 2- 14 Rime 协议栈结构框图

上图中单跳单播各个缩写含义如下：

rucb

rucb 是单跳单播的最顶层，将数据以块为单位进行传输(Bulk transfer)。

ruc

ruc 是指 Reliable communication。可靠通信由两层实现：Stubborn transmission、Reliable transmission。该层主要实现确认和序列功能(acknowledgments and

sequencing)。

suc

suc 指 Stubborn transmission，是可靠通信的另一层。suc 这一层在给定的时间间隔不断地重发数据包，直到上层让其停止。为了防止无限重发，需要指定最大重发次数(maximum retransmission number)。

ibc

ibc 表示 identified sender best-effort broadcast，将上层的数据包添加一个发送者身份(sender identity)头部。

uc

uc 意思是 unicast abstraction，将上层的数据包添加一个接收者头部。

abc

abc 意思是 anonymous broadcast，匿名广播。即将数据包通过无线射频驱动(radio driver)发出去，接收来自无线射频驱动所有的包并交给上层。

2.5.2 建立连接

使用 Rime 协议栈进行通信之前，需要建立连接。Rime 协议栈提供单跳单播、单跳广播、多跳三种功能。在此，仅介绍单跳单播(Single-hop unicast)连接建立过程。

建立连接的实质是保存该连接一些信息(如发送者、接收者)，Rime 协议栈用一系列结构体保存这些链接状态信息。Rime 每一层都有相应的连接结构体(以_conn 结尾)，上层嵌套下层，如下：

rucb_conn --> runicast_conn --> stunicast_conn --> unicast_conn --> broadcast_conn --> abc_conn

每个连接结构体都有相应的回调结构体(以_callbacks 后缀结尾)，该结构体的成员变量实为发送、接收函数指针。当接收到一个数据报，会调用该结构体相应的函数。回调结构体层次如下：

rucb_callbacks --> runicast_callbacks --> stunicast_callbacks --> unicast_callbacks --> broadcast_callbacks --> abc_callbacks

综上，连接建立_open、连接结构体_conn、回调结构体_callbacks 间的关系如下图：

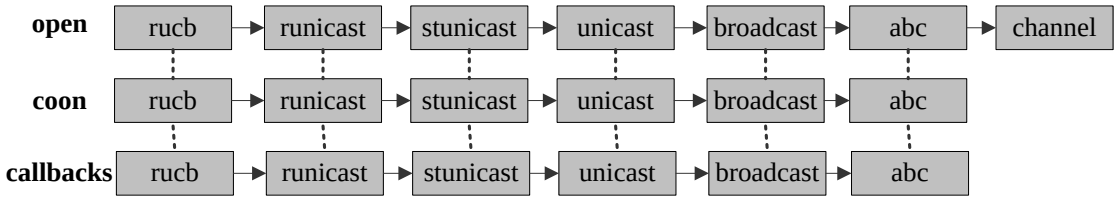

图 2- 15 open、coon、callbacks 对应关系

(1)连接结构体

建立连接，实质是初始化结构体 rucb_conn 各个成员变量，结构体 rucb_conn 定义如下：

```
struct rucb_conn
{
   struct runicast_conn c;
   const struct rucb_callbacks *u;
   rimeaddr_t receiver, sender;
   uint16_t chunk;
   uint8_t last_seqno;
};
```

结构体 rucb_conn 各成员变量含义如下：

c

uc(unicast abstraction)将上层的数据包添加一个接收者头部传递给下一层，这里的 c 指的是下一层连接结构体。

u

结构体 rucb_callbacks 有 3 个函数指针成员变量写数据块 write_chunk、读数据块 read_chunk、超时 timedout，需要用户自己实现。

receiver、sender

用于标识接收者和发送者。这里的 receiver 是指目的节点的接收地址。

chunk

数据块数目。

last_seqno

一次数据发送多个片段的最后一个序列号，当接收端接收到数据时，判断其序列号是否等于最后一个序列号，若等于则不接收(即接收到最后一个数据块，停止接收)。

2.5.3 数据发送

Rime 协议栈建立连接后，就可以进行通信了(发送、接收数据)，Rime 协议栈提供单跳单播、单跳广播、多跳三种功能。在此，仅介绍单跳单播(Single-hop unicast)发送数据情型。

Rime 是层次型协议栈，整个发送数据过程是通过上层调用下层服务来完成的，具体如下：

rucb_send --> runicast_send --> stunicast_send_stubborn --> unicast_send --> broadcast_send --> abc_send --> rime_output --> NETSTACK_MAC.send

rucb 是块传输(Bulk transfer)层，可以理解成传输层，数据发送函数 rucb_send 源代码如下：

```
int rucb_send(struct rucb_conn *c, const rimeaddr_t *receiver)
{
  c->chunk = 0;
  read_data(c);
  rimeaddr_copy(&c->receiver, receiver);
  rimeaddr_copy(&c->sender, &rimeaddr_node_addr);
  runicast_send(&c->c, receiver, MAX_TRANSMISSIONS);
  return 0;
}
```

c->chunk 将数据块数目初始化为 0，read_data 进行一些 Rime 缓冲区初始化相关工作。rimeaddr_copy 函数设置接收者 receiver 和发送者 sender 的 Rime 地址，rimeaddr_node_addr 用于标识本节点的 Rime 地址。接下来，调用下一层的发送函数 runicast_send 完成发送。

2.5.4 数据接收

Rime 协议栈建立连接后，就可以调用数据接收函数 recv 来接收数据，整个接收数据过程是通过上层调用下层服务来完成的，具体如下：

recv --> recv_from_stunicast --> recv_from_uc --> recv_from_broadcast --> recv_from_abc

函数 recv 首先判断该数据包是不是最后一个序列(数据包被拆分的情况下)，若不是，将收到的数据写入物理存储介质。recv 函数流程图如下：

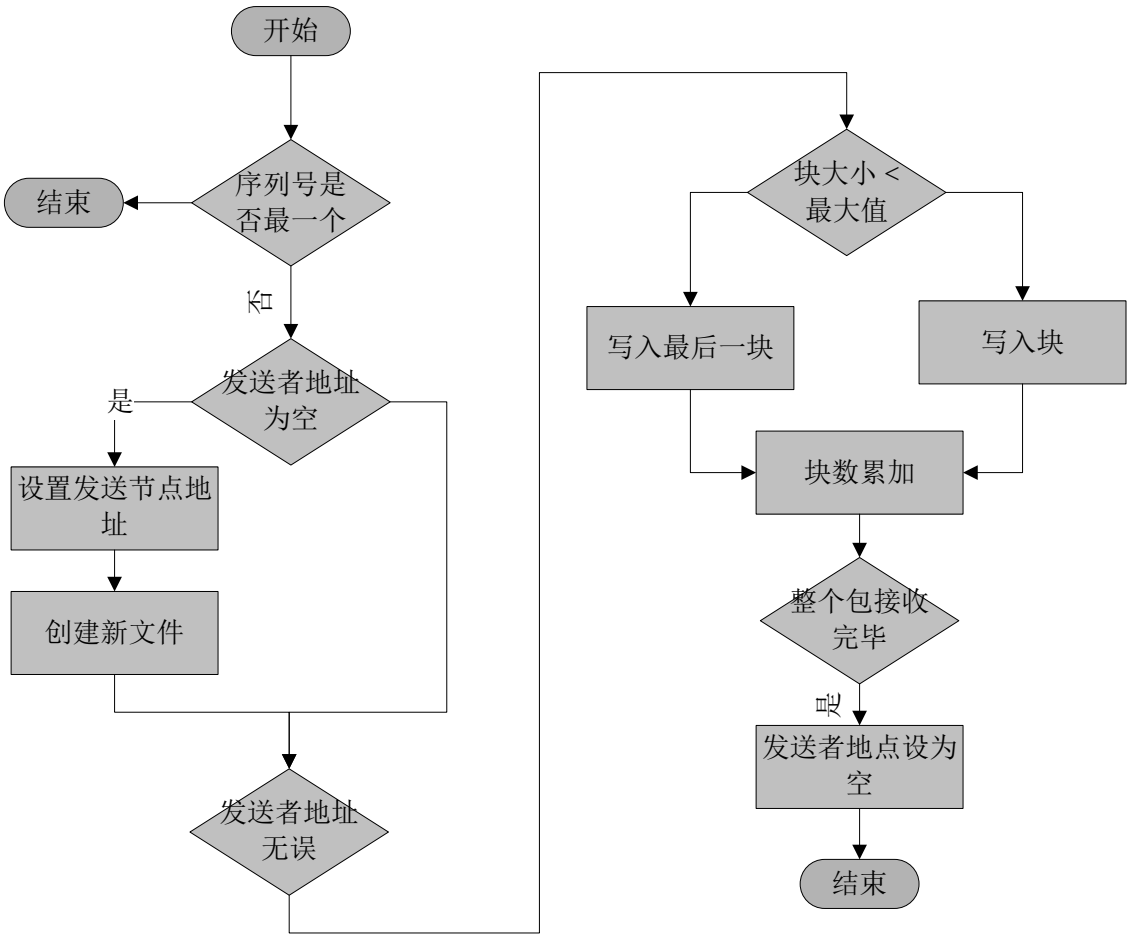

图 2- 16 recv 函数流程图

函数 recv 首先判断接收到的包是不是最后一个序列(数据包太大时，需要拆分)，如果是最后一个就返回(用最后序列号标识包传递完毕)。若不是最后一个序列，意味着还有数据要接收。接着判断发送者地址是否为空，若是，说明节点未曾接收该包的任何序列，则建立文件以存放数据。确保发送地址无误之后，若块小于块的最大值(RUCB_DATASIZE)，即这是数据包最后的一块，写入最后一块，否则正常写入这块的数据。把块的数目累加，接着判断这块是否是最后一块(最后一块意味着数据包传输完毕)，若是则将发送者地址设为空，否则返回。

2.5.5 释放连接

数据通信完毕之后，需要释放连接，以供其他进程使用。关闭连接实质上是将相应的连接结构体从链接表中删除。整个调用过程如下：

rucb_close -> runicast_close -> stunicast_close -> unicast_close -> broadcast_close -> abc_close -> channel_close -> list_remove

2.6 本章小结

本章深入浅出介绍 Contiki 操作系统内核和 Rime 协议栈的技术细节。首先，从全局视角出发描述了整个系统是如何运行的，即通过反复执行所有高优先级进程以及处理事件的方式。接着，循序渐进对 Contiki 两个核心机制进行剖析。先是介绍了 Protothreads 原理以及如何减少内存使用，分析了进程控制块以及进程调度，包括总体调度策略、进程状态转换、进程初始化、创建进程、启动进程、进程退出。随后介绍了事件机制，包括事件产生、事件调度、事件处理。除此之外，还分析了定时器这类特殊事件，包括创建定时器以及系统如何管理这些定时器。最后，剖析了 Rime 协议栈，先给出整体结构，而后分别介绍连接建立、数据发送、数据接收、释放连接。

参考文献

[1] 刘国兴,余镇危.无线传感器网络综述[J]. International Conference on Internet Technology and Applications,2011

[2] 杨卓静,孙宏志,任晨虹.无线传感器网络应用技术综述[J].中国科技信息,2010(13).

[3] Jennifer Yick, Biswanath Mukherjee, Dipak Ghosal.Wireless sensor network survey[J].Computer Network.2008,52(12): 2292-2330

[4] 贺慧琳,肖强华.无线传感器网络数据收集研究综述[J].电脑知识与技术,2011(30).

[5] Muhammad Omer Farooq,Thomas Kunz.Operating Systems for Wireless Sensor Networks: A Survey[J].Sensors.2011

[6] 余向阳. 无线传感器网络研究综述[J]. 单片机与嵌入式系统应用,2008,No.9208:8-12.

[7] Dunkels, A.,Gronvall, B.,Voigt, T..Contiki-a lightweight and flexible operating system for tiny networked sensors[J].29th Annual IEEE International Conference on Local Computer Networks.2004,455-462

[8] Dunkels Adam,Schmidt Oliver,Voigt Thiemo.Protothreads:Simplifying event-driven programming of memory-constrained embedded systems[J].Proceedings of the Fourth International Conference on Embedded Networked Sensor Systems.2006,29-42

[9] Adam Dunkels, Oliver Schmidt.Protothreads Lightweight, Stackless Threads in C. SICS Technical Report,2005

[10] 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

[11] Hewage Kasun,Keppitiyagama Chamath,Thilakarathna Kenneth.TikiriDev: A UNIX-like device abstraction for Contiki[J]. Real-World Wireless Sensor Networks.2010,p74-81

[12] Tsiftes Nicolas,Eriksson Joakim,Dunkels Adam.Low-power wireless IPv6 routing with ContikiRPL[J].Proceedings of the 9th ACM/IEEE International Conference on Information Processing in Sensor Networks,2010:406-407

[13] Hornsby A.,Bail E..μXMPP: Lightweight implementation for low power operating system Contiki[J].International Conference on Ultra Modern Telecommunications & Workshops.2009,1-5

[14] Paul Tomsy,Kumar G.,Santhosh.Safe contiki OS: Type and memory safety for contiki OS[J].International Conference on Advances in Recent Technologies in Communication and Computing,2009:169-171

[15] Casado Lander,Tsigas Philippas.ContikiSec: A secure network layer for wireless sensor networks under the Contiki Operating System[J].Nordic Conference on Secure IT Systems,2009:133-147

[16] Bruno L.,Franceschinis M.,Pastrone C.,Tomasi R.,Spirito M..6LoWDTN: IPv6-enabled Delay-Tolerant WSNs for Contiki[J].International Conference on Distributed Computing in Sensor Systems and Workshops.2011,1-6

[17] ContikiWiki: http://www.sics.se/contiki/wiki/index.php/Main_Page

[18] The Contiki OS: http://www.contiki-os.org/

[19] 周皓.IP 传感器网络下的普适计算系统设计[D].上海交通大学,2010.

[20] Djenouri Djamel,Balasingham Ilangko.Power-aware QoS geographical routing for wireless sensor networks -Implementation using Contiki[J].International Conference on Distributed Computing in Sensor Systems,2010

[21] 俄立波.文件系统在无线传感器网络中的应用研究[D].上海交通大学.2009.

[22] Cao, Q.; Abdelzaher, T.; Stankovic, J.; He, T. The LiteOS Operating System: Towards Unix Like Abstraction for Wireless Sensor Networks. In Proceedings of the 7th International Conference on Information Processing in Sensor Networks (IPSN 2008), St. Louis, MO, USA, 22–24 April 2008.

[23] Sun Jun-Zhao.Dissemination protocols for reprogramming wireless sensor networks: A literature survey[J].International Conference on Sensor Technologies and Applications,2010:151-156

[24] Sun Jun-Zhao.OS-based reprogramming techniques in wireless sensor networks: A survey[J].IEEE International Conference on Ubi-Media Computing,2010:17-23

[25] 张集文,李士宁,贾军博,周涛.无线传感器网络重编程技术的研究与设计[J].计算机测量与控制,2009(07)

[26] 张羽,周兴社,蒋泽军,王丽芳.传感器网络远程网络重编程服务安全认证机制研究[J].计算机科学,2008(10)

[27] 蒋泽军,张英,王丽芳,方智毅.无线传感器网络重编程安全认证机制研究[J]西北工业大学学报,2009(06)

[28] 张羽,周兴社,Yee WeiLaw,Marimuthu Palaniswami.一种抗污染攻击的传感器网络重编程方法[J]西北工业大学计算机学院,2011(03)

[29] 任超,张羽,方智毅.一种基于散列链的无线网络重编程安全认证机制[J].自然科学进展,2009(10)

[30] 方智毅,王丽芳,张羽,蒋长清.无线传感器网络重编程中 DoS 攻击初探[J].西北工业大学学报,2009(05)

[31] 孟硕培.无线传感器网络节点重编程研究与设计[D].浙江大学.2008

[32] 俄立波.文件系统在无线传感器网络中的应用研究[D].上海交通大学.2009.

[33] Adam Dunkels, Fredrik Österlind, Zhitao He. An adaptive communication architecture for wireless sensor networks[C]. in Proceedings of the Fifth ACM Conference on Networked Embedded Sensor Systems. 2007.

[34] 苏铅坤, 罗克露, 周强. 基于无线传感器网络文件系统 Coffee 的研究[J]. 计算机技术与发展, 2013, 23(1):67-70.

[35] 董玮. 面向无线传感网络的嵌入系操作系统设计[D]. 浙江：浙江大学. 2010.

[36] Crossbow Technology, Inc. Mote In-Network Programming User Reference Version 20030315, 2003. http://webs.cs.berkeley.edu/tos/tinyosl.x/doc/Xnp.pdf.

[37] Jonathan W. Hui and David Culler. The dynamic behavior of a data dissemination protocol for network programming at scale. In Proc. SenSys '04,Baltimore, Maryland, USA, November 2004.

[38] Thanos Stathopoulos, John Heidemann, and Deborah Estrin, A Remote Code Update Mechanism for Wireless Sensor Networks. Technical report CENS-TR-30, University of California, Los Angeles,Center for Embedded Networked Computing,November, 2003, CA, USA, 2003

[39] S-J. Park, R. Vedantham, R. Sivakumar, and I. F.Akyildiz. A scalable approach for reliable downstream data delivery in wireless sensor networks. In Proceedings of the ACM International Symposium on Mobile Ad Hoc Networking and Computing (MobiHoc), pages 78-89, May 2004.

[40] Shah Bhatti, James Carlson, Hui Dai, Jing Deng, Jeff Rose, Anmol Sheth, Brian Shucker, Charles Gruenwald, Adam Torgerson, Richard Han,MANTIS OS: An Embedded Multithreaded Operating System for Wireless Micro Sensor Platforms, ACMKluwer Mobile Networks & Applications (MONET) Journal, Special Issue on Wireless Sensor Networks, August 2005.

[41] Niels Reijers and Koen Langendoen, "Efficient Code Distribution in Wireless Sensor Networks, "2nd ACM Int. Workshop on Wireless Sensor Networks and Applications, San Diego, CA,September 2003.

[42] S. Vi, H. Min, Y. Cho, and J. Hong, "Molecule: An adaptive dynamic reconfiguration scheme for sensor operating systems ", Computer Communications, 3 1 (4), pp. 699-707, 2008

[43] Jingtong Hu, Chun Jason Xue, Yi He, and Edwin H.M.Sha, Reprogramming with Minimal Transferred Data on Wireless Sensor Network, The 6th IEEE International Conference on Mobile Ad Hoc and Sensor Systems (MASS 2009), Macau SAR, P.R.C.,Oct. 12- 15, 2009

[44] Rajesh Krishna Panta, Saurabh Bagchi, and Samuel P. Midkiff, Zephyr: Efficient Incremental Reprogramming of Sensor Nodes using Function Call Indirections and Difference Computation. Annual Technical Conference (USENIX), June 14- 19 2009, San Diego, CA, USA

[45] J. Koshy and R. Pandey. Remote incremental linking for energy-efficient reprogramming of sensor networks. In Proceedings of the second European Workshop on Wireless Sensor Networks, 2005.

[46] Pedro Jose Marron, Matthias Gauger, Andreas Lachenmann, Daniel Minder, Olga Saukh, and Kurt Rothermel, FlexCup: A Flexible and Efficient Code Update Mechanism for Sensor Networks, K. Romer,H. Karl, and F. Mattern (Eds.): EWSN 2006, LNCS 3868, pp. 2 12-227, 2006.

[47] Seung-Ku Kim, Jae-Ho Lee, Kyeong Hur, and DooSeop Eom, Tiny Function-Linking for EnergyEfficient Reprogramming in Wireless Sensor Networks, 2009 Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, pp. 208-213, 2009

[48] Seung-Ku Kim, Jae-Ho Lee, Kyeong Hur, and DooSeop Eom, Tiny Function-Linking for EnergyEfficient Reprogramming in Wireless Sensor Networks, 2009 Third International Conference on Mobile Ubiquitous Computing, Systems, Services and Technologies, pp. 208-213, 2009

[49] S.S. Kulkarni and L. Wang. MNP: Multihop network reprogramming service for sensor networks. Technical Report, Michigan State University, 2004.

[50] Jaein Jeong, David E. Culler. Incremental Network Programming for Wireless Sensors. IEEE Sensor and Ad Hoc Communications and Networks(SECON), 2004:25-33.

[51] Abrach H, Bhatti S, Carlson J, et al. MANTIS：System Support for Multimodal Networks of In-situ Sensors[C]. Proceedings of the 2nd ACM InternationalWorkshop on Wireless Sensor Networks and Applications, 2003: 50-59.

[52] 刘莉,黄海平. 基于 MantisOS 的无线传感器网络应用开发模型[J]. 信息技术,2010,v.34;No.22306:127-129.

[53] Österlind Fredrik, Dunkels Adam, Eriksson Joakim, et al. Cross-Level Sensor Network Simulation with COOJA. Proc of the 31st IEEE Conference on Local Computer Networks. Piscataway, NJ: IEEE, 2006: 641-648.

[54] 徐顶鑫,易卫东. 基于 Contiki/COOJA 平台的 Deluge 协议性能测试[J]. 计算机研究与发展,2011,v.48S2:290-294.