**CONTIKI报告**

**一 环境配置**

鉴于虚拟机的资源浪费以及windows系统下GCC编译的难度以及速度问题，可以选择在UBUNTU系统下面进行开发环境配置。（以下环境设置都在ubuntu12.04下完成）

1. GCC编译器设置

按照以下语句在terminal下进行安装交叉编译器以及必要的环境设置

sudo apt-get install build-essential binutils-msp430 gcc-msp430 msp430-libc binutils-avr gcc-avr gdb-avr avr-libc avrdude openjdk-7-jdk openjdk-7-jre ant libncurses5-dev doxygen git

以上所需安装的都是必要的交叉编译器已经所需的开发工具，比如github管理器。

1. ARM交叉编译文件

ARM交叉编译文件在apt-get文件库中缺失，有两个支持包，我选择了以下：

<https://sourcery.mentor.com/public/gnu_toolchain/arm-none-eabi/arm-2008q3-66-arm-none-eabi-i686-pc-linux-gnu.tar.bz2>

下载文件包，并且在合适的地方进行解压。

进入terminal，用sudo –s打开root权限，进行如用户目录下的./profile 文件的修改，在文件中加入以下语句：

export PATH=$PATH:$HOME/arm-2008q3/bin

修改之后进行文件编译之前就不必进行path的修改。

如果权限不够的话，进入root的话 profile文件在/etc文件下，用gedit编写

1. contiki文件下载

contiki开发包可以再官网下载，但是有鉴于官网的资料有时候滞后，甚至于会有些资料缺失（比如在tool下的cc2538-bsl脚本的缺失），建议从github下进行fork的文件，采用下语句：

git clone git://github.com/contiki-os/contiki.git contiki

尾部的contiki为下载的路径，可以进行自行修改。

此外由于cc2538-bsl文件的github地址不再contiki之下，我们需要执行如下操作：

git clone git://github.com/JelmerT/cc2538-bsl contiki/tools/cc2538-bsl

（注意看github项目上面的wiki）

**二 硬件设备配置**

目前手头有的硬件为CC2538EM片上系统与SmartRF06EB开发板，SmartRF06 EB开发板片上带有XDS100v3烧写系统，因此可以进行基本的开发功能，在UBUNTU系统上请按照contiki系统的文件platform CC2538DK文件夹中的readme文件，进行配置。

在测试过程中发现contiki的在ubuntu下面的文件烧写成问题，按照readme文件中测试也有问题。在github中发现tools目录下有个cc2538-bsl，此文件为PYTHON脚本，可以进行CC2538的烧写功能，但是发觉还是不行。

经过调查这种情况是自带的系统中未将bootloader功能未被开启。需要进行以下步骤进行一次功能的烧写（windows系统下）：

1下载IAR EMBEDDED WORKBENCH IDE与TI公司页面下的swrc272a程序。

2 打开swrc272a文件，搜索setup.c文件将某个结构修改进行如下修改：

\_\_root const lockPageCCA\_t \_\_cca @ ".flashcca" =

{

0xF3FFFFFF, // Bootloader backdoor enabled

0, // Image valid bytes

FLASH\_START\_ADDR // Vector table located at flash start address

};

3编译，对板子上面socket安装参考下面的网址，进入SmartRF Flash Programmer2进行烧写。

http://processors.wiki.ti.com/index.php/CC2538\_Bootloader\_Backdoor

在这个问题上卡了很久，不过还好解决了。

之后进入Ubuntu系统，用指令./cc2538-bsl –help参看指令系统并且进行持续下载。

以上只是进行了bootloader方式的烧写，uniflash的使用尚有问题，待后期进行下研究。暂且放在一边。

Bootloader 问题详述：

在reamde.md中有说明运用：make cc2538dk-demo.upload能够自动用 bootloader方式进行烧写，经过尝试不行，以下是我的使用正确的方法语句(在platform下修改Makefile.cc2538dk中修改类似语句即可，我已经修改过所以能够直接测试成功)

在tools/cc2538dk-bsl下有一个文件：

python cc2538-bsl.py -p /dev/ttyUSB1 -b 115200 -e -w ../../examples/cc2538dk/cc2538-demo.bin

需要注意的是，一是比特率，二是端口问题，第三是文件的路径寻找问题。

此外需要注意的是，因为系统问题，（新建设的系统似乎讲contiki文件权限设置为root了因此需要root下操作,所以出现了以下的问题，但是这个问题除了每次开机麻烦一些之外没有太大影响）PATH会重启之后取消我们设置的编译路径问题，我们需要进入/etc下用source profile进行一次读取文件操作，才会使得PATH正确设置。

类同的重启后系统也会无视我们之前的设置，需要进行一下：modprobe ftdi\_sio vendor=0x403 product=0xa6d1(用ls /dev查看下是否被读取)

（cc2538demo功能我没有正确测试。需要做一个hello-world的功能测试）

hello-world.c测试结果：

使用语句：make TARGET=cc2538dk hello-world.upload 进行编译下载

根据网站下载xgcom <http://www.linuxidc.com/Linux/2014-03/98196.htm>

查看ttyUSB1，速率调整为115200,进行测试，结果成功

**三 Makefile系统解读**

Makefile在对于系统编译极为重要，对Makefile进行解读。

Contiki的Makefile系统包括以下文件：

\* Makefile: 在project工程目录底下，是项目的主要makefile，是编译入口

\* Makefile.include: 全系统主要makefile，在根目录底下，主要进行对嵌入式系统的编译环境的设置

\* Makefile.$(TARGET) ：TARGET指向开发平台名字，在platform具体平台目录下，被Makefile.include所调用，执行一些关于平台的编译设置

\* Makefile.$(CPU) ：CPU指向开发平台所用的微控制器，在CPU中各种微控制器目录下，被Makefile.include所调用，执行与CPU构架有关的编译设置。

\*Makefile.$(APP) ：APP为各种系统提供的调用功能，包括EMAIl等，在各种app文件目录下。

Contiki环境下的makefile系统已经尽量简化，使用起来很方便，后面会提及使用问题，但是需要对makefile工作进行一些说明。Makefile在运行之后，将被调用的其他makefile文件及其所有依赖文件进行一次重新组合，因此在读源代码的时候，需要理解，有些情况下调用头文件有些混论，这是makefile系统对源代码系统重新组合的结果，因此当调用情况下需要一些情况注意下，可以通过解读example对头文件系统进行下总结。可以这么理解，Makefile文件调用Makefile.include并将include所在的文件夹下的所有文件移动至项目文件目录下，Makefile.include继续调用app等Makefile并进行文件移动，最后组成一个完整的系统项目文件数。

Contiki的Makefile文件系统极为好用，在项目文件下使用命令行

make TARGET=($target) DEFINES=

其中TARGET可以将TARGET=cc2538dk这样子的系统平台使用指令放在工程项目文件夹的Makefile.target中，DEFINES类同，这样子情况下可以使得在编译过程中少写后面部分，以方便调试需要特别支持的是DEFINES可以放在Makefile.($platform).defines文件中，并且这些define是contiki-main.c中一些宏定义，可以进行参数的传递，根据平台的不同与依赖不同，也有所不同，根据具体情况进行分析。具体的使用方式还需要进行分析。

以下是基本使用方式

CONTIKI\_PROJECT = （$Project）

all: $(CONTIKI\_PROJECT)

CONTIKI = ../..

include $(CONTIKI)/Makefile.include

其中Project为项目的工程名字，根据工程项目文件夹位置，CONTIKI=../..有所不同，CONTIKI是路径变量，可以根据具体情况下盖，此外，Makefile.target跟Makefile.($platform).defines也可以根据具体需要进行编写。

Contiki的Makefile系统已经相当好用，使用起来也很方便。如果有时间，将会继续研究一遍，只是其中的一些正则式有所不理解，还需要学习。

**四 Core文件夹中的核心代码**

上面都是一些环境以及基本的使用方式解说，下面是一些程序代码方面的问题。

首先我需要解释下Contiki工程的文件组成

* platform： 开发平台所需要的文件及其代码文件
* doc：一些开发文档，虽然我个人觉得写得不是很好。
* cpu: 开发平台依赖的微控制器文件
* examples: 开发者个人的开发目录建议放在这个文件夹底下，此外还有一些使用示例可供参考。
* core：内部是系统的核心，系统的构架文件都在这个文件夹底下
* tools：一些实用的开发工具
* 根目录：根目录有一些重要的readme文件，此外最重要的是Makefile.include文件，是Makefile系统核心文件。

下面谈谈大体的Contiki开发结构：

Contiki在platform中有个文件叫做contiki-main.c这个是本系统的系统入口，里面有main这个程序入口，作者在开发过程中希望开发者不用关注到main的，而是采用protothread跟event这种方式进行系统的开发。大体上，我们只需要在examples下面建立我们自己的工程文件夹。

一 系统的内核构架

Contiki系统依赖于Prototread构架，这种构架很重要的好处就是节省嵌入式芯片的存储空间，网上说的也比较多，也

多加赘述，基本运行结构是：

开发者建立进程-进程被推入运行进程链中-运行进程（进程运行结束,返回exit等信息，后被推出进程链中）-进行运行中触发事件-将事件推入事件链中-运行事件并退出事件链中

需要说明的是，每次主程序进行一次循环中，会都进行一次对事件链的查询跟执行事件中事件链上方的事件。这种结构在使用过程中有一些细节，我也有一些想法等后期进行测试。

**A）核心进程调用体系**

**1 进程中间的中断与延续（LC）**

进程设计依赖于一个叫做Coroutines的设计模式，参看下面地址：

<http://www.chiark.greenend.org.uk/~sgtatham/coroutines.html>

这个设计主要用于在各个Protothread函数中进程间切换，继续，以及等待。通过这个设计当进程需要进行中断运行，等待信息的时候，可以从某一特定运行地点直接运行，而非从头开始。

这个设计在本系统中被称为locale continuation，简写为LC，主要在以下两个文件中（路径:\cor\sys）:  
 \* lc.h

\*lc-switch.h

Coroutines系统请参看所给网址，比较精巧的一种设计。

编写LC过程中需要调用到GCC编译器提供的特殊参数：\_\_LINE\_\_。这个参数主要是负责记录当前运行语句的地址（或者说行数）。这个参数与Coroutines合作，使得进程函数的继续与中断。

基本上LC设计思维被封装入了process操作中，对外部开发者而言不必要关系细节，设计者已经将细节完成的很好了。

**2 protothread**

Protothread的关键文件有以下三个

* pt.h
* process.h
* process.c

pt.h是基于lc结构实现的，主要提供了一些宏的实现，下面所列举有：

1. PT\_INIT(pt)：用LC\_INIT()宏进行参数中的lc参数的初始化。创建了PT\_YIELD\_FLAG，是实现PT\_YIELD一系列宏的关键参数
2. PT\_BEGIN:在process.h中被PROCESS\_BEGIN所封装，进程开始都必须进入一次PROCESS\_BEGIN调用，核心是LC\_RESUME宏。
3. PT\_END:在process.h中被PROCESS\_END所封装，PROCESS\_END进程结束之前必须调用一次，核心为PT\_INIT,LC\_END并返回参数PT\_EDNED。（这里PT\_INIT也从侧面反映了每次进程结束必须退出进程链的处理。）
4. （以下的几个宏都巧妙的运用了return的技巧）PT\_WAIT\_UNTIL（pt,contition）：核心是LC\_SET,并且调用了在条件不符合的时候返回PT\_WAITING,下次进入进程将会从这里直接进入。作用是等待其他进程或者事件返回给合适的参数之后，才进入下面的工作。
5. PT\_WAIT\_WHILE（pt, condition）:本质上是调用PT\_WAIT\_UNTIL宏，只是对状态进行了下处理。
6. PT\_WAIT\_THREAD(pt,thread)：本质上调用PT\_WAIT\_UNTIL,只是将thread用PT\_SCHEDULE进行了处理，起作用是等待与本进程进行交互的进程返会PT\_EXITED参数。
7. PT\_SPAWN：在代码中解释是产生子进程，就我的意见，是将两个进程联系在一起，以父子进程的形式存在。如果是产生一个新的进程，可能在后面的程序中需要调用process\_start()将子进程推入进程链中。（当然如果一开始是autostart就不必要了）
8. PT\_EXIT:调用PT\_INIT()并返回PT\_EXITED，这个宏跟PT\_END不同点在于，返回数值的不同，PT\_EXITED会被其他进程所知悉，并且进行处理。
9. （以下几个操作关键是PT\_YIELD\_FLAG的操作）PT\_YIELD与PT\_YIELD\_UNTIL:根据进程内部的进程的操作对PT\_YIELD\_FLAG进行操作，并且返回PT\_YIELDED参数。

本头文件中的宏定义都采用了do{}while(0)形式是为了保证宏定义中结构的完整性，不会被外部的代码语句破坏代码的实现。大部分本头文件中的宏定义都被process.h中的宏定义所封装，因此使用者不必对pt.h中的定义太过细致的理解。

process.h是进程调度的核心部分，封装了pt.h中的大部分宏定义，并且定义了参数的重要几个运行函数。因此需要重点理解，需要理解宏定义使用规则。（这部分是重点也是难点，需要进行下思考怎么进行说明，而且在运行过程中有很多细节需要注意，有部分理解我还未进行过初步的实验，只是在对代码理解上的推理，比如如果使用process\_start推入，而非autostart的话，需要进行下接收process\_intitail参数的处理，避免process\_start之后，进程被运行又推出进程链。进程与事件的活动也是如此）

以下部分参考博客Jelline的一些资料，解释下与进程，事件相关的概念：

（函数说明可以参看在线sourceforge网站：contiki.sourceforge.net/docs/ 里面有更具体的参数表介绍，我这里不再说明，只是对调度的概念与细节进行表述）

上面说Contiki是进程与事件进行系统的调度，进程触发事件，事件返回事件情况，进程互相通信。因此这里我们必须说明下进程的概念。

进程的本质是个链表，每个进程根据所需占用一定的内存区块，通过链表的方式进行链接，这样子情况下能够使得每个进程随着自己的需要进行内存的占用，对于内存区块较小的处理器而言，比较有优势。事件也采用了如此的策略，一切策略都基于内存受局限的处理器优化。

总体而言两者的都是一致的策略方式，以链表结构，并且内部都是相似的数据搭配。唯一的不同是在系统调用过程中的处理方式与优先度。

进程块结构：

struct process {

struct process \*next;

#if PROCESS\_CONF\_NO\_PROCESS\_NAMES

#define PROCESS\_NAME\_STRING(process) ""

#else

const char \*name;

#define PROCESS\_NAME\_STRING(process) (process)->name

#endif

PT\_THREAD((\* thread)(struct pt \*, process\_event\_t, process\_data\_t));

struct pt pt;

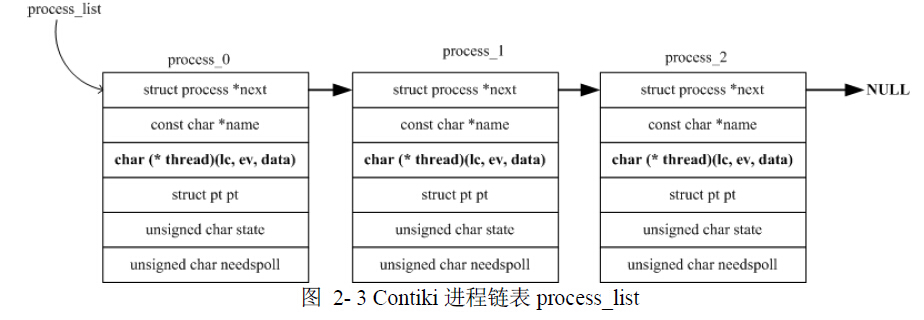
unsigned char state, needspoll;

};

结构中\*next指向下一个进程，后面程序中可以理解，越新推入进程链的进程越快执行，其中PT\_THREAD是个宏，最后形成一个本进程中的执行函数，这个函数在执行过程中需要输入三个参数：第一个参数是进程运行的lc位置，第二个是传递过来的事件标示，在文件前面定义过一些通用的事件标示，也可以由其他进程申请，然后传递过来，第三个是进程运行过程中需要的参数，接受任何类型的参数。返回值为一个char字符。

state指出进程运行的状态，相对而言needspoll更为重要，如果这个参数不为1，即使在进程链中，也不会被执行，知道他的needspoll被修改。

在运行过程中结构如下：



以上是基本的结构，在使用过程中，用户只是定义跟启动一个个进程，而不需要关心while（1）里面的具体操作，下面对用户怎么定义并启动一个进程。

我们知道一个进程需要一个process结构，并且重点是定义char （\*thread）(lc,ev,data)，用户不必关心后台怎么运行只要关注与怎么设计此运行函数。

在实现进程的宏定义套用很多宏定义，我们再实现过程中只需要进行这么一些结构。

#include "contiki.h"

#include "debug-uart.h" /\* For usart\_puts()\*/

#include <stdio.h> /\* For printf() \*/

PROCESS(hello\_world\_process, "Hello world"); /\*参照二\*/

AUTOSTART\_PROCESSES(&hello\_world\_process); /\*参照三\*/

/\*Define the process code\*/

PROCESS\_THREAD(hello\_world\_process, ev, data) /\*参见四\*/

{

    PROCESS\_BEGIN(); /\*参照5.1\*/

    usart\_puts("Hello, world!\n"); /\*向串口打印字符串"Hello,world"\*/

    PROCESS\_END(); /\*参考5.2\*/

}

PROCESS宏定义了一个process结构体与process结构体对应的程序，并在PROCESS\_THREAD()这个宏定义中进行了函数的实现，这个函数的实现以PROCESS\_BEGIN()开启，并以PROCESS\_END()结束，期间进行代码的进行实现。

AUTOSTART\_PEOCESS()是推入进程的两种方式，可以用process\_syn\_post()也就是process\_start()进行手动的启动。

宏定义实现比较复杂，请参看Jelline跟源代码核对，除了需要注意下##name这种宏定义的技巧。（他们代码的结构跟规范真好看。）

类似的事件也是采用相同的策略，唯一的不同点在于事件都是在进程中触发，通过process\_post()操作进行事件链的推入，从目前分析结果而言，事件中进程必须在进程链中才能启动，这点还需要进一步实验证实（可商榷）。

事件的结构如下（从结构体分析可以得知，其实是在process基础上通过process\_alloc\_event()分配了一个process\_event\_t ev参数以及生成此事件的process所供给的data）：

struct event\_data

{

process\_event\_t ev;

process\_data\_t data;

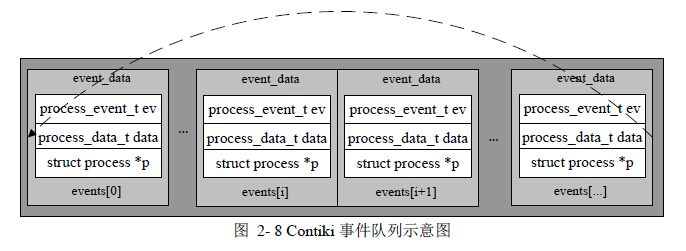
struct process \*p;

};

typedef unsigned char process\_event\_t;

typedef void \*process\_data\_t;

运行时候需要用process\_post\_synch()推入事件链中，运行时候如下：



进程与事件最大的区别在于，在每次while（1）过程中会对所有进程进行轮询，运行结束的进程将会被推出进程链中，而事件只对事件链最上方的事件进行操作，操作结束推出进程链，以此类推。如果用process\_post\_synch模式进入的话，不会进入事件连接中，而是直接进行运行，从这个角度而言个人并不倾向把这种模式认为事件，但尊重编者的想法。  
 在开始进入进程中的关键函数的之前，有一些零碎的宏定义需要说明，上面说到process.h封装了大部分pt.h的宏定义并且提供了一些定义。

新添加的定义有一部分是参数的定义，指定了一些参数，作为事件标示，这部分可以容易查看，不加以描述。

特意说明下：

#define PROCESS\_EXITHANDLER(handler) if(ev == PROCESS\_EVENT\_EXIT) { handler; }

上面在某些情况下，进程还需要特别处理一些其他事件信息进行对应，以免还未进入实际上的工作就被推出进程链，当进程发出PROCESS\_EVENT\_EXITED参数的时候，会向进程链通告，这种情况下，进程必须准备一个处理这个事件信息的方案。PROCESS\_EXITHANDLER（）可以接受一个handler，就思考而言，这个handler可以是一个宏定义的代码块，也可以是一个函数，看具体操作。注意及时利用lc操作以及return退出函数进程，以免对后面的程序造成影响。类似handler处理还有一些，不细讲。

process.h文件中实现了一些对外函数，但是关键性的一些函数也在process.c中实现因此我想在这里一起说明。

（进程的进入需要 start - 此外如果重新启动睡眠的程序我们需要poll他）

参与进程运作的一些关键函数：

从上至下的分析，在main（）中while（1）调用了process\_run()函数，此函数调用了do\_poll(),do\_event，前者对进程链进行处理，后者对进入事件链即由process\_post()推送的事件处理。

do\_poll():本函数对进程链中的进程进行轮询，进入是就修改poll\_requested=0;并进入进程链的处理，只有当进程的needspoll为1时候才会进行进程的处理，一旦执行则会置零，因此这点我们需要特别注意下。以下是理解needspoll作用的例子，放在这里有点不合适，但是我害怕我自己忘记了。

比如在etimer\_process\_poll就进行了置一操作，这个例子在两个进程间通信的操作。参看这个网址：<http://blog.chinaunix.net/uid-9112803-id-2974353.html>。历程结构可以如下：

两个进程被推入进程链中间，但是needspoll=0；因此不会进入进程实现操作，当etimer溢出的时候，将会调用process\_poll()函数对函数进行needspoll参数操作，然后进程链中的：PROCESS\_THREAD(print\_hello\_process, ev, data)会被调用。产生事件：PROCESS\_THREAD(print\_world\_process, ev, data)；以此重复，这样子情况下，进程链不会重复调用一些处在僵尸状态的进程，也能够适应随着不同触发条件触发不同事件以应付不同的系统需求的目的。

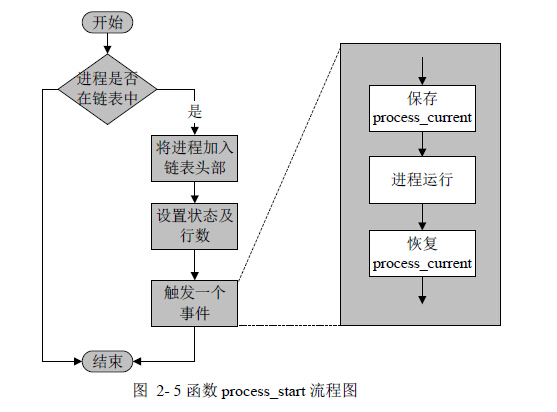
对needspoll的讨论也从侧面证实hello-world中实际上并未进入进程链中，因为他的needspoll并未真的被置一。

Needspoll的操作可能重要性比state重要。

回归到另一个重要函数：do\_event(0:

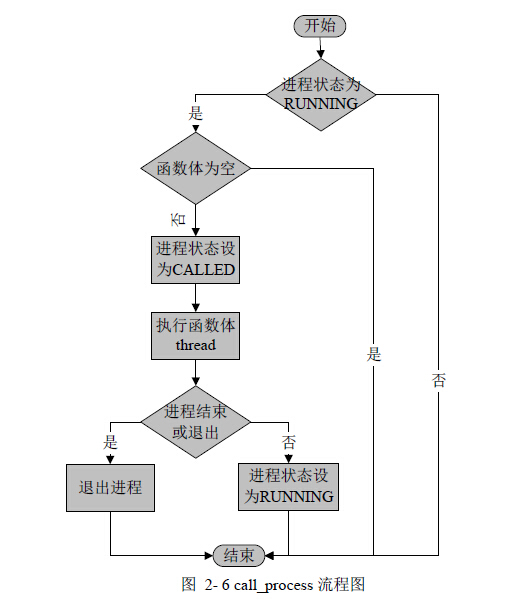
只是调用链表最上层的事件并进行操作，与do\_poll的对整个进程链进行操作有所不同。需要注意的只是传递某些特定事件的时候么，如PROCESS\_BROADCAST(实际值为NULL)时候需要执行某些特殊操作。

1. process\_start：是进程中的关键函数，每个进程进入进程链过程中必须调用，我们知道AUTOSTART\_PROCESS()本质上也是调用了此函数，只是autostart的方式已经在主程序过程中写好了。以下是process\_start()函数的解析;

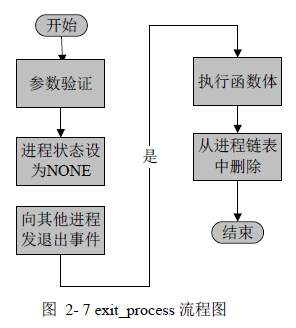


本函数将需要用链表的方式将事件推入事件链中，其中特别需要注意的是这种推入类似于堆栈的模式，后入先执行。在第三步中执行将会呼叫process\_post\_synch()事件调用方式，传递事件标示：PROCESS\_EVENT\_INIT,上文说的是process\_post\_synch()的事件并不直接进入事件链中而是直接执行事件，call\_process()是具体执行核心函数，也有一些执行要点。后面将会说明。

1. call\_process():call\_process作为核心函数，实际上进程的具体执行就在这个函数中实现。从代码上面看执行也比较简单，校验一些state已经某些限制条件之后就运行具体的进程运行代码（ret = p->thread(&p->pt, ev, data);），不过需要注意的是之后有个if代码块，从这个代码块可以得出一个信息是，在复杂的系统中，我们需要特别注意下需要当传递过来PROCESS\_EVENT\_INIT事件后做下处理，否则在process\_start模块就会执行，而在进程链执行过程中将会什么都没有。（比如例子中的hello\_world中，实际上在代码进入进程链之后立即被执行之后退出进程链，实际上主函数在执行空进程。这种情况下需要特别的这注意，因为有时候系统就会出现你只是执行了一次之后就不执行的情况，解决方式是1 用handler的方式进行处理， 2 用while（1）的形式）特别的进程会在执行结束后呼叫exit\_process()函数，进行从进程链中的退出。执行逻辑如下：



1. exit\_process():进程操作中，这个函数其实是个很重要的一个部分，执行过程中过程中一些细节，首先看下执行逻辑：



需要注意的是，exit\_process()参数表中有两个，当两个参数一致，即为从本进程中退出，则可以省去执行函数体这个步骤。而不一致的时候意味着从调用函数的进程中向其他的函数发出退出的指令，则在编写目标进程过程中预留出处理这方面的进程块。向其他进程发退出事件时候，将会执行一次其他进程的函数体，从这个部分可以得出，一旦有进程会发出进程退出的事件的时候，我们需要进行这个块的处理，否则将会使得其他进程的函数执行，使得程序跑飞。（有时候他们命名比较细致，有些地方不容易分辨，比如PROCESS\_EVENT\_EXIT与PROCESS\_EVENT\_EXITED，我在中间可能写错一些）

d） 在这里我们不在赘述其他的一些相对不重要而简单的函数，如：process\_exit（）；process\_init()；process\_poll（），这些函数

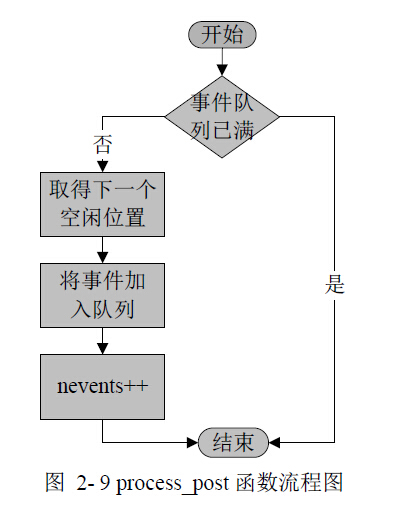
在这里我们进行一次总结，needspoll,以及在process\_start()等操作中有许多细节，如果只是简单的执行一两个hello-world我们不需要特别在意，当进程已经进入一个等级的复杂度之后，进程间的互动调用等都需要特别的小心谨慎。

本文件中还有几个事件产生与调度的函数：

我们知道事件会分配一个事件标志，除了已经在文件中定义过的一些事件标志，用户可以使用process\_alloc\_event()函数申请一个事件标志。

事件产生有两种方式，一种是同步，另一种是异步产生，同步是process\_post\_synch()，这种方式在上面带过，就不再赘述。另一函数是process\_post()这里重点讲讲这个函数。

这个函数运行结构如下：



事件队列的大小可以又文件内部的事件大小参数进行修改，简单来看，推送间事件链的是一个栈的方式，但是实际上是个循环队列的方式，亦即在不断推送事件的过程中未执行的事件数nevent与执行到的参数fevent的和是相同的，当达到上限的时候，会从event[0]重新覆盖起，这样子需要注意，每次循环中只是执行一次事件，需要担心事件未执行之前被覆盖。process\_post()函数只是执行一个推送的过程，需要关注的是，事件与进程是挂钩的，观察事件的结构可以得知，事件的实现基于进程，此进程可以是产生事件的进程相同，也可以不同，目前我还存在一个疑问是事件是否是事件链中的事件，存疑，待证实。

Process头文件与代码文件是本系统系统调度的核心，需要花上时间去了解。类似subprocess.h之类的，这个文件比较简单，而且易懂，重要性也不大。

process\_post():只是简单的推送一个进程进入事件内

**B）定时器**

定时器对于一个系统而言极为重要，Contiki基于事件驱动与Protothread机制，事件可以是外部时间也可以是内部中断，对于系统调度而言，定时器很重要。特别的对于通信功能而言，其重要性不言而喻。

**1 clock.h**

一系列定时器功能（timer.h,etimer.h等）都是基于clock.h文件的功能，这个文件是系统较为底层的部分，是系统与平台的交界文件，总体思路是使用平台芯片自带的定时器功能，利用中断功能进行技术，计时时间一旦开机即开始。（当然基于不同的平台，有不同的驱动策略，例如有些芯片提供精确的计时功能，或者计时模块，这时候我们直接提取即可。）这部分基于平台的特异性，不同的平台的实现情况与策略有所不同，由于我们在使用系统过程中不必要进行更底层的分析，或许后期我会进行下更仔细CC2538平台的clock.c分析，但是目前为止没有必要。只是做一个大体功能上面的分析。

clock.h提供了维持上层功能的一些函数，大部分情况下我们不必要对这些功能进行使用，更多的使用系统提供的timer函数与文件：

1. void clock\_init(void):clock系统的初始化函数，这个函数必须在主函数中被调用，以用来支持其他的timer函数的使用。
2. clock\_time\_t clock\_time(void):返回平台的目前计时器时间，从目前信息看来，返回的是系统的tick计数器的值。
3. unsigned long clock\_seconds(void):返回系统的秒数值，是一个计数的全域参数，而非类似于YY:MM:DD时间形式。
4. void clock\_set\_seconds(unsigned long sec):设定秒数值，就我目前的思路，我猜测是使得系统设定特定的时间点。
5. void clock\_wait(clock\_time\_t t):等待系统的tick值。
6. void clock\_delay\_usec(uint16\_t dt):通过中断的方法，延迟特定的microsecond数。以下三个在大部分的平台下不会使用，并没有实现，可能是为了后期系统的发展而预留的接口。
7. int clock\_fine\_max(void):
8. unsigned short clock\_fine(void):
9. void clock\_delay(unsigned int delay):

这部分与平台相关性很大，有鉴于我们在使用过程中暂时不必要对更为底层的代码进行研究，这部分就暂时先分析道这里。

**2 timer**

Timer文件作为Contiki计时系统的重要文件，基于底层的clock系统，而且其他三种时间类型都基于这个基础之上，在注释中间也说到，我们需要显示的调用这个文件以使得系统的计时功能正常工作。正如文件中说明，一共是四种功能：设置，重置，重启，以及检测是否时间到期。这种到期必须手动的检测，从这个角度以及etimer的经验，我认为后面三种时间模式这是必须依附于event系统或者process结构进行操作。这是后话。

这里附注下CCIF，我查了下，这个是Contiki系统下私有的一个申明，为“Contiki Core Interface”的缩写，需要动态加载程序的环境下，具体而言只在win32环境下有需要，可以查看win32平台下的contiki-conf.c文件。

请参看<http://www.abbreviations.com/CCIF>。

回到timer文件上面来。timer.h文件中定义了一些外部接口，但是最为重要的是定义了timer结构，其中有两个成员两边，一个起始时间点，另一个是间隔距离。

在timer.c文件中，具体实现如下：

首先是void timer\_set()函数，用以进行timer结构的赋值初始化，需要调用clock系统中clock\_time()，赋予目前时间点，用于记录此结构的导入系统时间点。

其实timer\_reset()与timer\_restart()可以放在一起来说，他们都是对timer结构中起始点进行赋值，一个方式是在timer->start基础上叠加interval的数量，后者则是重新调用了系统当前的时间点，唯一的区别是后者会导致一些周期性调用的功能的时间便宜。因此可能用reset方式在这种情况下合理性更好。

int timer\_expired(),返回一个判断，看是否已经超过了间隔时间长度了，不过里面为了避免mspgcc的一个编译上面的问题，采用了一个比较奇怪的写法。

此外系统还提供了一个timer\_remaining(),计算还剩下多少时间即逾期。

整体操作比较简单，代码难度很小，主要是基于clock系统的函数而实现的。需要注意的只是timer结构。

系统还提供了stimer，结构跟接口都跟timer一样，唯一区别的是他使用的是clock\_seconds（）也就是秒数而非tick进行计时与表示，这些接口使用方式跟timer一样，因此只是在这里提一句而不仔细介绍。

**3 etimer（后期修正事项：对函数的参数没有仔细说明，有时候说明不够到位）**

Etimer意味着事件定时器，event timer的缩写，每个事件与一个process结构挂钩，每次通过timer逾期的查询，会触发一个事件，进入事件链中等待处理。某一个由上面一节对timer的分析可以得知，无论是什么时间逾期，系统都不会主动上报系统，需要通过事件手动查询，因此etimer采用的是通过加载一个etimer\_process的进程，这个加载在contiki-main.h中有定义。在进程中对各个etimer链中的定时器进行查询是否逾期，进行操作。

经过分析可以得知etimer采用的是链表结构，因此首先是etimer中定义了etimer的结构这种结构基于timer结构与process结构，用链表进行构架：

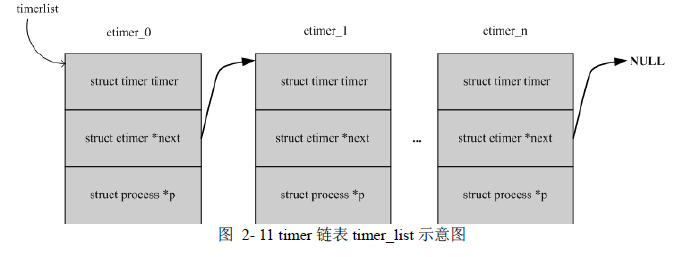
struct etimer {

struct timer timer;

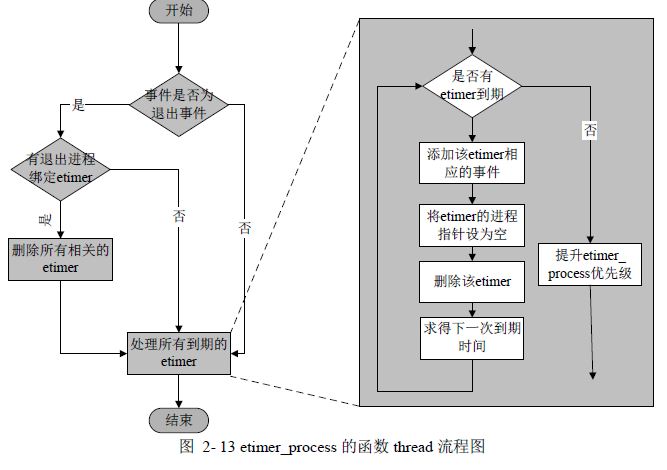
struct etimer \*next;

struct process \*p;

};



我们可以从这个进程中进行分析，实际上这个进程也是很经典的一个使用模板，例如遇见PROCESS\_EVENT\_EXITED如何处理，事件需要重复poll的话，我们需要如何修改needspoll参数等。



代码中合理的运用了while（1）与PROCESS\_YIELD()宏定义，这使得代码的运行极为有效率。其次，etimer\_request\_poll其实是调用process\_poll()函数接口，对etimer函数进行不断的提高优先级。这个在上面我们也分析到了

此外需要对etimer提供的一些外部接口与内部的函数进行分析。

文件内部有一个函数叫做update\_time()，这个函数主要是更新next\_expiration参数的，这个参数存储最接近expired状态的etimer计时器所即将expired的时间长度。

通过查看内部的函数add\_timer（）可以得知，与推送事件不同，每个etimer都与调用它的进程想挂钩，即etimer->p=PROCESS\_CURRENT();这个函数的结构是先调用etimer\_process\_poll()提高etimer的优先度，然后进入检测是否etimer是否重复，没错的话将etimer推入etimer链中。

调用add\_timer()有三个函数：etimer\_set(),etimer\_reset();etimer\_restart()这三者，最有区别的是etimer\_reset()跟etimer\_restart()，他们分别基于timer\_reset()跟timer\_restart()如果使用周期性的调用的话，可能reset的方式更为合用。

除了与timer文件相同的etimer\_set()等，其他需要注意的是，etimer\_expiration\_timer()返回了下一个即将expired的计时器，如果etimer链中无数据，则返回为了0。此外，etimer\_expired()向外提供了一个检测某个计时器是否expired并且处理过事件，因为每次处理过计时器触发的事件之后，都会讲计时器中的进程结构清为NULL。

总结下，etimer通过在进程中挂载一个etimer的事件，每次检测之后都进行是否expired的检测，expired的计数器触发事件并且推出计时器链中，并且将此计时器的进程结构清零。在例子：

<http://blog.chinaunix.net/uid-26893932-id-3378337.html>

就很好的展示了怎么使用etimer，怎么使用etimer进行事件的触发。

**4 ctime**

ctimer意为callback timer，翻译过来是是回调函数计时器。从官网的sourceforge来看这个文件提供了一种回调函数的调用机制，一旦函数expired就会使得函数进行一次函数回调。

在开始讨论ctimer之前可能需要对lib文件夹下的list系统进行分析，因为ctimer调用这个系统中的一部分函数。List是Contiki系统中对于链表的提供函数，系统并不复杂。

首先list.h中提供了几个宏定义，通过这些宏定义可以进行设定一个链表的表头。

抽取宏定义相关代码以及宏展开之后：

typedef void \*\* list\_t;

#define LIST\_CONCAT2(s1, s2) s1##s2

#define LIST\_CONCAT(s1, s2) LIST\_CONCAT2(s1, s2)

#define LIST(name) \

static void \*LIST\_CONCAT(name,\_list) = NULL; \

static list\_t name = (list\_t)&LIST\_CONCAT(name,\_list)

typedef定义了一个list\_t，这个void参量设置使得链表能够接收一切形式的链表成员，并且赋给合适的结构。需要注意的是，list\_t指针的指针，也就是说，这种方式下，链表需要接收的是一个指针参数。

LIST\_CONCAT2与LIST\_CONCAT()实际上只是一个功能两重封装，从后面看了这个文件中只是利用了为了形成链表的名字，形成$(name)\_list这种链表名字，生成一个链表的表头。

相同的效果是

#define LIST\_STRUCT(name) \

void \*LIST\_CONCAT(name,\_list); \

list\_t name

从注释来看是为了在结构中插入链表结构而用的，用LIST\_STRUCT\_INIT()宏定义初始化，在ctimer中不使用。但是这是为帮助使用者定义符合本文件的结构体而使用的宏定义。即形成一个第一参数指向同类型的指针。

本文件中提供了以下函数，其功能极其使用方式很容易理解，可以再网站中查询，就不必要在此分析。

void list\_init(list\_t list);

void \* list\_head(list\_t list);

void \* list\_tail(list\_t list);

void \* list\_pop (list\_t list);

void list\_push(list\_t list, void \*item);

void \* list\_chop(list\_t list);

void list\_add(list\_t list, void \*item);

void list\_remove(list\_t list, void \*item);

int list\_length(list\_t list);

void list\_copy(list\_t dest, list\_t src);

void list\_insert(list\_t list, void \*previtem, void \*newitem);

void \* list\_item\_next(void \*item);

分析下代码，其中有个关键性的问题，在list.c中定义了一个结构list：

struct list {

struct list \*next;

};

这种结构会被反复的对一些item进行类型的转换，因此需要注意的是，作为结构中的item必须是一个结构体，并且第一个结构体是指向同等类型的指针。比如下面的ctimer结构体的第一个参数就是struct ctimer \*next;，这就符合了规范。（这块是重点，不理解的话就无法理解文件中的强制转换以及会导致运行出错）。

实际上这种链表结构就是我们的进程链使用的提取出来一些动作形成的。只要理解上一段的解释，这个文件就不必要继续细细说明。（标注：回头去看这块的指针使用，结构体的使用也需要想想，特别是类型转换过程中，）

理解了list（即链表操作）之后，可以进入ctimer的分析，ctimer基本构架跟etimer相近，起独有的结构体是基于etimer结构体，在符合list的要求上形成的，其结构体如下：

struct ctimer {

struct ctimer \*next;

struct etimer etimer;

struct process \*p;

void (\*f)(void \*);

void \*ptr;

};

可以看出，ctimer符合list的结构需求，其中有一个etimer结构体，跟一个进程挂钩，提供了一个函数参数，这个就是回调的函数，以及回调函数需要的参数指针。

本文件中的结构可以看出，函数定义了一个Ctimer process进程，进程在初始化过程中在头部调用了etimer\_set（）结构，推送至etimer的轮询中，第一次循环中的推送其实是一个空的etimer结构因此可以（不过可能还需要进行一些测试，以防真的有作用，我的想法是这个可能是执行个空的表头。），可以查看ctimer\_set()可以得知，每个ctimer的etimer成员都与ctimer\_process相挂钩。每次ctimer的etimer结构逾期，则会推送事件至ctimer\_process中，检测ctimer中的结构etimer确定是那个ctimer逾期，并进行回调函数的调用，并退出ctimer链中。需要说明的是，其中用了PROCESS\_CONTEXT宏定义对，这是系统对换了背景进程，因为回调的函数可能会对进程有所使用，类似于ctimer中etimer需要调换PROCESS\_CONTEXT以使得etimer挂载在ctimer\_process上。剩下的一些接口函数比较简单，需要的事项也少。因此不做介绍，需要的话请参看contiki的sourceforge文件

此外进程中采用了while（1）嵌套PROCESS\_YIELD\_UNTIL()方式等待etimer轮询推送。由于ctimer进程不需要每次都执行，因此needspoll不需要置一。

如果需要对函数进行理解请参看下面的地址，里面有相应的ctimer的例程

<http://contiki.sourceforge.net/docs/2.6/a00012>

）嵌套PROCESS\_YIELD\_UNTIL()方式等待etimer轮询推送。由于ctimer进程不需要每次都执行，因uart uart使用了0号口，我们你需要注意的是，uart的输出只是单个输出，特别的接受中断的时候接受了一个程序指针，这样接收的程序如下(进程推送)

<http://www.ibr.cs.tu-bs.de/projects/mudtn/doxygen/a00191_source.html>

接口为serial-line，开放的进程端口并不是很好用，建议重新写一个对外的串口接口。

然后看门狗会一段时间内进行重新挂载进程亦即重启，这点笔记好玩