**接口(interface)**

NesC程序主要由各式组件（component）构成，组件和组件之间通过特定的接口（interface）互相沟通。一个接口内声明了提供相关服务的方法（C语言函数）。例如数据读取接口（Read）内就包含了读取（read）、读取结束（readDone）函数。接口只是制定了组件之间交流的规范，也就是通过某一个接口，只能通过该接口提供的方法实现两个组件之间的交流。但是接口终归只是接口，只是一组函数的声明，并为包含对接口的实现。例如以下便是读取接口的代码：

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gifinterface Read<val\_t> {  
    
  command error\_t read();  
  
    
  event void readDone( error\_t result, val\_t val );  
}

只有函数的声明，但是没有函数体，所以需要有一个组件来实现（implementation）这个接口。实现某一个接口的组件，称之为提供者（provider），而使用该接口进行对话的，称之为用户（user）。接口内的函数分两类，一类为命令（command），另一类为事件（event）。用户可以呼叫某一组件提供的接口命令，然后等待相应的事件。简单的假设就是：组件A提供了Read接口以便其他组件与之对话，组件B呼叫组件A的Read接口的read命令来读取某一个数据，例如温度，然后等温度读取完毕之后，系统返回一个readDone（读取结束）的事件给组件B。

**组件（component）**

NesC程序由组件构成。组件内主要是包含了对各类接口的使用（uses）和提供（provides）。例如组件A提供了Read接口，那A就需要负责实现Read接口内的read命令，也就是read命令的函数体，即“具体这个值是如何读取出来的”。因为命令（command）是由接口的提供者（provider）负责实现的。如果组件B使用了A提供的Read接口，那在读取数据结束以后，系统会返回给B一个“读取结束”的事件，而B则需要负责处理这个事件，即“数据读取完毕以后，我用这个数据干什么”，将值返回给计算机，或者是通过无线发送给其他传感器等等，所以事件（event）是由接口的使用者（user）来负责实现的。

组件分为两类。分别是模块（module）和配置（configuration）。模块内包含了程序的主干，也就是对各类命令和事件的实现，是NesC程序的可执行代码的主体。而配置则是负责将各个模块，通过特定的接口连接起来，其本身并不负责实现任何特定的命令或者事件。

以TinyOS附带的Blink（闪烁发光二极管）程序为例：

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif// BlinkC.nc  
#include "Timer.h"  
  
module BlinkC @safe()  
{  
  uses interface Timer<TMilli> as Timer0;  
  uses interface Timer<TMilli> as Timer1;  
  uses interface Timer<TMilli> as Timer2;  
  uses interface Leds;  
  uses interface Boot;  
}  
implementation  
{  
  event void Boot.booted()  
  {  
    call Timer0.startPeriodic( 250 );  
    call Timer1.startPeriodic( 500 );  
    call Timer2.startPeriodic( 1000 );  
  }  
  
  event void Timer0.fired()  
  {  
    dbg("BlinkC", "Timer 0 fired @ %s.\n", sim\_time\_string());  
    call Leds.led0Toggle();  
  }  
   
  event void Timer1.fired()  
  {  
    dbg("BlinkC", "Timer 1 fired @ %s \n", sim\_time\_string());  
    call Leds.led1Toggle();  
  }  
   
  event void Timer2.fired()  
  {  
    dbg("BlinkC", "Timer 2 fired @ %s.\n", sim\_time\_string());  
    call Leds.led2Toggle();  
  }  
}

http://images.cnblogs.com/OutliningIndicators/ContractedBlock.gif代码

//BlinkAppC.nc  
configuration BlinkAppC  
{  
}  
implementation  
{  
components MainC, BlinkC, LedsC;  
components new TimerMilliC() as Timer0;  
components new TimerMilliC() as Timer1;  
components new TimerMilliC() as Timer2;  
  
  
BlinkC -> MainC.Boot;  
  
BlinkC.Timer0 -> Timer0;  
BlinkC.Timer1 -> Timer1;  
BlinkC.Timer2 -> Timer2;  
BlinkC.Leds -> LedsC;  
}

Blink程序由两个组件构成。BlinkC.nc为模块，BlinkAppC.nc为配置。

在模块BlinkC的声明内（module BlinkC {...}）内表明了该程序需要用到的全部接口。因为Blink程序的主要目的是将TelosB传感器上的三盏LED发光二极管以不同的频率闪烁。所以我们需要三个精度为毫秒（TMilli）的计时器接口（Timer），分辨使用as关键字重命名为Timer0，Timer1和Timer2。既然需要点亮发光二极管，自然需要一个操控发光二极管的接口，也就是Leds，最后就是程序启动负责初始化的接口Boot。接下去在实现部分（implementation {...}）。在实现部分需要实现所有我们用到的接口的事件，以为在这个程序里面，我们只是使用了接口，而作为这些接口的用户，我们只需要负责去实现他们的事件。这些接口内的命令，则由接口的提供者负责实现。

这里主要是两个事件，一个是Boot接口的booted事件，另一个是计时器被触发的fired事件。在booted事件中，也就是程序启动以后，我们的主要任务就一个，启动三个计时器：

call Timer0.startPeriodic( 250 );  
call Timer1.startPeriodic( 500 );  
call Timer2.startPeriodic( 1000 );

0号的频率为4Hz，1号的频率为2Hz，2号的频率为1Hz。这里startPeriodic是一个启动计时器的命令，呼叫命令需要使用call关键字。同样，因为是命令，所以它们由接口的提供者负责实现，我们只负责使用就可以了。另一个需要我们处理的事件就是计时器的触发，因为有三个计时器，所以需要书写三个触发事件：

event void Timer0.fired()  
{  
dbg("BlinkC", "Timer 0 fired @ %s.\n", sim\_time\_string());  
call Leds.led0Toggle();  
}  
  
event void Timer1.fired()  
{  
dbg("BlinkC", "Timer 1 fired @ %s \n", sim\_time\_string());  
call Leds.led1Toggle();  
}  
  
event void Timer2.fired()  
{  
dbg("BlinkC", "Timer 2 fired @ %s.\n", sim\_time\_string());  
call Leds.led2Toggle();  
}

先不关心dbg的那一行。我们可以看到0号计时器触发的时候，我们切换0号发光二极管的状态（如果是亮的则熄灭，如果是灭的则点亮）；1号计时器触发时则切换1号发光二极管；3号计时器同理。同样的道理，led0Toggle，led1Toggle和led2Toggle属于Leds接口的三个命令，只管用call呼叫使用便可。

接下去是看配置BlinkAppC。这个组件本身并不使用或者提供任何接口，所以在其声明部分为空（configuration BlinkAppC{}）。而在其实现（implementation）部分则需要实现对组件的连接。因为BlinkC模块使用了Boot、Leds和Timer接口，所以必须指明这些接口都是由其他哪些组件提供的。所以：

components MainC, BlinkC, LedsC;  
components new TimerMilliC() as Timer0;  
components new TimerMilliC() as Timer1;  
components new TimerMilliC() as Timer2;  
  
BlinkC -> MainC.Boot;  
  
BlinkC.Timer0 -> Timer0;  
BlinkC.Timer1 -> Timer1;  
BlinkC.Timer2 -> Timer2;  
BlinkC.Leds -> LedsC;

先使用component关键字标明，这个程序当中，总共要用到哪几个组件。其中包括我们自己编写的BlinkC模块。还有负责提供Boot接口的MainC组件，负责提供Leds接口的LedsC组件。还有提供Timer接口的TimerMilliC，其属于泛型（generic）配置，支持被实例化。这里先不细说，因为我们需要用到三个计时器，所以需要使用new关键字创建三个计时器的实例，然后分别用as被重命名为Timer0、Timer1和Timer2。

再往下就是组件之间的连接了。BlinkC使用了Boot接口，而MainC正好提供了BlinkC所需的Boot接口，所以我们将他们进行连接。箭头所指方向为从使用者指向提供者。

BlinkC->MainC.Boot  
// 或者像下面这样也是可以的。  
MainC.Boot<-BlinkC

因为BlinkC内部就使用了一个Boot接口，所以BlinkC后面的Boot被省略了。完整的书写格式为：

// 意为：Blink组件内使用的Boot接口由MainC组件提供。  
BlinkC.Boot->Mainc.Boot

接着是控制发光二极管的Leds接口，由LedsC组件提供。这里也进行了简写，完整的书写格式为：

BlinkC.Leds->LedsC.Leds

计数器的连接同理。

通过Blink程序，可以帮助我们理解NesC程序的构成和编程思路。这理解当然还有很多其他的技巧。这篇随笔并不是定稿，随着学习研究的深入，还会不断继续完善。