目 录

第一章 前言

1 TinyOS 的安装

TinyOS 有两种安装方式，一种是使用安装向导自动安装，另一种是全手动安装。不管使用哪种方式，都需要安装相同的 RPM。（RPM 即 Reliability Performance Measure，是广泛使用的用于交付开源软件的工具，用户可以轻松有效地安装或升级 RPM 打包的产品。）

一、在 Windows 平台下下载和安装 TinyOS 自动安装程序

TinyOS 自动安装程序下载地址为：

http://webs.cs.berkeley.edu/tos/dist-1.1.0/tinyos/windows/tinyos-1.1.0-lis.exe。TinyOS1.1.0 安装向导提供的软件包有：

-TinyOS1.1.0

-TinyOS Tools 1.1.0

- NesC 1.1.0
- Cygwin
- Support Tools
- Java 1.4 JDK & Java COMM 2.0
- Graphviz
- AVR Tools
  - avr-binutils 2.13.2.1
  - avr-libc 20030512cvs
  - avr-gcc 3.3-tinyos
  - avarice 2.0.20030825cvs
  - avr-insight cvs-pre6.0-tinyos

用户可以选择“完全”安装和“自定义”安装两种类型之一。完全安装包括以上所有内容，而自定义安装允许用户选择自己需要的部分。

安装的粒度是单个的包。例如，用户可以选择安装 avr-binutils，而不选择 avarice。模块的选择可以通过模块树对话框进行。

用户需要选择一个安装目录。所有选择的模块都会安装在这个目录下。以下称这个安装目录为 INSTALLDIR。

1、JDK

如果用户选择安装 JDK 模块，则会弹出一个对话框问是否阅读了 Sun 的版权声明等内容，若用户选择“No”安装将结束；否则安装进出继续。

如果用户没有选择安装 JDK，则安装程序将执行两项检查：

（1）查找 1.4 版的 JDK：安装程序在注册表中查找 HKEY_LOCAL_MACHINE\Software\JavaSoft\Java Development Kit\1.4\JavaHome 表项，如果存在检查通过；否则，建议安装一个正确的 JDK1.4。

（2）查找 java COMM：如果找到 JAVA_HOME\lib\javax.comm.properties 文件则检查通过（上一不检查通过后 JAVA_HOME 就会被设置好）；否则建议安装 java 的 COMM 包。

2、Cygwin

如果用户选择安装 cygwin，那么 cygwin1.1.0 包中的所有内容将会被复制到 INSTALLDIR/cygwin-installationfiles 目录中。同时 setup.exe 将被调用，将那些文件执行自动安装

到 INSTALLDIR/cygwin。

如果用户选择不安装 cygwin，安装程序将在注册表中查找表项 HKEY_LOCAL_MACHINE\Software\Cygnus Solutions\cygwin\mounts v2\（'/'安装点－'/' mount point）以定位 cygwin，若找不到，安装程序将放弃安装，因为包含 RPM 的所有部分都需要 cygwin。因此 cygwin 是必不可少的。

为了将模块 RPM 放置在正确的地方，必须弄清楚先前安装好的 cygwin 的具体位置，因此，关键名“native”将被找回并将其值（通常如 c:\cygwin 之类的）赋值给代表已知的 cygwin 位置的变量。

3、安装向导继续安装

用户选择好安装路径以及安装类型以后，安装向导复制所有必需的文件并进行必要的注册以及环境变量的修改等。这一步做完之后就只剩下 RPM 或 cygwin 的安装程序了。

下面列出的文件都复制完后，将启动 cygwin 安装程序；cygwin 安装完后，将从 cygwin 的 shell 上安装具有 RPM 的模块。所有日志文件都保存在

/home/Administrator/<RPM-name>.log

中。

TinyOS

-文件：

- TinyOS RPM
- 一个定制的.bashrc 文件放置在[INSTALLDIR]\cygwin\home\Administrator 中。

- 注册：无

-环境变量：

- TOSROOT 设置成 INSTALLDIR\tinyos-1.x

TinyOS Tools

-文件：

- TinyOS Tools RPM

-注册：无

-环境变量：无

NesC

- 文件：
  - NesC RPM
- 注册：无
- 环境变量：无

Cygwin

- 文件：
  - tinyos-1.1.0 cygwin 包放在[INSTALLDIR]\cygwin-installfiles 中
- 注册：无

-环境变量：无

Java

- Files:
  - JDK1.4.1_02 安装文件目录树复制到[INSTALLDIR]\jdk_1.4\j2sdk1.4.1_02.中

- 注册：
  -HKEY_LOCAL_MACHINE/Software/JavaSoft 树被复制到注册表中，其值以[INSTALLDIR]为前缀。
- 环境变量：无

Graphviz
-文件：
  - Graphviz 安装目录树复制到[INSTALLDIR]\ATT\Graphviz 中。
- 环境变量：
  - PATH：加上 Graphviz bin 目录。

AVR Tools
- 文件：
  - 每个工具都有一个 RPM
- 注册：无
- 环境变量：无

最后，设置一个环境变量：MOTECOM=serial@COM1:mica

注意：TinyOS 自动安装向导虽然允许用户可以自己决定选择安装某些部分，也可选择不安装某些部分，但是除非使用者对 TinyOS 各个不同模块、工具之间的交互及其联合工作的版本完全清楚，强烈建议选择完全安装。

在开始安装之前，要将所有与 TinyOS 相关的安装内容及工具全部删除。

必须以具有管理员权限的用户安装 TinyOS，否则的话，安装不可能成功而且还会留下残损的文件。

二、手动安装

首先，将与前面已经安装过的 TinyOS 相关的所有内容全部删除，否则只可能引起问题。

第一步：从 http://java.sun.com 上下载 JDK1.4，安装在适当的地方。

第二步：从 http://webs.cs.berkeley.edu/tos/dist-1.1.0/tools/windows/tinyos-cygwin-1.1.zip 上下载 cygwin 安装包，解压后运行 install.bat 脚本。

第三步：从 http://java.sun.com/products/javacomm/ 上下载 Sun 的 javax.comm 包，在 cygwin shell 命令行提示下按如下步骤安装（假定 JDK 安装在 c:\Program Files\jdk 下）：

1） 解压 javacomm20-win32.zip；
2） cd commapi；
3） cp win32com.dll "c:\Program Files\jdk\jre\bin"；
4） chmod 755 "c:\Program Files\jdk\jre\bin\win32com.dll"；
5） cp comm.jar "c:\Program Files\jdk\jre\lib\ext"；
6） cp javax.comm.properties "c:\Program Files\jdk\jre\lib"；

（此时按 javax.comm 包中的说明，运行 BlackBox 程序试试，如果正常就好；否则，尝试将上述几个文件复制到 c:\Program Files\java 路径下对应的目录中，并设置好环境变量，再运行 BlackBox。）

第四步：从 http://webs.cs.berkeley.edu/tos/dist-1.1.0/tools/windows/graphviz-1.10.exe 上下载 graphviz，将其安装在适当的路径下；

第五步：从 http://webs.cs.berkeley.edu/tos/dist-1.1.0/tools/windows 上下载如下几个 rpm：

avr-binutils-2.13.2.1-1w.cygwin.i386.rpm
avr-gcc-3.3tinyos-1w.cygwin.i386.rpm
avr-insight-pre6.0cvs.tinyos-1w.cygwin.i386.rpm
avr-libc-20030512cvs-1w.cygwin.i386.rpm
再从 http://webs.cs.berkeley.edu/tos/dist-1.1.0/tinyos/windows 上下载
nesc-1.1-1w.cygwin.i386.rpm
tinyos-tools-1.1.0-1.cygwin.i386.rpm
tinyos-1.1.0-1.cygwin.noarch.rpm

一共 8 个 rpm，在 cygwin shell 命令行下转到这些 rpm 文件存放的目录执行如下命令进行安装："rpm --ignoreos -ivh *.rpm"。这些 TinyOS 的安装包在安装过程中需要编译并执行 java 代码，因此需要占用一定时间。命令执行完毕后，TinyOS 即被安装到 cygwin 的/opt/tinyos-1.x 目录下。

第六步：至此，安装工作已成功完成。要想知道更多信息和细节，请查看 /opt/tinyos-1.x/doc/index.html；若想要安装更多的包，请参看 Installing and Updating Packages。

安装和更新包

在 http://webs.cs.berkeley.edu/tos/dist-1.1.0/tinyos/linux 和 http://webs.cs.berkeley.edu/tos/dist-1.1.0/tinyos/windows 目录下包含许多可选择安装的包，还包含 TinyOS 的核心安装包的更新包。可随时下载适当的 rpm，使用如下命令安装：

`rpm -ivh <rpm 文件名>`（第一次安装）
`rpm -Uvh <rpm 文件名>`（更新）。

2 TinyOS 支持多种不同设备

TinyOS 开发环境包括许多特点，这些特点使得对各种不同设备进行程序设计变得十分简单易行。这些特点包括：

1. 直接支持大量不同的编程器（编程接口）和方式，包括：
   - MIB500（crossbow）或其他标准并行端口编程主板；
   - MIB510（crossbow）基于串行端口的编程设备；
   - Atmel AVRISP（AVR In-System Programmer）标准；
   - EPRB 以太网编程主板。
2. 允许使用唯一的地址属性对每个设备进行编程，而不必每次编译应用程序。

下面描述如何在 TinyOS1.1 中使用上述这些特点：

1. 使用编程器

在 TinyOS 中使用的标准编程软件是 μ 内置系统编程器，即 UISP。作为 TinyOS 的一部分，UISP 根据编程器硬件以及期望的程序装载行为（如擦除、验证、程序加载等）获取不同的参数。TinyOS 可随时根据用户发出的“install”（加载）或“reinstall”（重新加载）命令使用正确的参数调用 UISP。用户只需指定正在使用的设备类型以及如何与之通信即可。要做到这一点，需要使用环境变量。

（1）MIB500 并口编程器

这是缺省的编程器设备，使用它时不需要指定额外的命令行参数。

（2）MIB510

定义：MIB510＝<dev>，其中<dev>是设备连接的串口名（即/dev/ttyS0）。

例子：bash% MIB510=/dev/ttyS1 make install mica。

（3）AVRISP

定义：AVRISP＝<dev>，其中<dev>是设备连接的串口名（即/dev/ttyS0）。

例子：bash% AVRISP =/dev/ttyS1 make install mica。

（4）EPRB

定义：EPRB=<host>，其中<host>是 EPRB 设备的域名或 IP 地址。

例子：bash% EPRB=123.45.67.89 make install mica。

2. 设备寻址

UISP 提供一种不必直接编辑 TinyOS 源代码而为节点设置唯一地址的方法。在程序装载期间设置节点地址使用如下语法：

make [re]install <addr> <platform>

其中，<addr>是期望的设备地址；<platform>是目标平台。命令 install 和 reinstall 的区别在于：前者为目标平台编译应用程序，并对设备设置地址和装载程序；后者仅为设备设置地址和装载程序。在使用地址时要注意，如下两个地址是保留值，不可使用：TOS_BCAST_ADDR（0xFFFF）和 TOS_UART_ADDR（0x007E）。

3 系统及硬件验证

使用嵌入式设备，调试应用程序十分困难，因此，工作前一定要确保所使用的工具工作正常以及各硬件系统功能完好。一旦某个部件或工具中真的存在某些问题而未及时发现，将耗费大量的时间去调试。下面介绍如何检查各硬件设备和软件系统。

一、PC 工具验证

如果在 Windwos 平台下使用 TinyOS 开发环境，需要使用 avr gcc 编译器、perl、flex、cygwin 及 JDK 及以上版本。“toscheck”是一个专门用来检验这些软件是否正确安装以及相应的环境变量是否设置完好的工具。

在 cygwin shell 命令行提示下，转到 tinyos-1.x/tools/scripts 目录，运行 toscheck，输出结果应该类似于：

toscheck

Path:

/usr/local/bin

/usr/bin

/bin

/cygdrive/c/jdk1.3.1_01/bin

/cygdrive/c/WINDOWS/system32
/cygdrive/c/WINDOWS
/cygdrive/c/avrgcc/bin

Classpath:
/c/alpha/tools/java:.:/c/jdk1.3.1_01/lib/comm.jar
avrgcc:
/cygdrive/c/avrgcc/bin/avr-gcc
Version: 3.0.2

perl:
/usr/bin/perl
Version: v5.6.1 built for cygwin-multi

flex:
/usr/bin/flex

bison:
/usr/bin/bison

java:
/cygdrive/c/jdk1.3.1_01/bin/java
java version "1.3.1_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_01)
Java HotSpot(TM) Client VM (build 1.3.1_01, mixed mode)

Cygwin:
cygwin1.dll major: 1003
cygwin1.dll minor: 3
cygwin1.dll malloc env: 28

uisp:
/usr/local/bin/uisp
uisp version 20010909

toscheck completed without error.

最后一行 十分重要，只有显式了这一行才表示安装无误；否则如果报告存在什么错误或问题，一定要将其修补好。

二、硬件验证

TinyOS 的 apps 目录下有个应用程序“MicaHWVerify”，它是一个专门用来测试 mica/mica2/mica2dot 硬件设备是否功能完好的工具。
转到目录/apps/MicaHWVerify 下，输入：

(mica platform) make mica

(mica2/mica2dot) PFLAGS=-DCC1K_MANUAL_FREQ=<freq> make [mica2|mica2dot]

若编译没问题，将输出一个内存描述，类似于：

compiled MicaHWVerify to build/mica2/main.exe

10386 bytes in ROM

390 bytes in RAM

avr-objcopy --output-target=srec build/mica2/main.exe build/mica2/main.srec。

如果输出结果与上述描述类似则说明应用程序已经编译好，下一步就将它加载到微粒中。将一个带电池的节点放到编程主板上，将微粒上的电源开关打开，这时，主板上的红色的 LED 灯将发亮。再把编程主板连接到 PC 机的并口（使用大的连接头－25 针）上，接下来就可以将程序加载到 mica 微粒中了。输入：

make reinstall [mica|mica2|mica2dot]。

如果输出类似于：

installing mica2 binary

uisp -dprog=<yourprogrammer> -dhost=c62b270 -dpart=ATmega128 --wr_fuse_e=ff --erase --upload if=build/mica2/main.srec

Atmel AVR ATmega128 is found.

Uploading: flash

Fuse Extended Byte set to 0xff，

说明编译工具及计算机的并口都已经在正常工作了。下面验证微粒硬件是否完好。首先确定加载了程序的微粒的 LED 像二进制计数器一样闪烁。然后利用串口线缆（小连接头－9 针）将编程主板连接到计算机串口（COM1）上。将要用来进行微粒硬件验证的工具是一个 java 应用程序，叫作“hardware_check.java”，就位于/apps/MicaHWVerify 目录下。编译并运行这个工具（若使用波特率为 57.6k 波特的 COM1 口），命令如下：

```
make -f jmakefile
MOTECOM=serial@COM1:57600 java hardware_check
```

。

（要想了解更多关于使用 hardware_check 如何指定串口或其他通信方法的信息，可参看"Serial-line communication in TinyOS"。）输出类似于：

hardware_check started

Hardware verification successful.

Node Serial ID: 1 60 48 fb 6 0 0 1d。

这个程序将检查微粒的序列号（除 mica2dot），flash 连接性，UART 功能以及外部时钟。若所有状态都正常，PC 机将输出硬件验证成功的信息；若有任何错误报告，则需要更换微粒。

三、无线传输验证

再取一个节点并安装 TOSBASE 应用程序（在/apps/tosbase 目录下），将这个节点作为无线网关。安装完毕后，将它放置在编程主板上，并将上面那个已经通过硬件验证的节点放到它附近。重新运行 java 应用程序“hardware_check”，输出应类似于：

hardware_check started

Hardware verification successful.

Node Serial ID: 1 60 48 fb 6 0 0 1d。

若将远处的微粒电源关掉或其坏掉，将显式“Node transmission failure（节点传输失败）”信息。

至此，若系统与硬件都通过上述所有测试，则说明二者都无问题，可以进行 TinyOS 的

开发工作了！

4 定制开发环境

在开发各种不同的应用程序时，需要根据特定的情况设定一些开发参数，这些参数包括：操作频率（mica2/mica2dot）、组标识、不同的组件库等。本部分介绍如何定制 TinyOS 开发环境。

一、makeloca 文件

apps 目录下的 Makerules 文件管理 TinyOS 应用程序建立（编译）过程的许多方面。原则上可以修改这个文件，但并不建议这么做，因为：

1） Makerules 文件并不直观；

2） 修改这个文件可能打断某个建立过程；

3） 修改这个文件可能对其他用户带来不便。

为了支持建立环境的定制，Makerules 文件会在 apps 目录下寻找 Makelocal 文件，并将之加入到建立进程中。而且对 Makelocal 文件作任何修改都很容易恢复（只需删除或重命名即可）。

因此，可以在 apps 目录下创建一个 Makelocal 文件，使用标准的 makefile 语法包含特定的定义，如：

```
# Makelocal File
```

PFLAGS += -I%T/../beta/MyBetaCode
DEFAULT_LOCAL_GROUP = 0x33
PFLAGS += -DCC1K_DEF_FREQ=916700000
EPRB=myprogrammer.foo.com。

上述 Makelocal 文件添加了一个搜索代码的路径../../beta/MyCode；定义了缺省的组标识（0x33）；为 mica2 系列微粒手动设置频率（91.67mHz）；并将主机名为“myprogrammer.foo.com”上的 EPRB 作为缺省的编程器。

附录：Make 及 Makefile

无论是在 Linux 还是在 Unix 环境中，make 都是一个非常重要的编译命令。不管是自己进行项目开发还是安装应用软件，我们都经常要用到 make 或 make install。利用 make 工具，我们可以将大型的开发项目分解成为多个更易于管理的模块，对于一个包括几百个源文件的应用程序，使用 make 和 makefile 工具就可以简洁明快地理顺各个源文件之间纷繁复杂的相互关系。而且如此多的源文件，如果每次都要键入 gcc 命令进行编译的话，那对程序员来说简直就是一场灾难。而 make 工具则可自动完成编译工作，并且可以只对程序员在上次编译后修改过的部分进行编译。因此，有效的利用 make 和 makefile 工具可以大大提高项目开发的效率。

1 Makefile 文件

Make 工具最主要也是最基本的功能就是通过 makefile 文件来描述源程序之间的相互关

系并自动维护编译工作。而 makefile 文件需要按照某种语法进行编写，文件中需要说明如何编译各个源文件并连接生成可执行文件，并要求定义源文件之间的依赖关系。makefile 文件是许多编译器－－包括 Windows NT 下的编译器－－维护编译信息的常用方法，只是在集成开发环境中，用户通过友好的界面修改 makefile 文件而已。

在 UNIX 系统中，习惯使用 Makefile 作为 makfile 文件。如果要使用其他文件作为 makefile，则可利用类似下面的 make 命令选项指定 makefile 文件：

$ make -f Makefile.debug

例如，一个名为 prog 的程序由三个 C 源文件 filea.c、fileb.c 和 filec.c 以及库文件 LS 编译生成，这三个文件还分别包含自己的头文件 a.h 、b.h 和 c.h。通常情况下，C 编译器将会输出三个目标文件 filea.o、fileb.o 和 filec.o。假设 filea.c 和 fileb.c 都要声明用到一个名为 defs 的文件，但 filec.c 不用。即在 filea.c 和 fileb.c 里都有这样的声明：

```
#include "defs"
```

那么下面的文档就描述了这些文件之间的相互联系：

```
-----------------------------------------------------------
  #It is a example for describing makefile
  prog : filea.o fileb.o filec.o
  cc filea.o fileb.o filec.o -LS -o prog
  filea.o : filea.c a.h defs
  cc -c filea.c
  fileb.o : fileb.c b.h defs
  cc -c fileb.c
  filec.o : filec.c c.h
  cc -c filec.c
-----------------------------------------------------------
```

这个描述文档就是一个简单的 makefile 文件。

从上面的例子注意到，第一个字符为 # 的行为注释行。第一个非注释行指定 prog 由三个目标文件 filea.o、fileb.o 和 filec.o 链接生成。第三行描述了如何从 prog 所依赖的文件建立可执行文件。接下来的 4、6、8 行分别指定三个目标文件，以及它们所依赖的.c 和.h 文件以及 defs 文件。而 5、7、9 行则指定了如何从目标所依赖的文件建立目标。

当 filea.c 或 a.h 文件在编译之后又被修改，则 make 工具可自动重新编译 filea.o，如果在前后两次编译之间，filea.C 和 a.h 均没有被修改，而且 test.o 还存在的话，就没有必要重新编译。这种依赖关系在多源文件的程序编译中尤其重要。通过这种依赖关系的定义，make 工具可避免许多不必要的编译工作。当然，利用 Shell 脚本也可以达到自动编译的效果，但是，Shell 脚本将全部编译任何源文件，包括哪些不必要重新编译的源文件，而 make 工具则可根据目标上一次编译的时间和目标所依赖的源文件的更新时间而自动判断应当编译哪个源文件。

Makefile 文件作为一种描述文档一般需要包含以下内容:

- 宏定义
- 源文件之间的相互依赖关系
- 可执行的命令

Makefile 中允许使用简单的宏指代源文件及其相关编译信息，在 Linux 中也称宏为变量。在用宏时只需在变量前加$符号，但值得注意的是，如果变量名的长度超过一个字符，在引用就必须加圆括号（）。

下面都是有效的宏引用：

$(CFLAGS)
$2
$Z
$(Z)
其中最后两个引用是完全一致的。

需要注意的是一些宏的预定义变量，在 Unix 系统中，$*、$@、$?和$<四个特殊宏的值在执行命令的过程中会发生相应的变化，而在 GNU make 中则定义了更多的预定义变量。

宏定义的使用可以使我们脱离那些冗长乏味的编译选项，为编写 makefile 文件带来很大的方便。

```
# Define a macro for the object files
OBJECTS= filea.o fileb.o filec.o
# Define a macro for the library file
LIBES= -LS
# use macros rewrite makefile
prog: $(OBJECTS)
cc $(OBJECTS) $(LIBES) -o prog
……
```

此时如果执行不带参数的 make 命令，将连接三个目标文件和库文件 LS；但是如果在 make 命令后带有新的宏定义：

```
make "LIBES= -LL -LS"
```

则命令行后面的宏定义将覆盖 makefile 文件中的宏定义。若 LL 也是库文件，此时 make 命令将连接三个目标文件以及两个库文件 LS 和 LL。

在 Unix 系统中没有对常量 NULL 作出明确的定义，因此我们要定义 NULL 字符串时要使用下述宏定义：

```
STRINGNAME=NULL
```

2 Make 命令

在 make 命令后不仅可以出现宏定义，还可以跟其他命令行参数，这些参数指定了需要编译的目标文件。其标准形式为：

target1 [target2 …]:[:][dependent1 …][;commands][#…]

[(tab) commands][#…]

方括号中间的部分表示可选项。Targets 和 dependents 当中可以包含字符、数字、句点和"/"符号。除了引用，commands 中不能含有"#",也不允许换行。

在通常的情况下命令行参数中只含有一个":"，此时 command 序列通常和 makefile 文件中某些定义文件间依赖关系的描述行有关。如果与目标相关连的那些描述行指定了相关的 command 序列，那么就执行这些相关的 command 命令，即使在分号和(tab)后面的 aommand 字段甚至有可能是 NULL。如果那些与目标相关连的行没有指定 command，那么将调用系统默认的目标文件生成规则。

如果命令行参数中含有两个冒号"::"，则此时的 command 序列也许会和 makefile 中所有描述文件依赖关系的行有关。此时将执行那些与目标相关连的描述行所指向的相关命令。同

时还将执行 build-in 规则。

如果在执行 command 命令时返回了一个非"0"的出错信号，例如 makefile 文件中出现了错误的目标文件名或者出现了以连字符打头的命令字符串，make 操作一般会就此终止，但如果 make 后带有"-i"参数，则 make 将忽略此类出错信号。

Make 命本身可带有四种参数：标志、宏定义、描述文件名和目标文件名。其标准形式为：

Make [flags] [macro definitions] [targets]

Unix 系统下标志位 flags 选项及其含义为：

-f file　指定 file 文件为描述文件，如果 file 参数为"-"符，那么描述文件指向标准输入。如果没有"-f"参数，则系统将默认当前目录下名为 makefile 或者名为 Makefile 的文件为描述文件。在 Linux 中，GNU make 工具在当前工作目录中按照 GNUmakefile、makefile、Makefile 的顺序搜索 makefile 文件。

-i　忽略命令执行返回的出错信息。

-s　沉默模式，在执行之前不输出相应的命令行信息。

-r　禁止使用 build-in 规则。

-n　非执行模式，输出所有执行命令，但并不执行。

-t　更新目标文件。

-q　make 操作将根据目标文件是否已经更新返回"0"或非"0"的状态信息。

-p　输出所有宏定义和目标文件描述。

-d　Debug 模式，输出有关文件和检测时间的详细信息。

Linux 下 make 标志位的常用选项与 Unix 系统中稍有不同，下面我们只列出了不同部分：

-c dir　在读取 makefile 之前改变到指定的目录 dir。

-I dir　当包含其他 makefile 文件时，利用该选项指定搜索目录。

-h　help 文挡，显示所有的 make 选项。

-w　在处理 makefile 之前和之后，都显示工作目录。

通过命令行参数中的 target ，可指定 make 要编译的目标，并且允许同时定义编译多个目标，操作时按照从左向右的顺序依次编译 target 选项中指定的目标文件。如果命令行中没有指定目标，则系统默认 target 指向描述文件中第一个目标文件。

通常，makefile 中还定义有 clean 目标，可用来清除编译过程中的中间文件，例如：

clean:

rm -f *.o

运行 make clean 时，将执行 rm -f *.o 命令，最终删除所有编译过程中产生的所有中间文件。

3　隐含规则

在 make 工具中包含有一些内置的或隐含的规则，这些规则定义了如何从不同的依赖文件建立特定类型的目标。Unix 系统通常支持一种基于文件扩展名即文件名后缀的隐含规则。这种后缀规则定义了如何将一个具有特定文件名后缀的文件（例如.c 文件），转换成为具有另一种文件名后缀的文件（例如.o 文件）：

.c:.o

$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

系统中默认的常用文件扩展名及其含义为：

.o 目标文件
.c C 源文件
.f FORTRAN 源文件
.s 汇编源文件
.y Yacc-C 源语法
.l Lex 源语法

在早期的 Unix 系统系统中还支持 Yacc-C 源语法和 Lex 源语法。在编译过程中，系统会首先在 makefile 文件中寻找与目标文件相关的.C 文件，如果还有与之相依赖的.y 和.l 文件，则首先将其转换为.c 文件后再编译生成相应的.o 文件；如果没有与目标相关的.c 文件而只有相关的.y 文件，则系统将直接编译.y 文件。

而 GNU make 除了支持后缀规则外还支持另一种类型的隐含规则--模式规则。这种规则更加通用，因为可以利用模式规则定义更加复杂的依赖性规则。模式规则看起来非常类似于正则规则，但在目标名称的前面多了一个 % 号，同时可用来定义目标和依赖文件之间的关系，例如下面的模式规则定义了如何将任意一个 file.c 文件转换为 file.o 文件：

%.c:%.o

$(CC) $(CFLAGS) $(CPPFLAGS) -c -o $@ $<

#EXAMPLE#

下面将给出一个较为全面的示例来对 makefile 文件和 make 命令的执行进行进一步的说明，其中 make 命令不仅涉及到了 C 源文件还包括了 Yacc 语法。本例选自"Unix Programmer's Manual 7th Edition, Volume 2A" Page 283-284

下面是描述文件的具体内容：

```
#Description file for the Make command
#Send to print
P=und -3 | opr -r2
#The source files that are needed by object files
FILES= Makefile version.c defs main.c donamc.c misc.c file.c \
dosys.c gram.y lex.c gcos.c
#The definitions of object files
OBJECTS= vesion.o main.o donamc.o misc.o file.o dosys.o gram.o
LIBES= -LS
LINT= lnit -p
CFLAGS= -O
make: $(OBJECTS)
cc $(CFLAGS) $(OBJECTS) $(LIBES) -o make
size make
$(OBJECTS): defs
gram.o: lex.c
cleanup:
-rm *.o gram.c
install:
@size make /usr/bin/make
cp make /usr/bin/make ; rm make
#print recently changed files
```

```
print: $(FILES)
pr $? | $P
touch print
test:
make -dp | grep -v TIME>1zap
/usr/bin/make -dp | grep -v TIME>2zap
diff 1zap 2zap
rm 1zap 2zap
lint: dosys.c donamc.c file.c main.c misc.c version.c gram.c
$(LINT) dosys.c donamc.c file.c main.c misc.c version.c \
gram.c
rm gram.c
arch:
ar uv /sys/source/s2/make.a $(FILES)
```

通常在描述文件中应象上面一样定义要求输出将要执行的命令。在执行了 make 命令之后，输出结果为：

```
$ make
cc -c version.c
cc -c main.c
cc -c donamc.c
cc -c misc.c
cc -c file.c
cc -c dosys.c
yacc gram.y
mv y.tab.c gram.c
cc -c gram.c
cc version.o main.o donamc.o misc.o file.o dosys.o gram.o \
-LS -o make
13188+3348+3044=19580b=046174b
```

最后的数字信息是执行"@size make"命令的输出结果。之所以只有输出结果而没有相应的命令行，是因为"@size make"命令以"@"起始，这个符号禁止打印输出它所在的命令行。

描述文件中的最后几条命令行在维护编译信息方面非常有用。其中"print"命令行的作用是打印输出在执行过上次"make print"命令后所有改动过的文件名称。系统使用一个名为 print 的 0 字节文件来确定执行 print 命令的具体时间，而宏$?则指向那些在 print 文件改动过之后进行修改的文件的文件名。如果想要指定执行 print 命令后，将输出结果送入某个指定的文件，那么就可修改 P 的宏定义：

```
make print "P= cat>zap"
```

大多数自由软件提供的是源代码，而不是现成的可执行文件，这就要求用户根据自己系统的实际情况和自身的需要来配置、编译源程序后，软件才能使用。

第二章 TinyOS 简介

本课介绍与 TinyOS 相关的一些主要概念，包括对组件 component、接口 interface、命令 command 及事件 event 等概念的描述，解释了 TinyOS 的编程模型，并详细描述了不同文件类型的角色。

本课还介绍了 nesC 语言，TinyOS 系统就是用该语言写的。对该语言的概念和语法有了基本的了解以后就可以在这个环境中编写自己的应用程序了。

1 简介

TinyOS 系统、库及应用程序都是用 nesC 语言写的语言写的，这时一种新的用于编写结构化的基于组件的应用程序的语言。nesC 语言主要用于诸如传感器网络等嵌入式系统。nesC 具有类似于 C 语言的语法，但支持 TinyOS 的并发模型，同时具有机构化机制、命名机制，能够与其他软组件链接在一起从而形成一个鲁棒的网络嵌入式系统。其主要目标是帮助应用程序设计者建立可易于组合成完整、并发式系统的组件，并能够在编译时执行广泛的检查。

TinyOS 定义了许多在 nesC 中所表达的重要概念。首先，nesC 应用程序要建立在定义良好、具有双向接口的组件之上。其次，nesC 定义了并发模型，该模型是基于任务（tast）及硬件事件句柄（hardware event handler）的，在编译时会检测数据争用（data race）。

1、组件

1） 说明

任何一个 nesC 应用程序都是有一个或多个组件链接起来，从而形成一个完整的可执行程序的。组件提供（provide）并使用（use）接口。这些接口是组件的唯一访问点并且它们是双向的。接口声明了一组函数，称为命令（command），接口的提供者必须实现它们；还声明了另外一组函数，称为事件（event），接口的使用者必须实现它们。对于一个组件而言，如果它要使用某个接口中的命令，它必须实现这个接口的事件。一个组件可以使用或提供多个接口以及同一个接口的多个实例。

2）实现

在 nesC 中有两种类型的组件，分别称为模块（module）和配置（configuration）。模块提供应用程序代码，实现一个或多个接口；配置则是用来将其它组件装配起来的组件，将各个组件所使用的接口与其它组件提供的接口连接在一起。这种行为称为导通（wiring）。每个 nesC 应用程序都由一个顶级配置所描述，其内容就是将该应用程序所用到的所有组件导通起来，形成一个有机整体。

nesC 的所有源文件，包括 interface、module 和配置，其文件后缀（扩展名）都是“.nc”。要了解有关命名规则方面的详细信息，请参看 TinyOS Coding and Naming Conventions 。

2、并发模型（Concurrency Model）

TinyOS 一次仅执行一个程序。组成程序的组件来自于两个方面，一部分是系统提供的组件，另一部分是为特定应用用户自定义的组件。程序运行时，有两个执行线程：一个称为任务（task），另一个称为硬件事件句柄（hardware event handler）。任务是被延期执行的函数，它们一旦被调度，就会运行直至结束，并且在运行过程中不准相互抢占。硬件事件句柄是用来相应和处理硬件中断的，虽然也要运行完毕，但它们可能会抢占任务或其他硬件事件句柄的执行。命令和事件要作为硬件事件句柄的一部分而执行必须使用关键字 **async** 来声明。

因为任务和硬件事件句柄可能被其他异步代码所抢占，所以 nesC 程序易于受到特定竞争条件的影响，导致产生不一致或不正确的数据。避免竞争的办法通常是在任务内排他地访问共享数据，或访问所有数据都使用原子语句。nesC 编译器会在编译时向程序员报告潜在的数据争用，这里面可能包含事实上并不可能发生的冲突。 如果程序员确实可以担保对某个数据的访问不会导致麻烦，可以将该变量使用关键字 **norace** 来声明。但使用这个关键字一定要格外小心。

要了解用 nesC 编程的更多信息，请参看 nesC Language Reference Manual。

2 应用程序举例：Blink

下面来看一个完整的应用程序的例子，通过这个例子来具体了解在 TinyOS 环境中使用 nesC 应用程序的结构和使用细节。应用程序 Blink 位于 apps/Blink 目录下，这时一个简单的测试程序，其作用是使微粒上的红色的 LED 灯以 1Hz 的频率闪烁。

Blink 应用程序由两个组件组成：一个名为“BlinkM.nc”的模块和一个名为“Blink.nc”的配置。前面讲过，所有应用程序都需要一个顶级配置文件，习惯上其名称与应用程序本身同名。本例中，“Blink.nc”就是 Blink 应用程序的配置，也是 nesC 编译器用来生成可执行程序文件的源文件。另一方面，“BlinkM.nc”是提供 Blink 应用程序实际实现的文件。“Blink.nc”是用来将“BlinkM.nc”模块与 Blink 应用程序所需的其他组件导通起来的。

将模块和配置予以严格地区分，可以使系统设计者快速地重新装配（snap together）应用程序，从而使得应用程序的设计和更新更加方便易行。例如：某个应用设计者可能提供一个仅简单地将一个或多个模块导通在一起的配置，实际上并不设计其中任何一个模块；同时，由另一个开发者来提供一套全新的适用于该应用范围的“库”模块。这样，将不同粒度的设计工作有效的分开，符合软件设计的一般规则。

当然，配置和模块有时也同时出现在一起，如本例中的 Blink 和 BlinkM。在 TinyOS 源文件树中，通常用类似 Foo.nc 的文件表示配置，而用类似 FooM.nc 的文件表示相应模块。当然，程序员完全可以使用其他方式的命名规则，但采取上述方式会使问题简单明了。TinyOS 中的其他命名规则请参看 summary。

2.1 Blink.nc 配置

nesC 的编译器为 ncc，它可以将包含顶级配置的文件编译成可执行的应用程序。一般而言，TinyOS 应用程序还拥有一个标准的 Makefile 文件，允许进行平台选择以及在调用 ncc 时使用某些适当的选项。

先来看看这个应用程序的配置源文件 Blink.nc：

Blink.nc

```
configuration Blink {
}
implementation {
  components Main, BlinkM, SingleTimer, LedsC;
  Main.StdControl -> SingleTimer.StdControl;
  Main.StdControl -> BlinkM.StdControl;
```

```
  BlinkM.Timer -> SingleTimer.Timer;
  BlinkM.Leds -> LedsC;
}
```

首先看关键字 configuration，它表明这时一个配置文件。开头的两行

```
configuration Blink {
}
```

只是简单地声明了该配置名为 Blink。跟模块一样，在声明后的这个花括号内可以指定 uses 子句和 provides 子句。这一点非常重要：配置可以提供和使用接口。

配置的实际内容是由跟在关键字 implementation 后面的花括号部分来实现的。Components 这一行指定了该配置要引用的组件集合，此例中是 Main，BlinkM，SingleTimer 和 LedsC。实现的剩余部分将这些组件使用的接口与提供这些接口的其他组件连接起来，即是前面所说的“导通”操作。

Mian 是在 TinyOS 应用程序中首先被执行的一个组件。确切的说，在 TinyOS 中执行的第一个命令是 Main.StdControl.init()，接下来是 Main.StdControl.start()。因此，TinyOS 应用程序在其配置中必须要有 Main 组件。接口 StdControl 是用来初始化和启动 TinyOS 组件的一个公共（通用）接口，它的源文件位于 tos/interfaces/StdControl.nc，代码如下所示：

StdControl.nc

```
interface StdControl
{
  /**
   * Initialize the component and its subcomponents.
   *
   * @return Whether initialization was successful.
   */
  command result_t init();

  /**
   * Start the component and its subcomponents.
   *
   * @return Whether starting was successful.
   */
  command result_t start();

  /**
   * Stop the component and pertinent subcomponents (not all
   * subcomponents may be turned off due to wakeup timers, etc.).
   *
   * @return Whether stopping was successful.
   */
  command result_t stop();
}
```

可以看出 StdControl 接口定义了三个命令（**command**），分别是 init()，start()，及 stop()。当组件第一次初始化时调用 init() 命令，启动时调用 start()命令。stop()命令是在组件停止时

调用，例如，将其控制的设备的电源断开。init() 命令可以被调用多次，但如果调用了 start()命令或 stop()命令以后就再也不能被调用。 特别地，StdControl 的有效调用模式为 init*(start | stop)*。这三条命令都具有“深”层次的语义；调用某个组件上的 init()命令必须使它调用其子组件上的所有 init()命令。

Blink 配置中有如下两行：

Main.StdControl -> SingleTimer.StdControl;

Main.StdControl -> BlinkM.StdControl;。

其作用时将 Main 组件的接口 StdControl 与 BlinkM 和 SingleTimer 中的 StdControl 接口导通起来。SingleTimer.StdControl.init()及 BlinkM.StdControl.init()将被 Main.StdControl.init()调用。同样的规则也适用于 start()命令及 stop()命令。

至于“被使用（used）”的接口，其子组件的初始化函数必须被使用组件显式地调用。如：BlinkM 模块使用接口 Leds，于是 Leds.init()命令要再 BlinkM.init()中被显式地调用。

nesC 使用箭头（->）来指示和标识接口间的关系，其意义为“绑定”，即左边的接口绑定到右边的实现上。换言之，使用接口的组件在左边，提供接口的组件在右边。

“BlinkM.Timer -> SingleTimer.Timer;”这一句话的意思是将组件 BlinkM 所使用的接口 Timer 与组件 SingleTimer 所提供的接口 Timer 导通起来。箭头左边的 BlinkM.Timer 引用名为 Time 的接口（tos/interfaces/Timer.nc），而箭头右边的 SingleTimer.Timer 则指向 Timer 的实现（tos/lib/SingleTimer.nc）。箭头的作用就是将其左边的接口与其右边的实现绑定起来。

nesC 支持同一个接口的多个实现。Timer 接口即是如此。SingleTimer 组件实现了一个单一的 Timer 接口；而另一个组件 TimerC（tos/system/TimeC.nc）使用 timer id 作为参数实现了多个 Timer。下一课将进一步讨论定时器的相关问题。

导通 也可以使用隐式的写法，如：

BlinkM.Leds -> LedsC;

就是

BlinkM.Leds -> LedsC.Leds;

简写形式。若箭头右边没有指定接口名，nesC 编译器缺省情况下尝试与箭头左边同名的接口进行绑定。

2.2 BlinkM.nc 模块

再来看看 BlinkM.nc 模块：

BlinkM.nc

```
/**
 * Implementation for Blink application.  Toggle the red LED when a
 * Timer fires.
 **/
module BlinkM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
```

```
    interface Leds;
  }
}
implementation {

  /**
   * Initialize the component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.init() {
    call Leds.init();
    return SUCCESS;
  }


  /**
   * Start things up.  This just sets the rate for the clock component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.start() {
    // Start a repeating timer that fires every 1000ms
    return call Timer.start(TIMER_REPEAT, 1000);
  }

  /**
   * Halt execution of the application.
   * This just disables the clock component.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  command result_t StdControl.stop() {
    return call Timer.stop();
  }


  /**
   * Toggle the red LED in response to the <code>Timer.fired</code> event.
   *
   * @return Always returns <code>SUCCESS</code>
   **/
  event result_t Timer.fired()
  {
```

```
    call Leds.redToggle();
    return SUCCESS;
  }
}
```

BlinkM 模块提供了 StdControl 接口，这意味着它必须实现这个接口。如前所述，要使 BlinkM 组件得以初始化和启动，必须要实现这个接口。BlinkM 模块还使用了两个接口，分别是 Leds 和 Timer。这意味着它可能调用这些接口中声明的任何命令以及必须实现这些接口中声明的任何事件。

Leds 接口（tos/interfaces/Leds.nc）定义了多个命令，如：redOn()，redOff()等等，其作用是将微粒上的 LED（红、绿、黄）灯打开或关闭。由于 BlinkM 组件使用 Leds 接口，因此它可调用其中任一命令。但请注意，Leds 仅仅只是一个接口，其实现由使用它的组件对应的配置文件指定。此例中，在 Blink.nc 中指定，为 LedsC，即是要由 LedsC 来实现 Leds 接口。LedsC 位于 tos/system/LedsC.nc，与 Timer.nc 一样，同属于 TinyOS 的系统组件。

Timer 接口似乎更有趣，其代码如下：

Timer.nc

```
/**
 * This interface provides a generic timer that can be used to generate
 * events at regular intervals.
includes Timer; // make TIMER_x constants available
interface Timer {

  /**
   * Start the timer.
   * @param type The type of timer to start. Valid values include
   *   'TIMER_REPEAT' for a timer that fires repeatedly, or
   *   'TIMER_ONE_SHOT' for a timer that fires once.
   *   @param interval The timer interval in <b>binary milliseconds</b>
(1/1024
   *   second). Note that the
   *      timer cannot support an arbitrary range of intervals.
   *      (Unfortunately this interface does not specify the valid range
   *      of timer intervals, which are specific to a platform.)
   *   @return Returns SUCCESS if the timer could be started with the
   *      given type and interval. Returns FAIL if the type is not
   *      one of TIMER_REPEAT or TIMER_ONE_SHOT, if the timer rate
is
   *      too high, or if there are too many timers currently active.
   */
  command result_t start(char type, uint32_t interval);

  /**
   * Stop the timer, preventing it from firing again.
   * If this is a TIMER_ONE_SHOT timer and it has not fired yet,
```

```
    * prevents it from firing.
    * @return SUCCESS if the timer could be stopped, or FAIL if the timer
    * is not running or the timer ID is out of range.
    */
  command result_t stop();

  /**
    * The signal generated by the timer when it fires.
    */
  event result_t fired();
}
```

可以看出，Timer 接口除了定义了两个命令 start()和 stop()以外，还定义了一个事件 fired()。

start()命令用于指定定时器类型及闪烁时间间隔。时间间隔的单位是毫秒（ms）。计数器的有效值为 TIMER_REPEAT 和 TIMER_ONE_SHOT。后者会在指定的时间间隔后停止闪烁（即仅闪一次），而前者会不停的闪烁直至 stop()命令执行。

应用程序是如何知道定时器时间到的呢？答案是它接收到了某个事件发生。Timer 接口提供了一个事件，即

event result_t fired();。

事件是当某个事情发生时接口的实现就发出信号（signal）的函数。在本例中，当指定的时间间隔到达时，fired()事件就被触发。（Timer.nc －－－SingleTiemer.nc －－－Timer.nc －－－TimerM.nc）。这是一个双向接口的例子：不仅提供被该接口使用者调用的命令，而且触发事件，该事件再调用接口使用者的处理函数。可以认为事件是接口的实现者将会调用的一个回调函数。使用接口的模块必须实现接口使用的事件。

BlinkM 模块剩下的部分就很简单了，它实现了 StdControl.init()，StdControl.start()及 StdControl.stop()等命令，因为它提供了 StdControl 接口。它还实现了 Timer.fired()事件，这一点是必须的，因为它必须实现其使用的接口的所有事件。

接口 StdControl 中 init()命令的实现只是简单的调用了 Leds.init()函数，从而将子组件 Leds 予以初始化。而 start()命令则是调用 Timer.start()函数，以创建一个反复循环计时器，其周期为 1000 ms。stop()命令用以终止计时器。每次 Timer.fired()事件被触发时，函数 Leds.redToggle()将被调用，从而使红色的 LED 灯发亮。

TinyOS 提供了一种在应用程序内将使用到的各组件之间的关系用图形化方法表示的工具。事实上，在 TinyOS 的源文件中包含了位于注释中的元数据，nesC 的编译器 ncc 可用之来自动生成 html 格式的文档。使用方法是在应用程序目录下，输入

make <platform> docs

命令。输出的结果文档将位于 doc/nesdoc/<platform>中，其中<platform>是正在使用的平台，如 mica 或 mica2 等。其中 doc/nesdoc/<platform>/index.html 是所有文档化的应用程序的索引页面。Blink 应用程序使用该方法生成的文档的索引页面内容为：

Source Tree

apps/
Blink/
Blink.nc

BlinkM.nc
SingleTimer.nc
tos/
interfaces/
Clock.nc
HPLPot.nc
Leds.nc
Pot.nc
PowerManagement.nc
StdControl.nc
Timer.nc
platform/
avrmote/
HPLInit.nc
HPLPotC.nc
mica/
HPLClock.nc
HPLPowerManagementM.nc
system/
ClockC.nc
LedsC.nc
Main.nc
NoLeds.nc
PotC.nc
PotM.nc
RealMain.nc
TimerC.nc
TimerM.nc

2.3 编译 Blink 应用程序

TinyOS 支持多平台。每个平台在 tos/platform 目录下都有自己的目录。此处，以 mica 平台为例。为 mica 微粒编译 Blink 应用程序只需在 apps/Blink 目录下输入：

make mica

命令即可。这里看不出任何调用 nesC 编译器的语句，要了解更多关于 make 的信息，请参看前面的附录。调用 nesC 编译器本身需要使用基于 gcc 的命令 ncc 。例如：

ncc -o main.exe -target=mica Blink.nc，

该命令将顶级配置 Blink.nc 编译成 mica 微粒中的可执行程序 main.exe。再使用命令

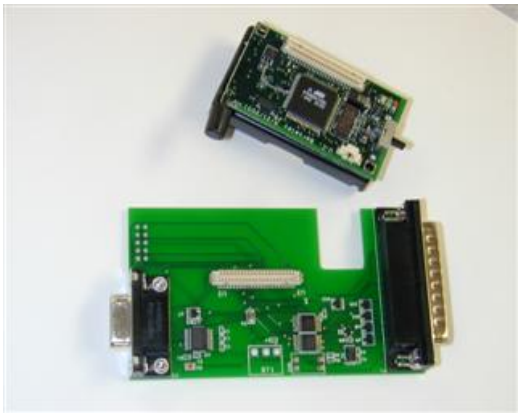
avr-objcopy --output-target=srec main.exe main.srec

来产生 main.srec 文件，该文件以一种可用来对微粒进行编程的文本格式来表示二进制的 main.exe 文件。然后再根据应用环境使用其他工具（如 uisp）将代码加载到微粒中。一般而言，并不需要手动调用 ncc 或 avr-objcopy 等，Makefile 都已经做好了一切。

2.4 加载并运行 Blink

现在可以将应用程序加载到微粒中并运行它了。本例中将使用 Mica 微粒以及基于并行端口的编程主板（mib500）。使用其他编程主板的方法请参看 http://webs.cs.berkeley.edu/tos/tinyos-1.x/doc/tutorial/programmers.html。

先将微粒主板放到编程主板上，如下图。再将 3 伏电源线接到编程主板的连接头上，或直接用电池供电。加电后，编程主板上的红色 LED 灯将会发亮。



Mica 微粒与编程主板

连接到编程主板上的 Mica 微粒

使用标准的 DB32 并行端口电缆将 32 针连接头插到 PC 机的并口上，输入：

make mica install，

若显式错误信息为：

```
uisp -dprog=dapa --erase
pulse
An error has occurred during the AVR initialization.
 * Target status:
   Vendor Code = 0xff, Part Family = 0xff, Part Number = 0xff


Probably the wiring is incorrect or target might be `damaged'.
make: *** [install] Error 2。
```

检查电源是否打开，或者电源电量是否充足以及 uisp 版本是否正确。

若使用的 PC 机是 IBM 的笔记本电脑，则必须使用不同的并口。只需在 apps/Makelocal 文件（若无此文件就创建一个）中加入一行：

PROGRAMMER_EXTRA_FLAGS = -dlpt=3。

Makefile 文件是特定用户的 Makefile 文件，要了解更多关于 Makelocal 文件的信息，请参看前面的章节——“定制开发环境”。

若程序加载没有问题，则提示信息类似于：

```
compiling Blink to a mica binary
```

```
  ncc -board=micasb -o build/mica/main.exe -Os -target=mica   -Wall -Wshadow
-DDEF_TOS_AM_GROUP=0x7d -finline-limit=200 -fnesc-cfile=build/mica/app.c   Blink.nc –lm
  avr-objcopy --output-target=srec build/mica/main.exe
  build/mica/main.srec
     compiled Blink to build/mica/main.srec
     installing mica binary
  uisp -dprog=dapa   --erase
  pulse
  Atmel AVR ATmega128 is found.
  Erasing device ...
  Pulse
  Reinitializing device
  Atmel AVR ATmega128 is found.
  sleep 1
  uisp -dprog=dapa   --upload if=build/mica/main.srec
  pulse
  Atmel AVR ATmega128 is found.
  Uploading: flash
  sleep 1
  uisp -dprog=dapa   --verify if=build/mica/main.srec
  pulse
  Atmel AVR ATmega128 is found.
  Verifying: flash
```

现在可将微粒从编程主板上取下来。打开电源，若红色的 LED 灯每秒闪烁一下则表明程序编译和加载成功！

清除 Blink 目录下的二进制文件可使用

make clean

命令。

如果仍然存在问题，请检查 TinyOS 十分安装正确或 Mica 硬件是否完好。相关内容请参看有关章节——“系统及硬件验证”。

1、 总结

本课简要地介绍了 nesC 的语法和特点。要想更详细地了解它，请参看 nesC Project Pages 以及 nesc/doc 目录下面的文档。

第三章 用事件驱动方式从传感器读取数据

本课将演示一个简单的传感器应用程序 Sense，它从传感器主板上的照片（photo）传感器上获取光强度值并将其低三位值显式在微粒的 LED 上。该应用程序位于 apps/Sense 目录下，其配置文件为 Sense.nc，实现模块文件为 SenseM.nc。

1 SenseM.nc 模块

先来看看其源代码，如下所示：

SenseM.nc

```
module SenseM {
  provides {
    interface StdControl;
  }
  uses {
    interface Timer;
    interface ADC;
    interface StdControl as ADCControl;
    interface Leds;
  }
}

implementation {

  // declare module static variables here



  /**
    * Module scoped method.  Displays the lowest 3 bits to the LEDs,
    * with RED being the most signficant and YELLOW being the least
significant.
    *
    * @return returns <code>SUCCESS</code>
    **/
  // display is module static function
  result_t display(uint16_t value)
  {
    if (value &1) call Leds.yellowOn();
    else call Leds.yellowOff();
    if (value &2) call Leds.greenOn();
    else call Leds.greenOff();
    if (value &4) call Leds.redOn();
    else call Leds.redOff();
```

```
    return SUCCESS;
  }
 /**
   * Initialize the component. Initialize ADCControl, Leds
   *
   * @return returns <code>SUCCESS</code> or <code>FAILED</code>
   **/
  // implement StdControl interface
  command result_t StdControl.init() {
    return rcombine(call ADCControl.init(), call Leds.init());
  }
  /**
   * Start the component. Start the clock.
   *
   * @return returns <code>SUCCESS</code> or <code>FAILED</code>
   **/
  command result_t StdControl.start() {
    return call Timer.start(TIMER_REPEAT, 500);
  }

  /**
   * Stop the component. Stop the clock.
   *
   * @return returns <code>SUCCESS</code> or <code>FAILED</code>
   **/
  command result_t StdControl.stop() {
    return call Timer.stop();
  }

  /**
   * Read sensor data in response to the <code>Timer.fired</code>
event.
   *
   * @return The result of calling ADC.getData().
   **/
  event result_t Timer.fired() {
    return call ADC.getData();
  }

  /**
   * Display the upper 3 bits of sensor reading to LEDs
   * in response to the <code>ADC.dataReady</code> event.
   * @return Always returns <code>SUCCESS</code>
   **/
```

```
  // ADC data ready event handler
  async event result_t ADC.dataReady(uint16_t data) {
    display(7-((data>>7) &0x7));
    return SUCCESS;
  }

}
```

与 BlinkM 类似，SenseM 提供了 StdControl 接口并使用了 Timer 和 Leds 接口，同时还使用了另外两个接口，分别是：ADC 接口——用于从模拟－数字转换器上存取数据；StdControl 接口——用于初始化 ADC 组件。

该程序还使用了一个新的组件 TimerC，代替前面使用过了的 SingleTimer。原因是 TimerC 允许使用多个定时器实例，而 SingleTimer 仅提供一个组件能使用的单个计时器。有关定时器 Timer 的相关问题后面还会讨论。

请注意这一行

```
interface StdControl as ADCControl;
```

其意义是本组件使用 StdControl 接口，但将该接口的实例命名为 ADCControl。使用这种方式，一个组件可以使用同一接口的多个实例，但可将它们分别命以不同的名字。例如：某个组件可能同时需要两个 StdControl 接口来分别控制 ADC 和 Sounder 两个组件，那么，可以按如下方式声明：

```
interface StdControl as ADCControl;
interface StdControl as SounderControl;
```

然后，使用该模块的配置将负责将每个接口实例与真实的实现导通起来。

事实上，在 TinyOS 中，如果不使用 as 语句提供接口名称，那么缺省情况下实例与接口同名，也就是说，语句

```
interface ADC;
```

实际上就是语句

```
interface ADC as ADC;
```

的简写形式。

下面来看看 StdControl 接口和 ADC 接口（都在 tos/interfaces 目录下）。接口 StdControl 是用来对组件（通常为一片物理硬件）进行初始化并对之加电；接口 ADC 则是用来从 ADC 信道获取数据。若数据在 ADC 信道上已经准备好了，则 ADC 接口会触发事件 dataReady()。请注意在 ADC 接口中使用了关键字 **async**，它表示所声明的命令和事件为异步代码。异步代码是可以对硬件中断予以及时响应的代码。

分析 SenseM.nc 源代码，不难看出，每当 Timer.fired()事件触发时就会调用 ADC.getData()函数；同样，当 ADC.dataReady()事件触发时，就调用内部函数 display()，该显示函数用 ADC 值的低序位上的数值来设置 LED。

注意到 StdControl.init()的实现中使用了函数 rcombine()，即

```
return rcombine(call ADCControl.init(), call Leds.init());
```

该函数是一个特殊的 nesC 连接函数，返回值为结果类型同为 result_t 的两个命令的逻辑“与”。

2 Sense.nc 配置

Sense 应用程序是如何知道 ADC 信道应该访问光传感器的呢？这正是 Sense.nc 配置所要解决的。

Sense.nc

```
configuration Sense {
  // this module does not provide any interface
}
implementation
{
  components Main, SenseM, LedsC, TimerC, Photo;

  Main.StdControl -> SenseM;
  Main.StdControl -> TimerC;

  SenseM.ADC -> Photo;
  SenseM.ADCControl -> Photo;
  SenseM.Leds -> LedsC;
  SenseM.Timer -> TimerC.Timer[unique("Timer")];
}
```

该文件代码中大部分语句与 Blink 中的类似，如将 Main.StdControl 与 SenseM.StdControl 接口导通起来，Leds 接口也类似。ADC 的导通语句：

```
SenseM.ADC -> Photo;
SenseM.ADCControl -> Photo;
```

将 ADC 接口（被 SenseM 使用的）绑定到一个新的称为 Photo 的组件上；ADCControl 接口也一样，而这个接口是 SenseM 使用的 StdControl 接口的一个实例。其实，语句

```
SenseM.ADC -> Photo;
```

是语句

```
SenseM.ADC -> Photo.ADC;
```

的简写形式；而语句

```
SenseM.ADControl -> Photo;
```

并非是

```
SenseM.ADC -> Photo.ADCControl;
```

语句的简写形式。为什么呢？来看看 Photo.nc 组件(在 tos/sensorboards/micasb 目录下)，它提供两个接口，分别是 ADC 接口和 StdControl 接口——而并无 ADCControl 接口。（事实上，ADCControl 只是 SenseM 组件中给 StdControl 接口的某个实例取的一个新名字）。nesC 编译器具有足够的智能可以区分出这一点，因为 SenseM.ADCControl 是 StdControl 接口的一个实例，它要与 Photo 提供的 StdControl 接口的一个实例绑定起来。（如果 Photo 提供两个 StdControl 接口，那么这儿就会出错，因为无法确定到底该跟哪一个进行绑定）换言之，语句

```
SenseM.ADControl -> Photo;
```

其实就是语句

```
SenseM.ADControl -> Photo.StdControl;
```

的简写形式！

3 定时器与参数化接口

语句

```
SenseM.Timer -> TimerC.Timer[unique("Timer")];
```

中包含了一种新的语法，称为“**参数化接口（parameterized interface）**”。参数化接口允许一个组件通过赋予运行时或编译时参数之而提供一个接口的多个实例。前面曾提到过，一个组件可提供一个接口的多个实例并给它们分别命以不同的名字，如：

```
provides {
 interface StdControl as fooControl;
 interface StdControl as barControl;
}
```

此处用到的思想与上述思想相同，或者说是同一思想的范化（generalization）。TimerC 组件中声明了这一句：

```
provides interface Timer[uint8_t id];
```

表明它可以提供 256 个 Timer 接口的不同实例，每一个实例对应一个 uint8_t 值！

本例中，希望 TinyOS 应用程序创建和使用多个定时器，且每个定时器都被独立管理。例如，某个应用程序组件可能需要一个定时器以特定的频率（如每秒一次）来触发事件以收集传感器数据；同时另外一个组件需要另一个定时器以不同的频率来管理无线传输。这些组件中每个 Timer 接口分别与 TimerC 中提供的 Timer 接口的不同实例绑定起来，这样每个组件就可以有效地获取它自己“私有”的定时器了。

使用 TimerC.Timer[someval]可以指定 SenseM.Timer 接口应该被绑定到方括号中的值（someval）所指定的 Timer 接口的那个实例。这个值可能是任意一个 8 位的正数。但是若方括号中指定某个特定值，如 38 或 42 等，很可能会导致与其他组件使用的定时器相互冲突（若其他组件的方括号内也使用相同的值）。为解决这个问题，TinyOS 提供了一个编译时函数 unique()，其功能是根据参数字符串产生一个独一无二的 8 位标识。此处

```
unique("Timer")
```

从一组响应的字符串“Timer”产生一个唯一的 8 位数字。只要参数中使用的字符串都相同，就可以保证使用 unique("")的每个组件都得到一个不同的 8 位数值。但是若某个组件使用 unique("Timer")，而另外一个组件使用 unique("MyTimer")，那么它们可能得到相同的数值。因此，当使用 unique()函数时，使用参数化接口本身的名字作为参数不失为一个好的做法。TinyOS 还提供另外一个编译时常数函数

```
uniqueCount("Timer"),
```

利用它可计算出使用 unique("Timer")的总次数。

4 运行 Sense 应用程序

跟前面的例子一样，只需在 Sense 目录下输入命令：

```
make mica install
```

即可编译应用程序并将其安装到微粒中。本例中需要将一个带有照片传感器的传感器主板连接到微粒上。例如 Mica 传感器主板使用 51 针的连接头。传感器主板的类型可以在 ncc 的命令行上使用-board 选项来选择。在 Mica 微粒上，缺省传感器类型时 micasb。若使用老式的“basicsb”传感器主板，须将-board basicsb 选项传给 ncc。可以用如下方法实现，编辑 Sense 目录下的 Makefile 文件，在包含 Makerules 的那一句前面加上这样一行即可：

SENSORBOARD=basicsb。

TinyOS 所支持的所有传感器主板都在 tos/sensorboards 目录下，每个目录对应一个，目录名称与主板名称一致。

这里有必要强调一下 photo 传感器的运行行为，因为它有点特殊。ADC 将照片传感器取得大样本数据转化为 10 位的数字。期望的行为是当节点在光亮处时 LED 关掉，而在黑暗中 LED 发亮。因此将该数据的高三位求反，所以在 SenseM 的函数 ADC.dataReady() 中有如下语句：

display(7 - ((data >> 7) & 0x7));。

5 练习

扩展该应用程序的功能，让传感器主板在黑暗处发出声音。TinyOS 中 Sounder 组件位于 tos/sensorboards/micasb/Sounder.nc 处，它提供 StdControl 接口。修改 SenseM.nc 文件，往其中加入 StdControl 接口并将之命名为 SounderControl；在 Sense.nc 文件中用适当的语句将其导通起来。不要忘记调用 init()函数来初始化 Sounder 组件。在 SenseM.nc 文件中，要使音响打开，需要调用 SounderControl.start()函数；要关闭，则要调用 SounderControl.stop()函数。

Source Tree

apps/

sense/

Sense.nc

SenseM.nc

tos/

interfaces/

ADC.nc

ADCControl.nc

Clock.nc

HPLADC.nc

HPLPot.nc

- Leds.nc
- Pot.nc
- PowerManagement.nc
- StdControl.nc
- Timer.nc

platform/

- avrmote/
  - HPLADCC.nc
  - HPLInit.nc
  - HPLPotC.nc
- mica/
  - HPLClock.nc
  - HPLPowerManagementM.nc

sensorboards/

- micasb/
  - Photo.nc
  - PhotoTemp.nc
  - PhotoTempM.nc

system/

- ADCC.nc
- ADCM.nc
- ClockC.nc
- LedsC.nc
- Main.nc
- NoLeds.nc
- PotC.nc
- PotM.nc
- RealMain.nc
- TimerC.nc
- TimerM.nc

第四章 用于处理应用数据的任务

本课介绍 TinyOS 中“任务（task）”的概念。任务在应用程序中是用来执行某些一般目的的“背景”处理的。本课以 SenseTask 应用程序为例，它是上一课所讲的应用程序 Sense 的修订版。

1 任务的创建和调度

TinyOS 提供由“任务”和“硬件事件句柄”组成的两级调度层次结构。前面讲过关键字 **async** 声明了可被硬件事件句柄执行的命令或事件。这意味着它们可在任何时候执行（可能抢占其他代码的执行），因此 **async** 命令和事件所做的工作应该尽可能地少而且应快速结束。除此之外，还要考虑被异步命令或事件访问的共享数据存在数据竞争使用的可能性。与硬件事件句柄不同，任务用来执行更长时间的处理操作，如背景数据处理等，同时，任务可以被硬件事件句柄所抢占。

任务在实现模块中使用如下语法声明：

```
task void taskname() { ... },
```

其中 taskname()是程序员任意指定的任务名称标识。任务的返回值类型必须是 void，而且不可一带参数。

分派任务的（推后）执行使用的语法形式为：

```
post taskname();。
```

可以从一个命令、事件、甚至是另外一个任务内部“布置（post）”任务。布置操作将任务放入一个以先进先出（FIFO）方式处理的内部任务队列。当某个任务执行时，它会一直运行直至结束，然后下一个任务开始执行。因此，任务不应该被挂起或阻塞太长时间。虽然任务之间不能够相互抢占，但任务可能被硬件事件句柄所抢占。如果要运行一系列较长的操作，应该为每个操作分配一个任务，而不是使用一个过大的任务。

2 SenseTask 应用程序

SenseTask 应用程序在 apps/SenseTask 目录下，它是第二课讲的 Sense 应用程序的改进版。SenseTaskM 组件中包含一个循环的数据缓冲区 rdata，用以存放最近的照片传感器的样本数据；putdata()函数的作用就是将新的样本数据插入到缓冲区中。dataReady()事件只是简单地将数据写进缓冲区，然后启动任务 processData()进行数据的处理。

SenseTaskM.nc

```
// ADC data ready event handler
async event result_t ADC.dataReady(uint16_t data) {
  putdata(data);
  post processData();
  return SUCCESS;
}
```

经过一段时间异步事件完成以后（可能有其他任务挂起等待执行）， processData()任务便得以执行。该任务计算当前 ADC 样本值之和并将其高三位显示在 LED 上。

```
SenseTaskM.nc, continued
  task void processData() {
    int16_t i, sum=0;

    atomic {
    for (i=0; i < size; i++)
      sum += (rdata[i] >> 7);
    }
    display(sum >> log2size);
  }
```

在任务 processData()中使用了关键字 **atomic** ，这样的语句在 nesC 中被称为“原子语句（atomic statement）”。代码中 atomic 的含义是其后花括号内的代码段在执行过程中不可以被抢占。在本例子中，对共享缓冲区 rdata 的访问是要受到保护的。

原子语句会推迟中断处理，从而使得系统的反应看起来不迅速。为了使这种影响降低到最小程度，nesC 中的原子语句应该尽可能地避免调用命令或触发事件，因为外部命令或事件的执行时间还要依赖于与之绑定的其他组件。

3 练习

想办法打断 processData()任务的执行，以便该任务每次被调度时只将 rdata 数组中的一个元素加入到数据之和中。processData()接着应该布置它自己以继续处理全部的数据之和，直到数组中最后一个元素被处理之后将其值显示在 LED 上。请注意这里存在一个并发问题，由于 ADC.dataReady()也可能会布置 processData()任务，因此需要增加一个标志，以防止前面的数据加和还没计算完，一个新的任务实例又启动。

第五章 组件组合与无线通信

本课介绍两个概念：组件图形的层次分解与使用无线通信。作为讲解的例子应用程序是 CntToLedsAndRfm 和 RfmToLeds。 CntToLedsAndRfm 应用程序是 Blink 程序的一个变种，所不同的是它将当前的计数值输出到两个输出接口，它们分别是 LEDs 接口和无线通信堆栈。应用程序 RfmToLeds 的功能是接收无线数据并将之显示在 LED 上。将 CntToLedsAndRfm 应用程序加载到传感器微粒中，它将会通过发射射频信号的无线传输方式将其计数值发送出去；同时，将 RfmToLeds 应用程序载入另一个微粒中，它将会把收到的数值显示在自己的 LED 上。这是一个分布式应用程序。

如果使用的是 mica2 或 mica2dot 微粒，有必要检查一下选择的无线电频率是否与微粒上使用的频率相兼容（433MHz vs 916MHz 微粒）。如果微粒上的标签丢失了，就要去查看电容器 C13（mica2）或 C12（mica2dot） 。若 C13/C12 不是板上组装（即电容器不在板子上面），那么其频段在 868/915MHz 范围内；若是板上组装（即电容器在板子上面），那么设备操作频率范围是 433MHz。到目前为止，还没有处理器可访问的设置方式来查看设备操作频段（如 EEPROM、FLASH 等）。为了让编译器知道正在使用的频率是多少，需要编辑 apps 目录下的 Makelocal 文件，有两种方法：一是定义 CC1K_DEF_PRESET 值（当前值在 tinyos-1.x/tos/platform/mica2/CC1000Const.h 文件中）；二是直接地显式定义频率值 CC1K_DEF_FREQ 。

1 CntToRfmAndLeds 应用程序

该应用程序仅包含一个配置，所有组件模块都在库（tos\lib）中。

CntToLedsAndRfm.nc

```
/**
This application blinks the LEDS as a binary counter and also send
a radio packet sending the current value of the counter.
**/


configuration CntToLedsAndRfm {
}
implementation {
  components Main, Counter, IntToLeds, IntToRfm, TimerC;

  Main.StdControl -> Counter.StdControl;
  Main.StdControl -> IntToLeds.StdControl;
  Main.StdControl -> IntToRfm.StdControl;
  Main.StdControl -> TimerC.StdControl;
  Counter.Timer -> TimerC.Timer[unique("Timer")];
  IntToLeds <- Counter.IntOutput;
  Counter.IntOutput -> IntToRfm;
}
```

使用

```
make mica docs
```

命令可查看其源文件树：

Source Tree

- apps/
  - CntToLedsAndRfm/
    - CntToLedsAndRfm.nc
- tos/
  - interfaces/
    - BareSendMsg.nc
    - ByteComm.nc
    - Clock.nc
    - HPLPot.nc
    - HPLUART.nc
    - IntOutput.nc
    - Leds.nc
    - Pot.nc
    - PowerManagement.nc
    - RadioCoordinator.nc
    - Random.nc
    - ReceiveMsg.nc
    - SendMsg.nc
    - StdControl.nc
    - Timer.nc
    - TokenReceiveMsg.nc
  - lib/
    - Counters/
      - Counter.nc
      - IntToLeds.nc
      - IntToLedsM.nc
      - IntToRfm.nc
      - IntToRfmM.nc
  - platform/
    - avrmote/
      - HPLInit.nc
      - HPLPotC.nc
      - HPLUARTC.nc
      - HPLUARTM.nc
      - InjectMsg.nc

mica/
ChannelMon.nc
ChannelMonC.nc
HPLClock.nc
HPLPowerManagementM.nc
HPLSlavePin.nc
HPLSlavePinC.nc
MicaHighSpeedRadioM.nc
RadioCRCPacket.nc
RadioEncoding.nc
RadioTiming.nc
RadioTimingC.nc
SecDedEncoding.nc
SlavePin.nc
SlavePinC.nc
SlavePinM.nc
SpiByteFifo.nc
SpiByteFifoC.nc
system/
AMStandard.nc
ClockC.nc
FramerAckM.nc
FramerM.nc
GenericComm.nc
LedsC.nc
Main.nc
NoLeds.nc
PotC.nc
PotM.nc
RandomLFSR.nc
RealMain.nc
TimerC.nc
TimerM.nc
UART.nc
UARTFramedPacket.nc
UARTM.nc

该应用程序中所使用的各个组件就一目了然了！

首先值得注意的是一个接口需求（如 Main.StdControl 或 Counter.IntOutput）可能分散到多个实现中去。本例中， Main.StdControl 接口绑定到 Counter、IntToLeds、IntToRfm 以及 TimerC 等组件(除 TimerC 在 tos/System 中，其余组件都在 tos/lib/Counters 目录下)

上。这些组件从其名称上就可以看出其含义：Counter 组件通过接收 Timer.fired()事件来维持一个计数器；IntToLeds 组件和 IntToRfm 组件提供 IntOutput 接口，该接口有一个命令 output()和一个事件 outputComplete()，前者带一个 16 位数值的参数，后者带一个 result_t 类型的参数。IntToLeds 组件将其值的低三位显示在 LED 上，而 IntToRfm 组件将 16 位数值通过无线电广播出去。

本例中，将 Counter.Timer 接口与 TimerC.Timer 接口导通，而 Counter.IntOutput 接口同时与 IntToLeds 和 IntToRfm 组件的响应接口绑定。这样，所有对 Counter.IntOutput.output()命令的调用都将会同时调用 IntToLeds 和 IntToRfm。这里需要注意的是，导通箭头既可以从左指向右，亦可反之。总之，箭头总是从使用接口的组件指向提供接口实现的组件。

假定使用 Mica 微粒，建立和装载该应用程序使用命令

```
make mica install;
```

运行结果是在微粒的 LED 上显示三位计数器，同时微粒还将计数值通过射频信号发送出去了。

2 IntToRfm：发送信息

IntToRfm 是一个简单的组件，它通过 IntOutput 接口接收一个输出值并将其通过无线电广播出去。TinyOS 中的无线通信使用活动消息（Active Message）（AM）模型，在该模型框架下，网络中的每个数据包都指定一个句柄 ID，句柄将在接收节点中被调用。可以把句柄 ID 看作是一个在消息头部中的整数或“端口号”。当接收到消息时，与句柄 ID 相关的接收事件将被触发。不同的微粒可以将相同的句柄 ID 与不同的接收事件相关联。

TinyOS 中，成功的通信包含如下五个方面的要素：

1） 指定要发送的消息数据；
2） 指定哪个节点接收消息；
3） 决定什么时候与输出消息关联的存储器可被重用；
4） 缓存进入的消息；
5） 处理接收到的消息。

在 Tiny 活动消息模型中，存储器管理是非常受限的，因为是在很小的嵌入式环境中使用它。

下面来看一看 IntToRfm.nc 文件的源代码：

IntToRfm.nc

```
configuration IntToRfm
{
  provides {
    interface IntOutput;
    interface StdControl;
  }
}
implementation
{
  components IntToRfmM, GenericComm as Comm;

  IntOutput = IntToRfmM;
```

```
  StdControl = IntToRfmM;

  IntToRfmM.Send -> Comm.SendMsg[AM_INTMSG];
  IntToRfmM.SubControl -> Comm;
}
```

该组件提供了两个接口：IntOutput 和 StdControl 接口。此处与前面的例子不同，在配置里面提供接口。前面的课程中配置仅仅只是将其他组件导通起来，并不提供接口。本例中，IntToRfm 配置本身就是一个可供其他配置导通的组件。

在实现部分，语句

```
components IntToRfmM, GenericComm as Comm;
```

中“GenericComm as Comm”声明了该配置使用了 GenericComm 组件，但给它取了个本地名称 Comm。使用本地名称的好处是，若想使用其他的通信模型，只需简单地将之替换 GenericComm，总共只需更改这一行，而不需要更改与 Comm 相关的每一行。

此例中还使用了新的语法，如：

```
IntOutput = IntToRfmM;
StdControl = IntToRfmM;
```

其中等号（＝）表示的意思是：IntToRfm 提供的 IntOutput 接口**“等同于（equivalent to）”** IntToRfmM 中的实现。此处不能使用箭头（->），因为箭头的含义是将使用接口与提供实现的接口导通起来。本例中，“＝”是将 IntToRfm 提供的接口与 IntToRfmM 中的实现等同起来，这样做了之后，这两个组件中的该接口其实就是指同一个东西了！

该配置中的最后两行是：

```
IntToRfmM.Send -> Comm.SendMsg[AM_INTMSG];
IntToRfmM.StdControl -> Comm;
```

其中最后一行很简单，将 IntToRfmM.StdControl 接口与 GenericComm.StdControl 接口导通起来。而它前面一行显示了参数化接口的另外一种使用方式，其意义是将 IntToRfmM 的 Send 接口与 Comm 提供的 SendMsg 接口导通起来。在 GenericComm 组件中声明提供了 SendMsg 接口：

```
provides {
  ...
  interface SendMsg[uint8_t id];
  ...
}
```

该组件提供了 256 个不同的 SendMsg 接口的实例，每一个实例占用一个 uint8_t 值。活动消息句柄 ID 就是通过这种方式导通在一起的。在 IntToRfm 组件中，将句柄 ID 为 AM_INTMSG 的 SendMsg 接口与 GenericComm.SendMsg 绑定。（AM_INTMSG 是一个全局值，定义在 tos/lib/Counters/IntMsg.h 中）。当命令 SendMsg 被调用时，句柄 ID 作为一个外部参数被传递给它。具体可参见文件 tos/system/AMStandard.nc （该文件是 GenericComm 组件的实现模块）：

```
command result_t SendMsg.send[uint8_t id]( ... ) { ... };
```

当然，此处参数化接口并非完全必需。事实上，句柄 ID 可以作为命令 SendMsg.send 的参数。这里只是为了说明如何在 nesC 中使用参数化接口这个问题。

3 IntToRfmM：实现网络通信

为了了解消息通信是如何实现的，先来看看在 IntToRfmM.nc 文件中 IntOutput.output()命令的定义：

```
IntToRfmM.nc
  bool pending;
  struct TOS_Msg data;
  /* ... */
  command result_t IntOutput.output(uint16_t value) {
    IntMsg *message = (IntMsg *)data.data;
    if (!pending) {
      pending = TRUE;

      message->val = value;
      atomic {
      message->src = TOS_LOCAL_ADDRESS;
      }

      if (call Send.send(TOS_BCAST_ADDR, sizeof(IntMsg), &data))
        return SUCCESS;

      pending = FALSE;
   }
    return FAIL;
  }
```

该命令使用了一个称为 IntMsg 的消息结构（在 tos/lib/Counters/IntMsg.h 中声明的）。它是一个简单的结构体，拥有两个域：val 和 src，前者是数据值，后者是消息的源地址。调用 Send.send()命令时需要三个参数：目的地址、消息大小和消息数据。消息数据的本地源地址字段使用全局常量 TOS_LOCAL_ADDRESS，而目的地址 TOS_BCAST_ADDR 是无线广播地址。

被 SendMsg.send()命令所使用的“原始”消息数据结构是结构体 TOS_Msg，声明在 tos/system/AM.h 文件中。它包含的字段有目的地址、消息类别（AM 句柄 ID）、长度、有效载荷等。最大有效载荷是 TOSH_DATA_LENGTH，其缺省值为 29。本例中，把 IntMsg 封装在 TOS_Msg 结构的有效载荷字段中。

SendMsg.send()命令是分相（split-phase）；当消息发送完成时，它就触发 SendMsg.sendDone()事件。若 send()执行成功，消息将排队以等待发送；若失败，消息发送组件将不能接收到消息。

TinyOS 活动消息缓冲区执行严格的论题所有者协议以避免昂贵的内存管理，然而仍然允许并发操作。如果消息层接收到 send()命令，那么它将拥有发送缓冲区，并且请求组件不应该修改缓冲区直到发送完毕（sendDone()事件发生）

IntToRfmM 模块使用一个挂起（pending）标志来跟踪缓冲区状态。若前面的消息仍在发送，由于不能修改缓冲区，必须放弃 output()操作并返回 FAIL。只有当发送缓冲区可用时才可以填充它并发送消息。

4 GenericComm 网络堆栈

GenericComm 组件是一个“通用”的 TinyOS 网络堆栈实现，位于 tos/system/GenericComm.nc 中。为了实现通信它使用了大量的低级接口：AMStandard 接口用来实现活动消息的发送和接收；UARTNoCRCPacket 用于在微粒的串口上通信；而 RadioCRCPacket 用来无线通信等等。要了解通信的物理细节，可查看这些低级接口的具体实现。如可查看 AMStandard.nc 文件看看 ActiveMessage 层是如何建立的。它实现了 SendMsg.send()命令，在其中布置了一个任务从消息缓冲区中取出消息并通过串行端口（当目的地址为 TOS_UART_ADDR 时）或者射频信号（当目的地址为其他值时）将之传输出去。可以进一步深入查看代码的不同层次，直到发现任何一字节的数据通过无线电或 UART 传输的机制。

5 使用 RfmToLeds 接收消息

RfmToLeds 应用程序由一个简单的配置来定义，它使用 RfmToInt 组件来接收消息，并使用 IntToLeds 组件将接收到的值显示在 LED 上。与 IntToRfm 类似，RfmToInt 组件使用 GenericComm 来接收消息。RfmToInt.nc 中的大部分代码都很简单，不过请注意这一行：

```
RfmToIntM.ReceiveIntMsg -> GenericComm.ReceiveMsg[AM_INTMSG];
```

这句话是用来指定 AM_INTMSG 句柄 ID 接收到的活动消息应该绑定到 RfmToIntM.ReceiveMsg 接口上。这里，箭头的方向可能有的让人迷惑不解。ReceiveMsg 接口位于 tos/interfaces/ReceiveMsg.nc 文件中，它仅仅只声明了一个事件 receive()，该事件将被一个指向接收到的消息的指针所触发。因此，尽管 ReceiveMsg 接口不提供任何可供调用的命令而只定义了一个可被触发的事件，RfmToIntM 组件仍然使用了该接口。

对进入的消息进行的内存管理本质上是动态的。消息到达后进入缓冲区，活动消息层将句柄类型进行解码并将之分发出去。应用程序组件通过 ReceiveMsg.receive()事件获得缓冲区，但是，为可靠起见，应用程序组件在接收完后必须返回一个指向缓冲区的指针，如：

RfmToIntM.nc

```
/* ... */
event TOS_MsgPtr ReceiveIntMsg.receive(TOS_MsgPtr m) {
  IntMsg *message = (IntMsg *)m->data;
  call IntOutput.output(message->val);

  return m;
}
```

由于应用程序已经完成数据处理，所以最后一行将原始消息缓冲区返回。若应用程序某组件需要保存信息内容以备后用，就需要将信息复制到一个新的缓冲区，或者为网络堆栈返回一个新的可用的消息缓冲区。

6 一些细节问题

TinyOS 消息在其头部包含一个“组 ID”，允许多个不同的微粒组共享相同的无线信道。若试验中存在多个微粒组，应该为每个组分配一个独一无二的 8 位数值的组 ID，以避免相

互间的消息发生冲突。缺省情况下，组 ID 值为 0x7D。可以在 Makefile 文件中通过定义预处理符号 DEFAULT_LOCAL_GROUP 来设置组 ID 值，如：

```
DEFAULT_LOCAL_GROUP = 0x42     # for example...。
```

此外，消息头部还有一个 16 位的目的节点地址。在编译时给组内的每个通信节点都要分配一个唯一的节点地址。但下列两个目的地址是受保护的： TOS_BCAST_ADDR (0xfff)——用于作广播地址；TOS_UART_ADDR (0x007e)——用于将消息发送到串行端口。

节点的地址除了上述两个受保护的值外可以是任意其他的数值。给节点指定本地地址使用的语法形式为：

```
make mica install.<addr>,
```

其中*<addr>* 是将要给微粒赋予的本地节点 ID 值。例如：

```
make mica install.38,
```

该命令不仅为某个 mica 微粒编译应用程序，同时还为该微粒设置一个值为 38 的 ID。

7 练习

1、 问题 1：如果同时为多个节点加载了应用程序 CntToLedsAndRfm 并将它们电源打开，结果会如何？

2、 将某个节点加载应用程序 CntToLedsAndRfm，另一个节点加载 RfmToLeds。打开 CntToLedsAndRfm，将可以看到 RfmToLeds 设备上显示邻居节点上的计数值。这两个节点正在实现无线网络通信。可以将它稍作修改以变成一个无线传感器网络并实现其通信吗？（提示：使用 apps/SenseToRfm。）

第六章 使用 TOSSIM 模拟 TinyOS 应用程序

1 TOSSIM 简介

TOSSIM 是 TinyOS 模拟器，它直接由 TinyOS 代码编译而来。使用命令

make pc

建立后，模拟程序直接在本地机器上的桌面运行。TOSSIM 能同时模拟成千上万个节点，模拟中的每个节点运行相同的 TinyOS 程序。

TOSSIM 提供运行时克配置的调试输出，允许使用者从不同角度检视应用程序的执行情况而不需要重新编译。TinyViz 是一个基于 Java 的图形用户界面（GUI），利用它可以可视化地控制和监视程序的运行，检查调试信息、无线电和 UART 数据包等。TOSSIM 还提供了多种与网络进行交互的机制，还可监视数据包的流量、静态或动态地往网络中注入数据包等。

2 建立和运行应用程序

在应用程序目录下输入

make pc

命令即可编译 TOSSIM。除了期望的 TinyOS 组件以外，还有几个与模拟器相关的文件需要编译，这些文件提供对许多功能的支持，如利用 TCP 套接字监视网络等。

进入到 apps/CntToLedsAndRfm 目录，该应用程序运行一个 4Hz 的计数器，并假定每个 mica 微粒有三个 LED。在每个计数器周期，应用程序将计数器的低三位显示在微粒的三个 LED 上，并将整个 16 位数值以一个数据包发送出去。前一课讲述了建立和安装该应用程序，执行后可以看到 LED 在闪烁。

通过命令 make pc 可以得到应用程序的 TOSSIM 模拟版。TOSSIM 的可执行程序是 build/pc/main.exe。可以通过命令

build/pc/main.exe -help

来查看该命令的用法简介。TOSSIM 有一个必需的参数——要参与模拟的节点数目。如输入命令 build/pc/main.exe 1 将运行只有一个节点的模拟程序。启动后，将看到一长串输出流从屏幕滚过，其中大部分是描述无线比特事件的。按下 ctrl＋c 停止模拟。

缺省情况下，TOSSIM 输出所有调试信息。由于无线比特事件的触发频率是 20 或 40 KHz，因此是模拟器中最频繁的事件，CntToLedsAndRfm 中的绝大部分输出就是这些事件的。但用户更关心的是数据包的输出和微粒 LED 的显示值，而非单个的无线比特。TOSSIM 输出可以通过在 shell 中设置 DBG 环境变量而进行配置，如输入命令：

export DBG=am,led,

这句话使得只有 LED 和 AM（活动消息）数据包才可以输出。再运行这个只有一个微粒的应用程序模拟，输出将类似于：

```
0: LEDS: Yellow off.
0: LEDS: Green off.
0: LEDS: Red off.
0: Sending message: ffff, 4
    ff ff 04 7d 08 20 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00
```

```
00 00 00 00 00 00 00 00 00 00 00 00 3b f3 00 00 01 00 00 00
0: LEDS: Yellow off.
0: LEDS: Green off.
0: LEDS: Red on.
0: Sending message: ffff, 4
    ff ff 04 7d 08 21 00 00 00 00 00 00 00 00 00 00 00 00 00 00
00
00 00 00 00 00 00 00 00 00 00 00 00 ac e6 00 00 01 00 00 00
```

数据包中的第六字节包含两字节计数器中的低位字节。本例中，第一个数据包中是十六进制的 0x20（32）；第二个是 0x21 （33）。由于 LED 显示计数器的低三位，因此前一种情况下灯全不亮，第二种情况下只有一个红灯亮。

几乎所有的消息都是以“0”开头的。这意味着该条消息是属于微粒 0 的。若运行两个微粒的模拟（build/pc/main.exe 2），几秒钟后，按 control-C 停止模拟，就可以看到微粒 0 和微粒 1 二者的消息。

把 DBG 设置为 crc，运行 CntToLedsAndRfm 的两个微粒的模拟，可以从输出看到两个节点都成功地从对方接收到数据包。

可以通过命令 build/pc/main.exe -help 来查看 DGB 的所有模式。

3 增加调试语句

TinyOS 为应用程序调试只用保留了四种 DBG 模式：usr1，usr2，usr3，以及 temp。在 TinyOS 代码中，调试消息命令如下：

```
dbg(<mode>, const char* format, ...);
```

其中 mode 参数指定在哪种 DBG 模式下将输出这条消息。在文件 tos/types/dbg_modes.h 中可以找到整套模式。参数 format 及其后面的其他参数指明将要输出的字符串，并且具有与 printf()相同的语义。例如，在编程器中打开 tos/lib/Counters/Counter.nc，在 Timer.fired()中返回语句之前加上这一行语句：

```
dbg(DBG_TEMP, “Counter: Value is %i\n”, (int)state);
```

将 DBG 设置为 temp 并运行一个微粒的模拟将可以看到计数器在增加。

一般而言，在运行模拟器时，TinyOS 代码中 DBG 的模式名是预先指定的名称。例如：am 是 DBG_AM，packet 就是 DBG_PACKET，而 boot 也即是 DBG_BOOT。

在运行模拟器时不仅可以指定多个模式，而且单个调试消息可以在多种模式下激活。在一个大的位掩码中每个模式占一位，并且可以对它们使用所有标准逻辑操作，如|，~。例如：

```
dbg(DBG_TEMP|DBG_USR1, "Counter: Value is %i\n", (int)state);
```

此时，只要 DBG 模式是 temp 或者 usr1，该句就会被打印出来。

dbg函数在tos/types/dbg.h和tos/platform/pc/dbg.c中实现

4 在 TOSSIM 中使用 GDB

TOSSIM 的一个很大的优点就是：由于它在 PC 机上本地运行，所以可以使用如 gdb 等传统的调试工具来调试程序。然而，由于 TOSSIM 是一个为大量微粒进行模拟的离散事件模拟器，所以传统的步进式调试技术只能工作在单个事件的基础上，而不能跨事件。

不幸的是，gdb 是为 C 语言而非 nesC 所设计的；nesC 的组件模型意味着单个命令可以有多个提供者；而引用某个命令需要指定组件、接口和命令。例如：要进入 LedsC 组件的 Leds 接口的 redOff 命令，需输入：

```
 gdb build/pc/main.exe  // start gdb

(gdb) break *LedsC$Leds$redOff
Breakpoint 1 at 0x804c644: file tos/system/LedsC.td, line 97.

run 1                    // run CntToLedsAndRfm with one mote
```

前置的*号是必不可少的，这样 gdb 可直接解析函数名，否则，它会寻找 LedsC 函数。

变量的命名方式类似，例如：为了打印出 LedsC 的 ledsOn 变量，可输入：

```
(gdb) print LedsC$ledsOn
$3 = '\0' <repeats 999 times>
```

事实上，正如以上输出显示的，这是不正确的。在 TOSSIM 中，ledsOn 并非一个简单的 uint8_t 类型变量，而是 1000 个 uint8_t 类型的数组。这是 TOSSIM 处理许多微粒状态的方法，将字段编译到 *n* 个元素的数组，其中 *n* 是最大模拟大小。任何时候某个微粒要访问某个组件的状态，要据其节点 ID 索引数组，因此，要引用某个微粒的状态，需要正确地索引数组：

```
(gdb) print LedsC$ledsOn[tos_state.current_node]
$2 = 0 '\0'
```

有一个简单的 gdb 宏可以处理这个，叫 VAR。将 tos/platform/pc/.gdbinit 复制到主目录，输入 quit 并重新启动 gdb。然后输入

```
(gdb) VAR LedsC$ledsOn
$3 = 0 '\0'
```

5 TinyViz:TOSSIM 用户界面

TinyViz 提供了一个可扩展的图形用户界面，用于测试、显示以及与 TinyOS 应用程序的 TOSSIM 模拟进行交互。使用 TinyViz，可以方便地跟踪 TinyOS 应用程序的执行情况，在感兴趣的事件发生时设置断点，可视化无线消息以及操作微粒的虚拟地点和无线连接性。而且，TinyViz 支持一个简单的“插件”API，允许用户自己编写 TinyViz 模块，以便以一种与应用程序相关的特定方式显示数据或与正在进行的模拟程序交互。

1、 开始

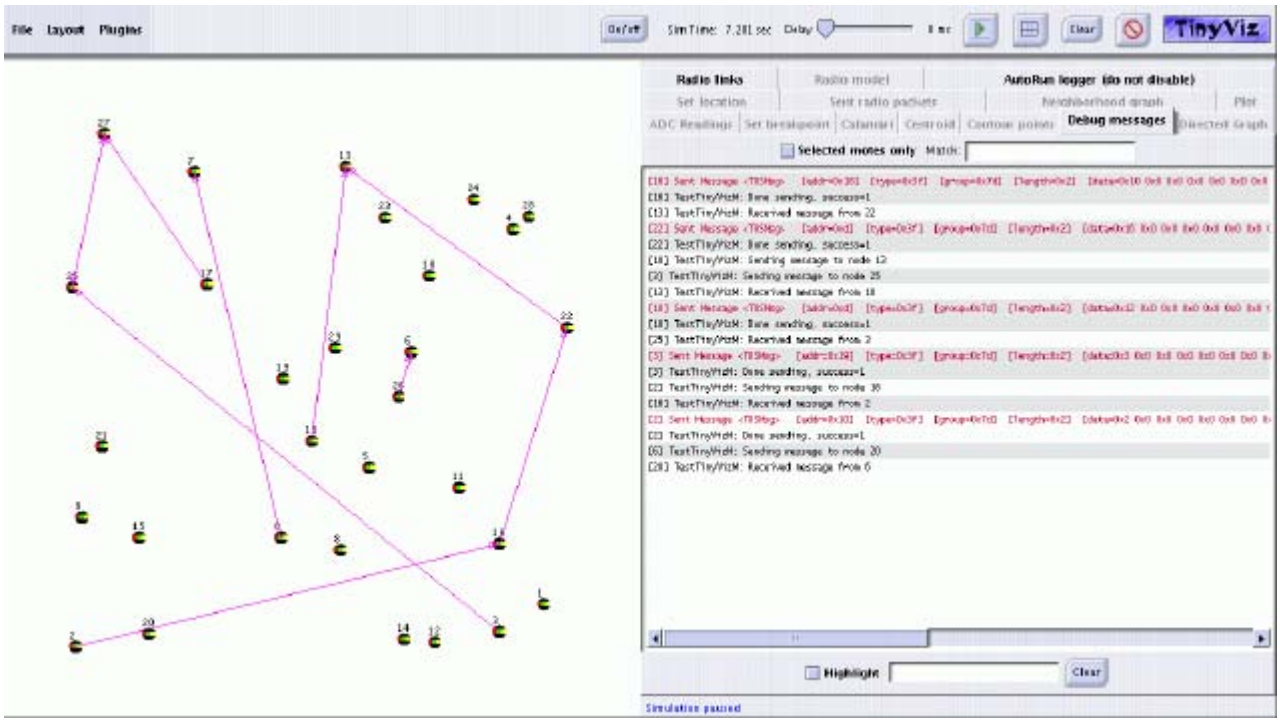
先来看看 apps/TestTinyViz 应用程序，其运行结果时微粒定期地向一个随机邻居发送消息。通过这个例子来展示一下 TinyViz 的特点。使用命令 make pc 建立应用程序。

为了编译 TinyViz，转到 tools/java/net/tinyos/sim 目录下输入 make。TinyOS 将把该程序编译为 tinyviz.jar，这是一个独立的 Java JAR 文件。在该目录下可以使用 tinyviz 脚本来运行它，再将该脚本设置到 PATH 环境变量中，就可以直接运行 Tinyviz 了。

按如下命令启动 TinyViz 并运行 TestTinyViz 应用程序：

```
export DBG=usr1
tinyviz -run build/pc/main.exe 30
```

运行窗口如下所示：



窗口的左半部分是传感器网络的图形显示；其右半部分是一个控制面板，利用它可以与一系列控制 TinyViz 工作的插件进行交互。

在显示微粒的左边窗口中，可以点击某个微粒查看其相关信息，也可以用鼠标拖出一个矩形区域以选择一组微粒，甚至还可以拖着这些微粒四处移动。TinyViz 中某些插件可能对上述这些操作予以响应，如显示节点的电能等。

按钮“pause/play”是一个开关按钮，用于控制模拟程序暂停或者继续执行；按钮“grid button”用于控制网格线的显示；“clear”按钮用于清除显示状态；“stop”按钮用于终止本次模拟；“delay”滑杆用于延迟每个 TOSSIM 事件的处理过程，使显示速度放慢。当微粒数目不太多时，想“实时”观看模拟运行的过程，使用“display”滑杆控制起来十分方便；“On/off”按钮用于控制所选节点的能量状态。

2、**插件**

TinyViz 插件使一种软件模块，用于监视来自于模拟的事件——如调试消息、无线传输消息等等，并通过绘制显示信息、设置模拟参数甚至驱动模拟本身（如为参与模拟的节点设置假定的传感器数值）等方式对所监视的事件予以响应。

TinyViz 本身拥有一套内建插件，在 tools/java/net/tinyos/sim/plugins 目录下。每个使用者还都可编写自己所需的插件。并非所有插件对所有应用程序都有用，例如：Calamari 插件主要使用于测试 Calamari 本地化服务的。当然内建插件中的大部分都是用于一般用途的，对大部分应用程序而言都是有用的。

插件可以通过选择使它起作用或不起作用，是否让某插件在某个模拟过程中起作用取决于使用者的兴趣。可以从 plugins 菜单中选择插件，一旦某个插件被选中，其对应的标签就会在右边的控制面板上激活，就可以在模拟中使用该插件的功能了。所有插件都被设计成相互独立的，因此各插件的使用十分方便灵活。

下面介绍几个主要的插件：

- **调试消息：**这是一个显示由模拟产生的所有调试消息的窗口。用户可以通过选中微粒组来选择查看消息，还可以高亮显示符合特定模式的消息。值得注意的是，用户所看到的调试消息集由 DBG 环境变量决定，正如在一个单机环境中运行 TOSSIM 一样。因此，如果只想查看 DBG_USR1 和 DBG_AM 类型的调试消息，可使用如下命令：

```
DBG=usr1,am tinyviz -run build/pc/main.exe 30
```

- **设置断点：**通过设置断点，当遇到某些特定条件时，模拟就会停下来。此时，可能的条件要么是匹配调试消息中的某个子字符串，要么是匹配所传送的无线消息的内容。用户可以设置多个断点，并可以通过在断点列表中选择使哪些断点起作用，哪些断点不起作用。
- **ADC 数值：**显示每个微粒附近的每个 ADC 信道的最近的数值。
- **发送的无线数据包：**显示所有发送的无线数据包的窗口，有点类似于调试消息插件。值得注意的是：调试消息插件也可以显示这种信息。
- **无线链路：**该插件图形化地显示无线消息活动。当某个微粒在广播消息时，该插件就在这个微粒周围画上一个蓝色的圆圈。而当某个微粒向另外一个微粒发送消息时，就在这两个微粒之间画上一个有向箭头。值得注意的是：不管这些传输是否成功，这些都会显示出来。如果某个微粒试图发送一个消息但失败了或包丢失了，箭头仍将画出。
- **设置位置：**该插件通过用于设置每个微粒的虚拟位置，这要用到 Location 接口，其位置在 apps/TestTinyViz/Location.nc。对每个微粒，设置三个“假造”的 ADC 信道值（分别对应 X 轴，Y 轴，和 Z 轴），这样 FakeLocation 组件就可以来读取这些值从而决定该微粒的虚拟位置。在模拟 TinyOS 应用程序时可以作为实际的位置服务的替代品。
- **无线模型：**该插件通过微粒间的位置及不同的无线链接模型来设置误码率。利用该插件可以在模拟中使用现实的无线链接模型。有两个内建模型：分别是“经验”模型和“固定半径”模型。“经验”模型"通过 RFM1000 射频信号对数据包的连接性进行户外跟踪；而“固定半径”模型中，相互间的距离在一个给定的范围内的所有微粒之间具有良好的连接性，而与其余微粒不相连。可以通过在控制面板中设置“比例因子”来重新设定模型的距离参数。**增加**比例因子将**降低**所选模型的连接范围。通过在显示区选择一个微粒可以查看其到其他节点的连接性——邻近的每条边上显示的数字表示一个数据包通过该链路传输的概率。改变比例因子并点击“更新模型”将更新模型参数，同时显示区的微粒将被移动。（as will moving motes around in the display.）

3、布局

微粒的布局由“布局”菜单所控制，有如下几种布局方式，包括：随机布局、基于网格的布局以及网格加随机布局（在网格布局的基础上再随机扰乱一下）。用户还可以保存以及从文件中加载布局。显示区中微粒的位置有两个用途，其一是当 RadioModelPlugin 插件起作用时用来决定微粒间的连接性；其二是当 LocationPlugin 插件起作用时用来设置微粒的虚拟位置。

4、尝试

下面就来动手体会一下上文提到的 TinyViz 的这些特点。

1. 输入命令 DBG=usr1, tinyviz -run build/pc/main.exe 30，启动 TinyViz。
2. 从 Plugins 菜单项中选择 **Debug messages，Radio links 以及 Set breakpoint** 等几个插件，并点击 pause/play 按钮继续执行模拟。
3. 在控制面板中点击 **Debug messages** 标签将显示由模拟产生的所有调试信息。点击某个微粒（如微粒 3），再点击“Selected motes only”将仅显示该微粒的相关调试信息。
4. 在控制面板底部的输入框中输入某个词语，如“Received”，再点击“Highlight”，将会把匹配该字符串的所有消息高亮显示出来，这对可视化地扫描感兴趣的消息十分有用。
5. 暂停本次模拟，点击“Set breakpoint”标签，将控制面板顶部的“Current breakpoints”横条下拉，选择“Add debug message breakpoint”。在“Message contains” 输入框中输入“Received message”，再点击“Enable breakpoint”。这将添加一个新的断点，每当输出一个匹配的调试信息时模拟程序将暂停。点击 pause/play 按钮继续执行模拟。过不久，模拟程序将暂停，控制面板的底部将显示类似于以下条目的信息：

   Breakpoint 0 fired: Debug message: [24] DebugMsgEvent [24: TestTinyVizM: Received message from 13]

   这意味着该调试信息触发了断点。点击 play 按钮将再一次继续模拟，并再次导致断点。
6. 要想消除该断点，从控制面板顶部的横条中选择“Current breakpoints”，并从断点列表中选中该断点，并点击“Disable breakpoint”即可。
7. 从 Plugins 菜单中选择 **Radio model** 插件，点击控制面板中“Radio model”标签，然后从无线模型列表中选择“Fixed radius (100.0)”，拖动鼠标框住所有微粒，将看到一个非常密集连接的网，这说明每个微粒都几乎与其他微粒相连，这是因为无线模型半径很大的缘故。在“Distance scaling factor”输入框中输入“4”，再点击“Update”按钮，这时无线模型将被更新，其结果是连接性网络将变得稀疏了。

   值得注意的是一旦重新开始模拟，由于断开了连接不能够再通信的节点间仍会继续发送消息，这是因为 TestTinyViz 应用程序要积累发送消息的节点列表，但是改变潜在连接性模型并不修改这个列表。然而，从节点上查看调试信息，将发现它们仅仅只会从与之相连的节点接收消息。

5、自动运行（AutoRun）——编写脚本运行 TOSSIM

TinyViz 的 **AutoRun** 特性允许用户通过在控制 TinyViz 的文件中设置参数来为 TinyOS 模拟的配置和运行编写“脚本”。有了这个功能，用户就可以自动调用插件、设置断点、同时运行多个模拟程序、将数据写入日志文件以及在每次模拟之前或之后执行某些命令等。如果利用 TinyViz 作为分析工具的话，这个特性将十分有用。

在 TestTinyViz 目录下有一个文件 apps/TestTinyViz/sample.autorun，就是一个脚本文件。模拟程序停止运行有三种可能：一是当指定的模拟运行时间用完（选用“numsec”选项时）；二是当指定的子串与调试信息相匹配（选用“stopstring”选项时）；三是模拟程序自己退出（如程序垮掉或显式调用 exit()时）。每个模拟程序的参数由一个空行将其分隔。在一个文件中，如果为某个模拟程序设置了一个参数，该文件中后面的其他模拟程序也将使用该参数，这样就避免了为每一次运行模拟程序而重新指定参数。

下面就是一个自动运行文件的例子：

```
# This is a sample TinyViz autorun file. To use it, run
#   tinyviz -autorun sample.autorun
# Set the layout
layout gridrandom
# Enable some plugins
plugin DebugMsgPlugin
plugin RadioLinkPlugin
plugin RadioModelPlugin
# Total number of simulated seconds to run
numsec 20
# Name of the executable file
executable build/pc/main.exe
# DBG messages to include
dbg usr1
# The radio model and scaling factor to use
radiomodel disc100
radioscaling 5
# Number of motes
nummotes 10
# Command to run before starting
precmd echo "This is a command that will run before the simulation"
# File to log all DBG messages to
logfile logfile-20.txt

# The blank line above indicates that we are starting another
simulation
# This time run with a different number of motes
nummotes 30
logfile logfile-30.txt
```

该文件指定了两个模拟，第一个有 20 个微粒，第二个有 30 个微粒。所有调试信息将分别写入两个不同的日志文件，并指定了几个不同的插件（通过在 tools /java /net /tinyos /sim /plugins 中指定其对应的 Java 类名）。

要使用该 autorun 文件运行模拟程序，只需输入：

```
tinyviz -autorun sample.autorun
```

这时，TinyViz 启动，启用并配置适当的插件，并自动为每个模拟程序运行 10 个模拟秒，然后退出。用户完全可以建立一个 AutoRun 文件运行一系列模拟程序然后去吃午饭，当用户回来时日志文件中所有的数据异准备好了。

自动运行还支持许多其他的特性，用户可以参看 tools/ java/ net /tinyos /sim /AutoRun.java 源文件中的 arConfig 类。值得注意的是文件中指定的所有选项并非被 AutoRun 文件本身使用，但是可用于插件的配置。例如：radiomodel 选项由 RadioModel 插件来配置无线模型。因此，用户可以编写自己的插件并使用 AutoRun 通过这种方式来配置它。

6、绑定 TinyViz 插件

其实TinyViz最有用的特性是可以让用户自己编写插件来与模拟程序交互。下面就是一个简单的绑定插件的例子。先来看一个简单的、文档完好的插件实现tools /java /net /tinyos/sim/plugins/RadioLinkPlugin.java。本质上来说，插件必须提供一个从TOSSIM模拟程序以及TinyViz框架接收事件的方法。插件通过内部状态、更新显示或有可能的话向模拟程序发送回命令等方式反应事件。TinyViz向插件传送初始化、调试消息等事件，还包括无线消息、微粒的位置改变以及新微粒假如到模拟中来等。插件还要提供一些其他方法，以便当该插件被启动或关闭以及微粒窗口被重绘时被调用。

要编写用户自己的插件，直接写Java程序即可，如RadioLinkPlugin.java。当前，所有的插件必须位于TinyViz目录的plugins子目录下（以后的版本会增加一个对“plugin path”的支持）。当修改了某个插件，还必须在TinyViz目录下（tools/java/net/tinyos/sim）通过make命令重新编译tinyviz.jar文件，仅在plugins目录下编译是不够的。

6 将来的用途

本课仅仅只是介绍了TOSSIM的一些功能和用途；例如，由于TOSSIM在比特级别上模拟TinyOS网络栈，因此，无线模型可以一些与此相关的困难问题。相似地，除了用户组件和路由协议外，用户还可以测试和调试较低级别的协议（如开始符号检测等）。TOSSIM System Description更加详细地介绍了这方面的能力并展示了TOSSIM实现的一些信息。

第七章 在 PC 机上显示数据

本课的目标是将传感器网络与 PC 机集成起来，让传感器数据在 PC 机上显示出来并将来自于 PC 的信息传回给微粒。首先，介绍桌面计算机通过串口来读取传感器网络上的数据的一些基本工具；然后展示了一个图形化地显示传感数据的 Java 应用程序；最后，介绍如何将数据传回给微粒。

1、Oscilloscope 应用程序

本课使用的微粒应用程序在 apps/Oscilloscope 目录下。该应用程序仅包含一个从照片传感器上读取数据的模块。每当读取到 10 个传感数据时，该模块就向串口发送一个包含这些数据的包。微粒仅仅只用串口发送数据包。（为了了解如何使用射频信号传输数据请参看 apps/OscilloscopeRF。）

先编译该应用程序并将其安装到一个微粒中。将传感器主板连接到微粒上以便可以获得光强数据。记得要根据传感器主板类型在 apps/Oscilloscope/Makefile 中设置 SENSORBOARD 选项，要么是 micasb，要么是 basicsb。

将带有传感器的微粒连接到与 PC 机串口相连的编程器主板上。值得注意的是，当前使用的 Mica 传感器主板的尺寸不支持将带有传感器的微粒直接插入编程器主板上。一种替代的办法是使用短电缆将编程器主板与传感器主板连接起来。

Oscilloscope 应用程序运行时，如果传感数据超过某一阈值（在代码中设置，缺省为 0x0300），红色的 LED 等将发亮。每当一个数据包被传回给串口时，黄色的 LED 等就发亮。

2、“监听”工具：显示原始数据包中的数据

为了在 PC 机和微粒之间建立通信，首先将串口电缆连接到编程器主板上，并检查 JDK 以及 javax.comm 包是否安装完好。 将 Oscilloscope 代码编译好安装到微粒中后，转到 tools/java 目录下，输入：

```
make
export MOTECOM=serial@serialport:baudrate
```

命令。环境变量 MOTECOM 在这里用于告诉 java Listen 工具要监听哪些数据包。此处 serial@*serialport:baudrate* 的意思是监听连接到串口的微粒，其中 serialport 是连接到编程器主板的串行端口，baudrate 是微粒的波特率。对于 mica 和 mica2dot 微粒，波特率是 19200，mica2 的是 57600 波特。还可以使用微粒名作为波特率，例如：

```
export MOTECOM=serial@COM1:19200 # mica baud rate
export MOTECOM=serial@COM1:mica  # mica baud rate, again
export MOTECOM=serial@COM2:mica2 # the mica2 baud rate, on a different serial port
export MOTECOM=serial@COM3:57600 # explicit mica2 baud rate
```

设置好 MOTECOM 参数后，运行如下命令：

```
java net.tinyos.tools.Listen
```

将得到类似于如下语句的输出信息：

```
% java net.tinyos.tools.Listen
serial@COM1:19200: resynchronising
7e 00 0a 7d 1a 01 00 0a 00 01 00 46 03 8e 03 96 03 96 03 96 03 97 03 97 03 97 03
97 03 97 03
7e 00 0a 7d 1a 01 00 14 00 01 00 96 03 97 03 97 03 98 03 97 03 96 03 97 03 96 03
```

96 03 96 03
7e 00 0a 7d 1a 01 00 1e 00 01 00 98 03 98 03 96 03 97 03 97 03 98 03 96 03 97 03 97 03 97 03
该程序仅仅只是简单地将从串口接收到的每个数据包的原始数据打印出来了。

接下来，执行 unset MOTECOM 命令，以免导致其他所有 Java 应用程序都使用该串口获取数据包。

如果没有正确地安装 javax.comm 包，那么程序将会提示不能够找到串口。如果屏幕上没有类似上面的那些数据输出，原因可能是使用的 COM 端口不对或者微粒到 PC 机之间的连接线路有问题。

3、数据格式分析

该应用程序只是简单地将来自于微粒的数据包打印出来。事实上，每个数据包包含多个数据字段，其中一些是一般的 Active Message 字段，定义在 tos/types/AM.h。消息的数据有效载荷定义在应用程序文件 apps/Oscilloscope/OscopeMsg.h 中。其完整的消息格式如下：

- **Destination address** (2 字节)
- **Active Message handler ID** (1 字节)
- **Group ID** (1 字节)
- **Message length** (1 字节)
- **Payload** (不超过 29 字节):
  - **source mote ID** (2 字节)
  - **sample counter** (2 字节)
  - **ADC channel** (2 字节)
  - **ADC data readings** (每 2 字节 10 个数据)

可以写成如下形式，看起来就一目了然了：

dest addr	handlerID	groupID	msg len	source addr	counter	channel	readings
7e 00	0a	7d	1a	01 00	14 00	01 00	96 03 97 03 97 03 98 03 97 03 96 03 97 03 96 03 96 03 96 03

值得注意的是，微粒发送的数据是 *little-endian* 格式的，例如，两字节数据 96 03 表示一个简单的传感数值，其中 most-significant-byte 字节为 0x03，least-significant-byte 字节为 0x96，即 0x0396 或者十进制数 918。

下面的代码是从 OscilloscopeM.nc 摘录出来的，其中说明了数据是如何写进数据包的：

OscilloscopeM.nc

```
async event result_t ADC.dataReady(uint16_t data) {
 struct OscopeMsg *pack;

 atomic {
   pack = (struct OscopeMsg *)msg[currentMsg].data;

   // add the new sensor reading to the packet
   // and update the number of bytes in this packet
   pack->data[packetReadingNumber++] = data;
```

```
      readingNumber++; // increment total number of bytes

      dbg(DBG_USR1, "data_event\n");

      // if the packet is full, send the packet out
      if (packetReadingNumber == BUFFER_SIZE) {
        post dataTask();
      }
    }

    if (data > 0x0300)
      call Leds.redOn();
    else
      call Leds.redOff();

    return SUCCESS;
  }
  task void dataTask() {
    struct OscopeMsg *pack;
    atomic {
      pack = (struct OscopeMsg *)msg[currentMsg].data;
      packetReadingNumber = 0;
      pack->lastSampleNumber = readingNumber;
    }

    pack->channel = 1;
    pack->sourceMoteID = TOS_LOCAL_ADDRESS;

    /* Try to send the packet. Note that this will return
     * failure immediately if the packet could not be queued for
     * transmission.
     */
    if (call DataMsg.send(TOS_UART_ADDR, sizeof(struct
OscopeMsg),
                                &msg[currentMsg]))
      {
        atomic {
          currentMsg ^= 0x1; // flip-flop between two message
buffers
        }
        call Leds.yellowToggle();
      }
  }
```

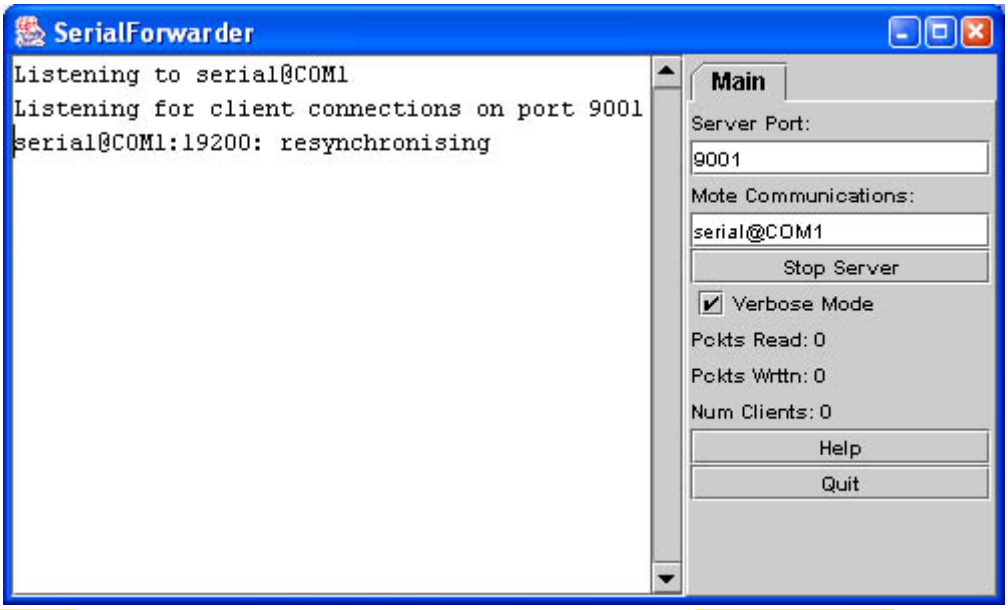
4、SerialForwarder 程序

监听程序是与微粒进行通信的最基本的方式;这种方式要做的事仅仅只是打开串口并将数据包“堆”到屏幕上而已。很明显，使用这种方式不易于将传感数据可视化地展现在用户面前。用户需要的是索取并发现来自于传感器网络的数据的一种更好的办法。

SerialForwarder 程序用来从串口读取数据包的数据并将其在互联网上转发，这样就可以写一些其他的程序通过互联网来与传感器网络进行通信。要运行串口转发器程序，转到 tools/java 目录下并运行如下命令：

```
java net.tinyos.sf.SerialForwarder -comm serial@COM1:<baud rate>
```

一个 GUI 窗口将被打开，看起来类似于：



参数-comm 告诉 SerialForwarder 使用串口 COM1 进行通信；该参数用于指定 SerialForwarder 将要进行转发的数据包来自于何处，使用语法与前面用到过的 MOTECOM 环境变量类似（可以运行 java net.tinyos.packet.BuildSource 以获得一个包含所有来源的列表）。与大多数程序不一样，SerialForwarder 并不理睬 MOTECOM 环境变量；必须使用 -comm 参数来指明数据包的来源（其原理是通过设置 MOTECOM 参数来指定一个串口转发器，串口转发器将与串口通信。不可指望 SerialForwarder 与它自己通信）。

参数<baud rate>用于指定 SerialForwarder 通信时的波特率。

SerialForwarder 并不显示其自身的数据包的数据，而是更新上面窗口右下部分所显示数据包的数量。一旦运行，串口转发器会在某个给定的 TCP 端口（缺省为 9001）监听网络客户端的连接，并简单地将来自于串口的 TinyOS 消息转发到客户端的网络连接上，或者相反。值得注意的是，多个应用程序可以一次同时连接到同一个串口转发器，它们都将从传感器网络获得一份消息拷贝。

要想了解更多的关于 SerialForwarder 以及数据包来源的信息，可参看 SerialForwarder Documentation。

5、启动 Oscilloscope 图形用户界面 GUI

下面来看看图形化地显示来自于微粒的数据。让串口转发器保持运行状态，再执行命令：

```
java net.tinyos.oscope.oscilloscope
```

这时会弹出一个图形化地显示来自于微粒的数据的窗口（如果提示错误信息“端口 COM1 正忙”，可能是因为 Listen 程序执行完后忘了重置 MOTECOM 环境变量）。该程序将通过网络

连接到串口转发器并获取数据，并解析每个数据包的传感数值，然后画出类似于如下所示的图形：

该图形的 X 轴表示数据包的数量值，Y 轴表示传感器的光强数值。如果微粒运行过久，数据包数值会较大，所以图中可能会显示不出传感器数据；这时只需将微粒的电源断开再打开，这样可将数据包计数器重置为 0。如果在显示区看不到传感器光强数据，需将其缩放到适当大小。

6、使用 MIG 与微粒进行通信

MIG (Message Interface Generator)即消息接口产生器，是一个用于自动产生 Java 类的工具，对应于微粒应用程序中使用的活动消息（Active Message）类型。MIG 读取在微粒应用程序中使用的消息类型的 nesC 结构定义，并为每个消息类型产生一个 Java 类，以处理消息字节格式中的各字段打包、拆包等细节问题。使用 MIG 可以避免在 Java 应用程序中解析消息格式的麻烦。

MIG 用来与 net.tinyos.message 包联合起来，并通过 MIG 产生的消息类提供大量发送和接收消息的例程。NCG(nesC Constant Generator) 即 nesC 常数产生器，是一个从 nesC 文件中提取常数以供其他应用程序使用，典型地用于与 MIG 连接。

现在来看看 Oscilloscope 程序中用于与串口转发器通信的代码 tools /java /net /tinyos/oscope/GraphPanel.java。首先，当一个数据包到达时，该程序连接到串口转发器并注册一个用于调用的句柄；所有这些通过 net.tinyos.message.MoteIF 接口来实现：

GraphPanel.java

```
        // OK, connect to the serial forwarder and start receiving
data
        mote = new MoteIF(PrintStreamMessenger.err,
oscilloscope.group_id);
        mote.registerListener(new OscopeMsg(), this);
```

MoteIF 表示一个向微粒发送消息并从微粒接收消息的 Java 接口。串口转发器的主机和端口号可从环境变量 MOTECOM 获取。MoteIF 接口用 PrintStreamMessenger 来初始化，并指明将状态消息（System.err）以及活动消息（可选）组 ID 号发送到何处。这个组 ID 必须与微粒中使用的组 ID。

此处为消息类型 OscopeMsg 注册了一个消息监听器（this）。OscopeMsg 是 MIG 从结构体 OscopeMsg 的 nesC 定义中自动产生的，这个结构体在前面看到过的 OscopeMsg.h 文件中。文件 tools/java/net/tinyos/oscope/Makefile 中展示了该类是如何产生的一个例子：

```
OscopeMsg.java:$(MIG) -java-classname=$(PACKAGE).OscopeMsg $(APP)/OscopeMsg.h
OscopeMsg -o $@
```

本质上，上述语句从头文件 apps/Oscilloscope/OscopeMsg.h 中的消息类型结构 OscopeMsg 产生 OscopeMsg.java。

GraphPanel 实现 MessageListener 接口，该接口定义了从串口转发器接收消息的接口。每当适当类型的消息到来时，GraphPanel 中的 messageReceived()方法就会被调用，如下：

GraphPanel.java

```
  public void messageReceived(int dest_addr, Message msg) {
    if (msg instanceof OscopeMsg) {
      oscopeReceived( dest_addr, (OscopeMsg)msg);
    } else {
      throw new RuntimeException("messageReceived: Got bad
message type: "+msg);
    }
  }

  public void oscopeReceived(int dest_addr, OscopeMsg omsg) {
    boolean foundPlot = false;
    int moteID, packetNum, channelID, channel = -1, i;

    moteID = omsg.get_sourceMoteID();
    channelID = omsg.get_channel();
    packetNum = omsg.get_lastSampleNumber();

    /* ... */
```

方法 messageReceived()需要两个参数：数据包的目的地址以及消息本身（net.tinyos.message.Message）。Message 是应用程序定义的消息类型的基类；本例中实际使用的消息类型是 OscopeMsg。

一旦拥有了 OscopeMsg，就可以利用 get_sourceMoteID()，get_lastSampleNumber()以及 get_channel()等方法从中抽取出想要的字段。不妨仔细查看 OscopeMsg.h 文件中的结构体 OscopeMsg，不难发现其中每个方法对应消息类型中的一个字段：

OscopeMsg.h

```
struct OscopeMsg
{
    uint16_t sourceMoteID;
    uint16_t lastSampleNumber;
    uint16_t channel;
    uint16_t data[BUFFER_SIZE];
};
```

MIG 产生的类中的每个字段至少拥有 8 个与之相关的的方法：

- isSigned_*fieldname* - 指出该字段是否是一个有符号的数量；
- isArray_*fieldname* - 指出该字段是否是一个数组；
- get_*fieldname* - 返回该字段的值；
- set_*fieldname* - 设置该字段的值；
- offset_*fieldname* - 返回该字段的偏移量（字节为单位）；
- offsetBits_*fieldname* -返回该字段的偏移量（比特为单位）；
- size_*fieldname* - 返回该字段的长度（字节为单位）；
- sizeBits_*fieldname* -返回该字段的长度（字比特单位）；

对于类型为数组的字段还会产生一些其他的方法。

方法 messageReceived()的其余部分功能是从消息中分离出传感器数据并将其放在图中。

7、通过 MIG 发送消息

使用 MIG 还可以给微粒发送消息。应用程序 Oscilloscope 可发送 AM_OSCOPERESETMSG 类型的消息，使微粒重置其数据包的数量值。GraphPanel.java 中的 clear_data()方法说明消息如何发送给微粒：

GraphPanel.java

```
          try {
            mote.send(MoteIF.TOS_BCAST_ADDR, new
OscopeResetMsg());
          } catch (IOException ioe) {
            System.err.println("Warning: Got IOException sending
reset message: "+ioe);
            ioe.printStackTrace();
          }
```

要发送消息，只需使用目的地址和消息数据作为参数调用 MoteIF.send()方法即可。此处可使用 MoteIF.TOS_BCAST_ADDR 作为广播地址，它与 nesC 代码中使用的 TOS_BCAST_ADDR 是一回事。

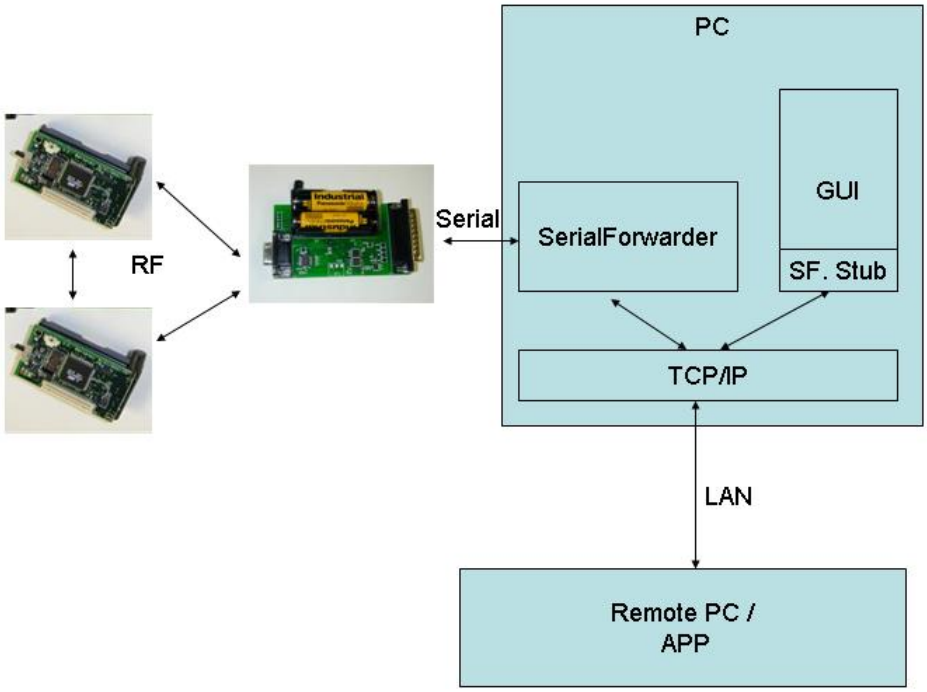
8、练习

通过射频信号将传感器的光强数据传输给另外一个通过串口发送数据的微粒。

应用程序 apps/Oscilloscope 使用串行端口和光传感器；而应用程序 apps/OscillosopeRF 使用无线信号传输传感器数据。可以使用一个微粒作为网关通过射频

信号接收数据并通过串行端口传输这些数据。应用程序 apps/TOSBase 正是作为此用的；它只是简单地在射频信号和 UART（双向）之间转发数据。

Extra Credit: 你能指出如何在 oscilloscope GUI 中同时显示来自于两个微粒的传感器数据吗？事实上，oscillosope GUI 已经具备显示来自于多个微粒的数据的能力，必须确保通过网络正确地传输和接收这些数据。建立方法参阅下图：



第八章：注入和广播数据包

本课讨论利用 PC 往传感器网络中注入数据的简单的 Java 工具以及多跳广播协议的用法。

1、注入数据包

下面演示一个应用程序，它将从射频信号中接收“命令”数据包，解释其含义后执行一系列操作，如打开和关掉 LED 灯。先将应用程序 apps/SimpleCmd 装载到一个微粒中，而将 apps /TOSBase 安装到另一个微粒中；接下来将编程器主板连接到计算机的串口上。将装有 apps/TOSBase 应用程序的微粒放置到编程器主板上。TOSBase 将作为基站；也就是 PC 机与装有 SimpleCmd 应用程序的微粒通信的网关。

现在使用如下命令运行前面讲过的 SerialForwarder：

```
java net.tinyos.sf.SerialForwarder -comm serial@COM1
```

Java 应用程序 tools/java/net/tinyos/tools/BcastInject 用来从 PC 向传感器网络注入命令包，可使用如下命令运行之：

```
java net.tinyos.tools.BcastInject <group_id> <command>
```

其中<group_id>是为该网络所使用的活动消息组 ID 号，以十进制数表示（如缺省组 ID 号是 0x7d，即十进制的 125）。参数 command 有许多选项，但在本课中 SimpleCmd 应用程序理解如下几种：

- led_on - 点亮黄色的 LED 灯；
- led_off - 熄灭黄色的 LED 灯；
- radio_louder - 增加无线发射的电量；
- radio_quieter - 减少无线发射的电量；

使用适当的参数运行 BcastInject 可以点亮或熄灭黄色的 LED 灯。查看 SimpleCmdM.nc 的源代码，可以看出应用程序只是简单地等待消息的到来并将消息中的某个字段解释为消息类型。值得注意的是 BcastInject 和 SimpleCmd 必须在命令类型的含义上相吻合。

2、练习

增加命令使 SimpleCmd 可以打开和关闭扬声器。可进行如下修改：

- 在 SimpleCmd.nc 文件中将 Sounder 组件与应用程序绑定；
- 在 SimpleCmdM.nc 文件中，初始化扬声器并增加可以打开和关闭扬声器的一条新命令。
- 在 SimpleCmdMsg.h 文件中增加一条对应的新命令类型；
- 在 BcastInject.java 文件中，在支持的命令类型集合中增加扬声器开关命令，并确信使用与 SimpleCmdMsg.h 中相同的命令代码。

3、多跳广播

本节将扩展 SimpleCmd 应用程序，使其能够将接收到的命令消息转发给网络中的其他微粒，其实现方法是一旦命令消息处理完毕后再将其广播出去。使用这种方式，可以形成一个简单的多跳路由网络以扩展微粒的通信范围。

先来看看文件 apps/SimpleCmd/BcastM.nc 中的代码。对于接收到的消息中包含的每个命令，除作相应的处理外，再将其广播出去。为了安装 Bcast 应用程序，需要编辑 apps /SimpleCmd/Makefile 文件，将这一行：

COMPONENT=SimpleCmd

改为：

COMPONENT=Bcast

BcastM.nc

```
  event TOS_MsgPtr ReceiveCmdMsg.receive(TOS_MsgPtr pmsg){
    TOS_MsgPtr ret = msg;
    result_t retval;
    struct SimpleCmdMsg *data= (struct SimpleCmdMsg *)pmsg->data;

    // Check if this is a new broadcast message
    call Leds.greenToggle();
    if (is_new_msg(data)) {
      remember_msg(data);
      retval = call ProcessCmd.execute(pmsg) ;

      // Return a message buffer to the lower levels, and hold on
to the
      // current buffer
      ret = msg;
      msg = pmsg;

    }
    return ret;
  }
```

为了判断某个命令消息以前是否接收到过，组件 BcastM 跟踪消息中序列号。如果当前消息的序列号在当前消息的 127 之内，将接收、处理并转发该命令。否则将其丢弃。

值得注意的是 BcastInject 程序在被调用时会维护其序列号（将其保存在文件 tools/java/bcast.properties 中）。如果将该文件移除，BcastInject 产生的序列号将会被重置为 1。如果真是需要如此以便将下一个到来的消息当作“新”的，只需将微粒断电在加电即可。

在无线栈丢弃消息（如栈崩溃时）的情况下，将可能接收到一系列大范围的序列号而非前面的序列号加 1，这是一种公平的未作加工的机制，在事件应用中用户可以加入其他处理过程，如数据包的确认等操作。

4、练习

1. 再来看看 SimpleCmdM.nc 文件，绿色的和红色的 LED 灯用来指明消息的跳数，即消息被转发的次数。将两个微粒安装 Bcast （在 apps/SimpleCmd 目录下）应用程序，并将其放置在适当的位置，使其中之一在 1 跳内接收命令消息，而另一个在 2 跳内接收消息。
2. 如前所述，广播协议十分脆弱：它依赖于 BcastInject 的序列号与微粒的序列号之差不超过 127 的属性。为分辨消息一种更好的办法是创造几个（如 5 个）最近看到的消息的一个索引。若当前接收到的消息不在索引内，该节点就应当转发该消息，BcastInject 就可不加修改的运行，否则就以一个随机的数字发送该消息。

第九章：数据收集应用程序

本课讨论一个远程数据收集和聚集的十分完整的应用程序，称为 SenseLightToLog。该应用程序扩展了 SimpleCmd，接收两个新的命令：一个命令使微粒收集传感器数据并将其写入 EEPROM；而另一个命令从 EEPROM 中读取传感器数据并通过无线电将其传送出去。

1、SenseLightToLog 应用程序

要了解 SenseLightToLog 高级别的功能，可参看 apps/SenseLightToLog/SimpleCmdM.nc 组件。它是 SimpleCmd 的一个扩展版本。任务 cmdInterpret()还要处理另外两个命令：START_SENSING：该命令调用 Sensing 接口，以指定取样速率收集指定数量的样本，并将这些样本数据存入微粒的 EEPROM 中。接口 LoggerWrite 用于将数据写入 EEPROM；READ_LOG：该命令从 EEPROM 中读取一行数据并以无线数据包的形式将其广播出去。

2、Sensing 接口

前面粗略地提到通过 Sensing 接口得到大量传感器数据的概念，这是通过 SenseLightToLog 组件来实现的。该接口提供 start()命令来初始化一系列传感器数据，当传感器工作完毕就触发 done()事件。

```
SenseLightToLogM.nc
  command result_t Sensing.start(int samples, int interval_ms) {
    nsamples = samples;
    call Timer.start(TIMER_REPEAT, interval_ms);
    return SUCCESS;
  }

  event result_t Timer.fired() {
    nsamples--;
    if (nsamples== 0) {
      call Timer.stop();
      signal Sensing.done();
    }
    call Leds.redToggle();
    call ADC.getData();
    return SUCCESS;
  }

  async event result_t ADC.dataReady(uint16_t this_data){
   atomic {
```

```
  int p = head;
  bufferPtr[currentBuffer][p] = this_data;
  head = (p+1);
  if (head == maxdata) head = 0; // Wrap around circular buffer
  if (head == 0) {
     post writeTask();
   }
  }
  return SUCCESS;
}

task void writeTask() {
  char* ptr;
  atomic {
    ptr = (char*)bufferPtr[currentBuffer];
    currentBuffer ^= 0x01; // Toggle between two buffers
  }
  call LoggerWrite.append(ptr);
}
```

当 start()被调用时，计时器启动。当计时器事件触发时，ADC.getData()将被调用 以获取传感器数据。ADC.dataReady()事件将传感器数据存储在一个循环缓冲区内。当收集到适当数目的样本数据时，Sensing.done()事件就会被触发。

当 SimpleCmd 收到一个 READ_LOG 命令时，它就会初始化 EEPROM 读操作（通过 LoggerRead）。当读操作完成时，它就以一个数据包的形式将这个数据广播出去。每个 log 条目有 16 字节长，当运行 BcastInject 工具时就会显示出来。

请注意 ADC.dataReady()同步事件中的原子语句，它们对共享的变量头部、缓冲区指针 bufferPtr 以及当前缓冲区 currentBuffer 等资源实施访问保护。命令 LoggerWrite.append()时被某个任务调用的，而非 ADC.dataReady()同步事件，因为 LoggerWrite.append()并非同步的，因此抢占其他代码并不安全，所以不应该从同步事件中调用。

3、Logger 组件、接口、用法和限制

Mica 微粒拥有一个嵌入的 512 k 字节的闪存 EEPROM，它是微粒的永久存储设备，对包括数据收集在内的许多应用程序来说是必不可少的，如传感数据以及调试跟踪。接口 EEPROMRead 以及 EEPROMWrite 是对对这些硬件的抽象。EEPROM 读写单位为 16 字节的数据块，称为行。读写 EEPROM 是分阶段的操作：必须首先初始化读或写操作，并且在执行另一个操作之前必须等待相应的已做完的事件的到来。

为进一步简化对 EEPROM 的访问，应用程序提供了 Logger 组件（位于 tos/system/LoggerM.nc 中）。Logger 维持一个指向（读或写）下一个 EEPROM 行的指针，将 EEPROM 看作一个循环缓冲区就可顺序地访问它。Logger 不读写 EEPROM 开头的数据，这个地方是预留给微粒存储永久数据之用的。例如，当对微粒进行网络编程时，该区域就用来保存微粒

的 TOS_LOCAL_ADDRESS 值。

接口 LoggerRead 和 LoggerWrite 分别用于读操作和写操作。LoggerRead 提供的命令有：

readNext(buffer) – 从 log 中读取下一行；

read(line, buffer) -从 log 中读取任一行；

resetPointer() – 设置当前行到 log 开头的指针；

setPointer(line) -设置当前行到指定行的指针；

而 LoggerWrite 提供的命令有：

append(buffer) – 添加数据到 log 中

write(line, buffer) – 将数据写道 log 的指定行；

resetPointer() -设置当前行到 log 开头的指针；

setPointer(line) -设置当前行到指定行的指针；

4、收集性能

组件 Logger 并不能提供非常高的性能。如果用户对高频取样（可达 5kHz）感兴趣可参看 apps/HighFrequencySampling 应用程序，其中使用 ByteEEPROM 组件。

5、使用 SenseLightToLog 收集数据

将一个微粒装载 SenseLightToLog 程序，另一个微粒装载 TOSBase 程序，并将传感器主板装到微粒上。

首先向微粒发送指令，让其收集传感数据。输入：

export MOTECOM=serial@COM1:19200，

并运行

java net.tinyos.tools.BcastInject start_sensing <num_samples> <interval>，

其中 num_samples 是取样数目（如 8 或 16），interval 是取样间隔（单位是毫秒），例如

java net.tinyos.tools.BcastInject start_sensing 16 100

当取样时红色的 LED 灯会闪烁。

要得到微粒上的数据，使用

java net.tinyos.tools.BcastInject read_log <mote_address>

其中 mote_address 是要读取的微粒地址，如：

% java net.tinyos.tools.BcastInject read_log 2

Sending payload: 65 6 0 0 0 2 0 0 0 0 0

serial@COM1:19200: resynchronizing

Waiting for response to read_log...

Received log message: Message <LogMsg> [sourceaddr=0x2]

Log values: 48 1 38 1 33 1 32 1 32 1 33 1 34 1 34 1

该程序将为响应的 read_log 命令等待 10 秒；若没有响应，请再试。若根本没有应答，可能是微粒没有得到命令（每当接收到一个命令时绿灯会亮），或者是微粒地址指定不争取。 每发送一个 read_log 命令将从微粒中读取下一个条目；若要重置读指针，只需将微粒重启。 EEPROM 中的数据是永久数据，但当前读指针是保存在瞬时存储器中的。

第十章 TinyDB：一种用于无线传感微粒的声明式查询系统

1 简介

本文主要介绍 TinyDB 查询处理系统的用法，该系统使用一种声明式查询接口从无线传感微粒中析取数据，这种接口类似于关系数据库系统中的 SQL 接口。本文假定读者已经熟悉 TinyOS 工具集，但不要求具有 C 语言的代码经验。本文的最后部分讲述用户如何在自己的软件中编写简单的 Java 程序来使用 TinyDB。

2 TinyDB

TinyDB 是一种从无线传感器网络中析取信息的查询处理系统。与 TinyOS 中其他数据处理解决方案不同的是，TinyDB 不需要用户编写嵌入式的 C 语言代码。相反，TinyDB 提供了一种简单、类似于 SQL 的接口来指定需要析取的数据，并为查询提供所需的参数，如数据更新的频率等，——一切与向传统数据库提交查询一样简单方便。对于任意一个查询，只需指定感兴趣的数据，TinyDB 就会从传感微粒中收集那些数据，并将之过滤、聚集以及选择路由，最终送到 PC 机。TinyDB 是通过节能的嵌入网络的处理算法做到这一点的。

要使用 TinyDB，必须将其 TinyOS 组件安装到传感器网络中的每个微粒上。TinyDB 为查询和析取数据的 PC 应用程序提供了一套简单的 Java API；同时还提供了一个使用该 API 的简单的图形化查询工具和结果显示界面。

TinyDB 的最主要目标就是期望简化程序员的工作，使得数据驱动的应用程序开发和部署尽可能地快速。有了 TinyDB，可以使程序员从一些细致繁琐的工作中解脱出来，不必再纠缠于包括传感器网络接口在内的处理传感器设备的较低级别的代码。

3 安装 TinyDB 并运行简单的查询

下面的例子需要使用三个传感器微粒。在每个微粒中安装 TinyDBApp 应用程序，并分别设置他们的 ID 为 0，1 和 2。打开三个微粒的电源，并将编号为 0 的微粒连接到 PC 机的串行端口上。（指定微粒的编号使用的命令为 make mica install.*nodeid*，其中 *nodeid* 即是想要编给微粒的号码）。

可以使用 tools/java/net/tinyos/tinydb 目录下的 TinyDBMain 类与这些微粒进行交互。先来编译这些 Java 类，在此之前，要确保以下几个包已经在环境变量 CLASSPATH 中：JLex.jar，cup.jar 以及 plot.jar；这三个包都在 tools/java/jars 中。在 tools/java/目录下有一个叫作“javapath”的小程序，可以通过运行它来设置 classpath 环境变量，只需将 CLASSPATH 的值置为该命令的输出即可（它会将新目录和 jars 设置到当前的 CLASSPATH 环境变量中）。在 bash（Cygwin 或 Linux）下输入命令：

```
export CLASSPATH=`path/to/tinyos/tools/java/javapath`
```

若是在 sh 或 csh 下，只需将“setenv CLASSPATH ...”代替“export CLASSPATH=...”即可。

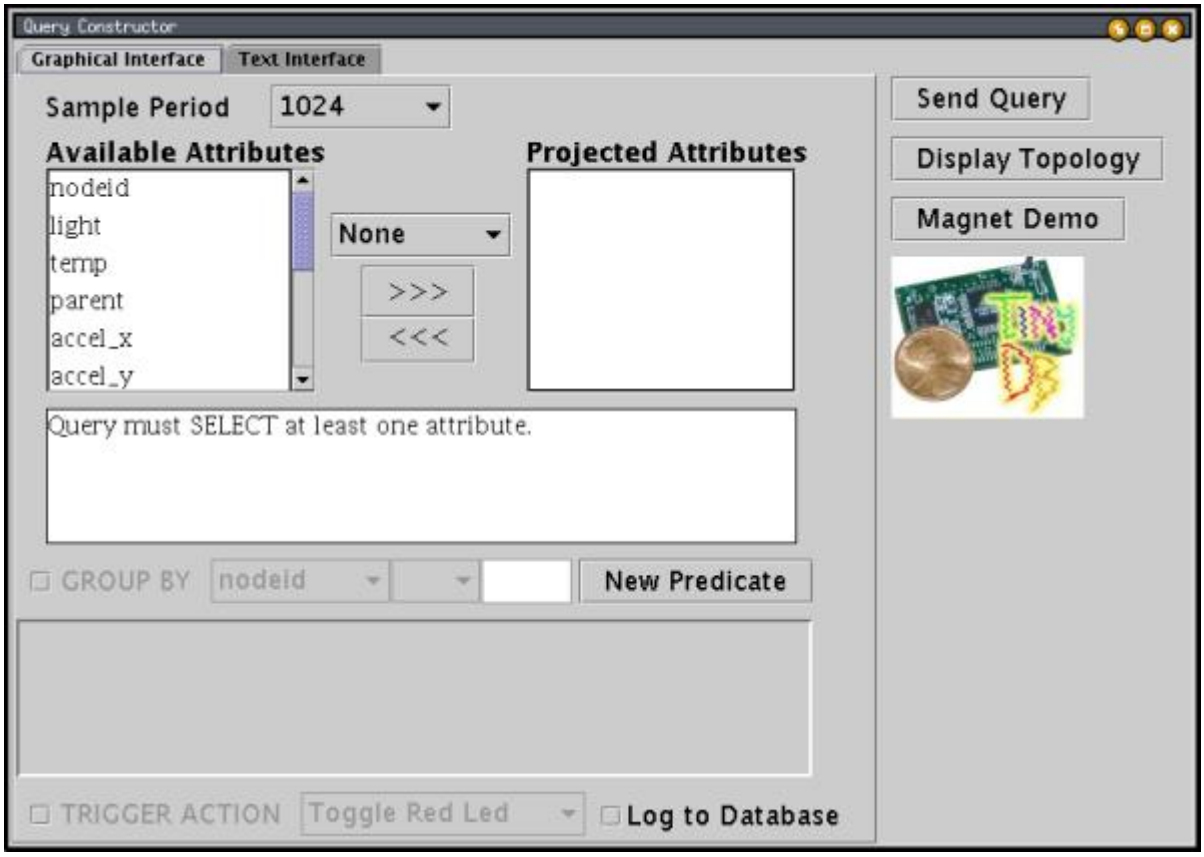
环境变量设置好后，就可编译 Java 类了，输入命令：

```
cd path/to/tinyos/tools/java/net/tinyos/tinydb
make
```

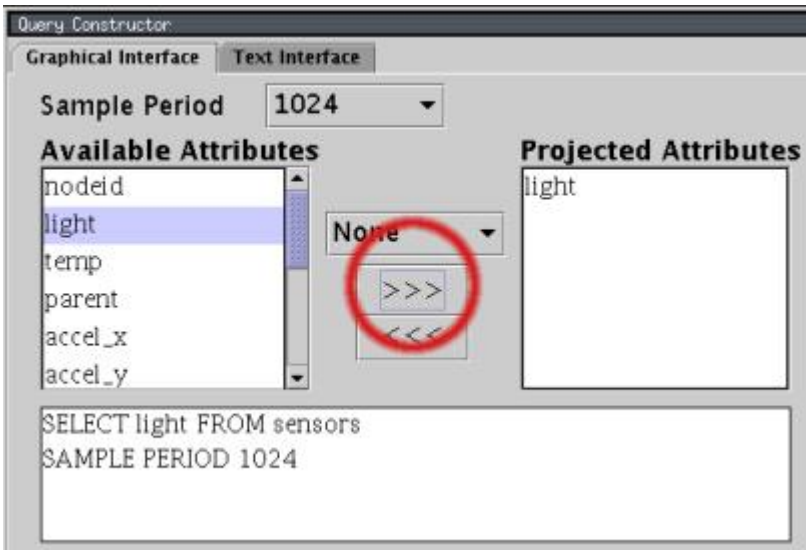
该命令需要执行几分钟，当编译 TinyDB 查询解析器时还会输出许多文本。命令执行完毕后，就可以启动 GUI 了！转到 tools/java 目录下，输入：

```
cd ../../..
java net.tinyos.tinydb.TinyDBMain
```

TinyDB GUI 就会显示如下：

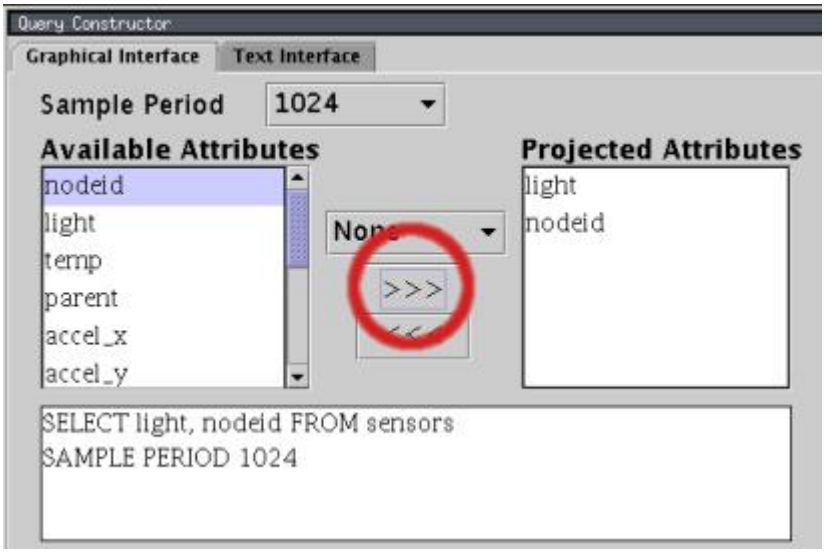


要在 GUI 中指定一个查询，只需从左边的“Available Attributes”的列表项中选择感兴趣的字段将其移动到右边的“Projected Attributes”中。如：要选择 light 属性，首先在左边选中“light”，再点击“>>>”按钮，显示如下：

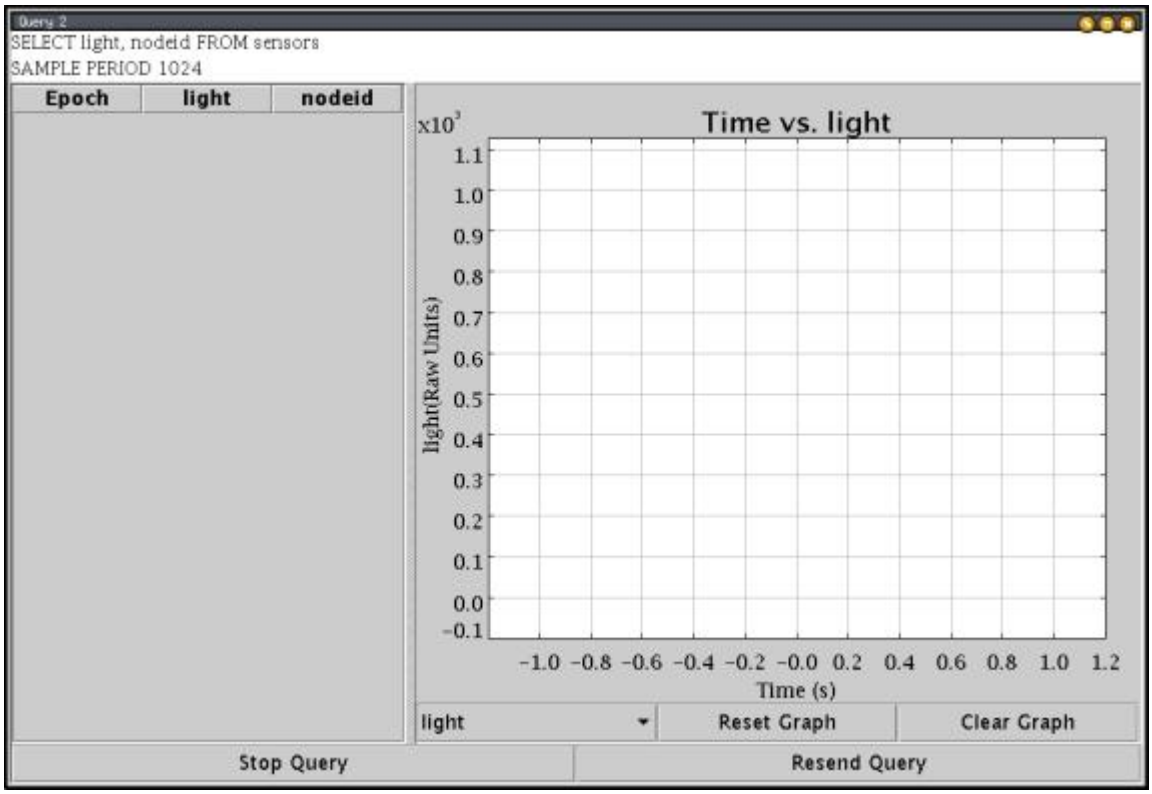


请注意属性列表下面的查询文本会随着查询条件的修改而更新。其中，SAMPLE PERIOD 子句的意思是每隔 1024 毫秒读取一个新的光强数据。用户可以在窗口顶端通过弹出式菜单更改取样间隔。

点击左边列表中的“nodeid”，再点击“>>>”按钮，显示如下：

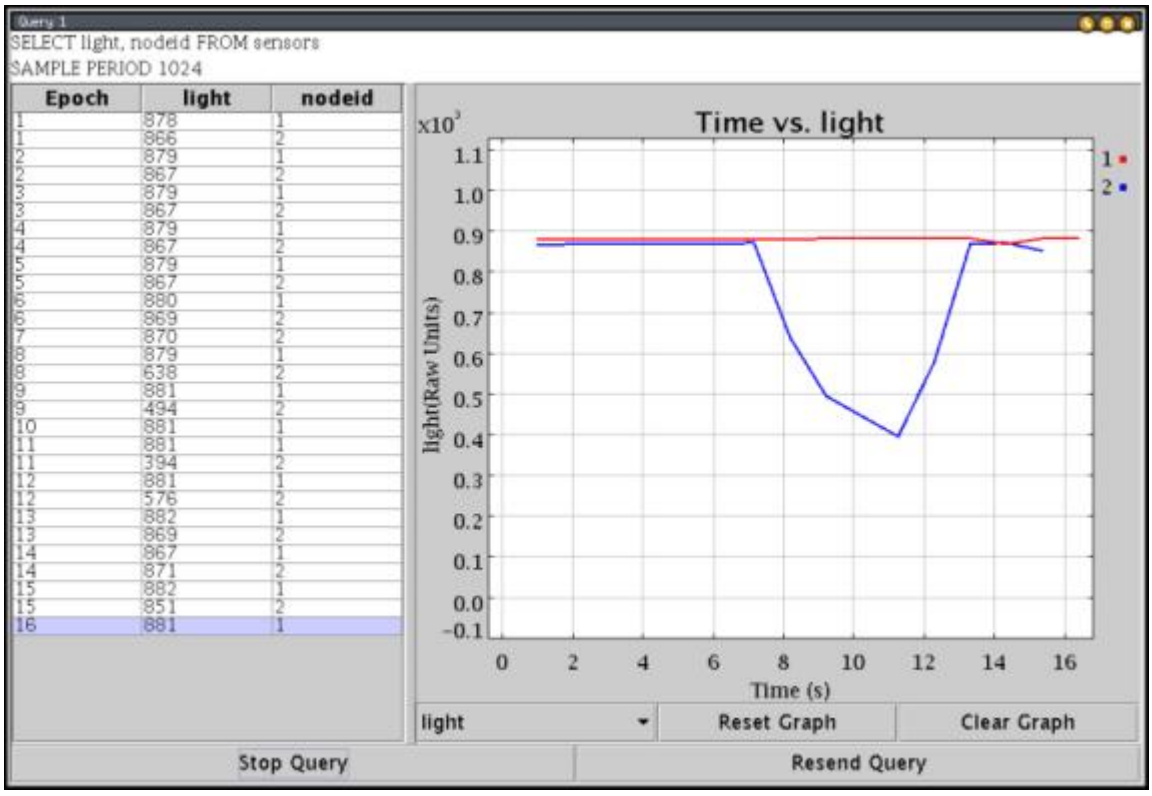


与光强值 light 不同，nodeid 并非一个物理上的传感器数据，而是通过命令“make mica install.nodeid”写入微粒的 ID 号。本文后面还会讲述如何自己编写属性以扩展 TinyDB 的功能。下面先来看看如何运行一个查询过程。点击“Send Query”按钮；结果窗口如下所示：



当上面这个窗口显示时，ID 号为 0 的微粒（基站）上的红色 LED 灯会闪烁几次，随后微粒 1 和微粒 2 上的红色 LED 灯会点亮。再过几秒中后，三个微粒上的黄色 LED 灯会以每秒 1 次的频率闪烁——这意味着查询正在正确的执行。如果 LED 灯在几秒内仍不开始闪烁，点击“Resend Query”以重新提交这个查询。

查询结果会传回到 GUI 中，显示微粒 1 和微粒 2 的光强数据。将微粒 2 遮盖起来，可以看到图中表示其数值的线会呈下降形式：



以上只是对 TinyDB 的一个简单介绍。下一节将将介绍 TinyDB 的高级特性，以适应更大范围内的数据聚集应用。

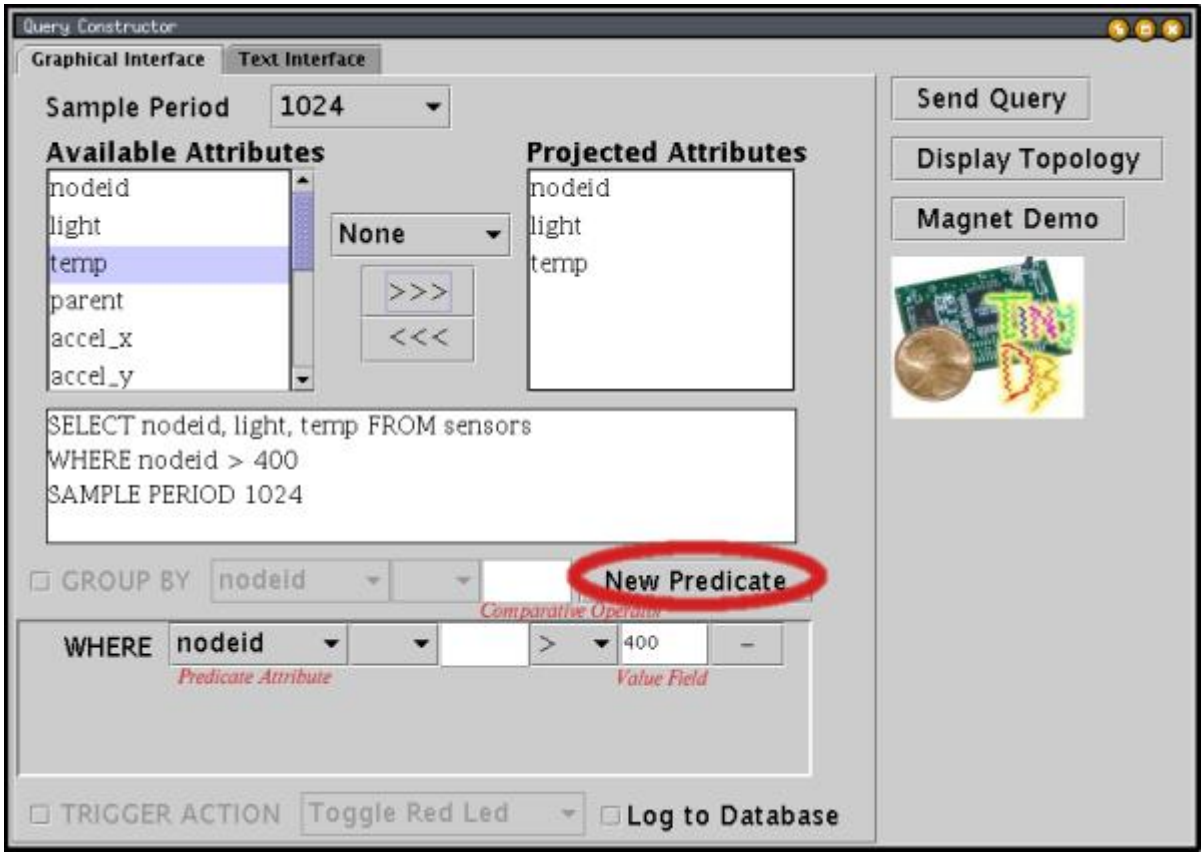
4 TinyDB 高级特性

TinyDB 包含许多其他的高级特性，有兴趣的读者可以参阅《TinyDB 参考手册》。本节将简要描述该查询语言的一些特性。

1. WHERE 子句：TinyDB 的查询语句可以包含一个 WHERE 子句以过滤掉那些不感兴趣的数据。例如：

```
SELECT
nodeid,light,temp FROM
sensors WHERE
light >  400
SAMPLE PERIOD 1024
```

要得到这样一个查询语句，使用“New Predicate”按钮添加一个谓词，从谓词属性菜单中选择“light”，从比较操作符菜单中选择“>”，并在数值字段中输入 400，如下所示：



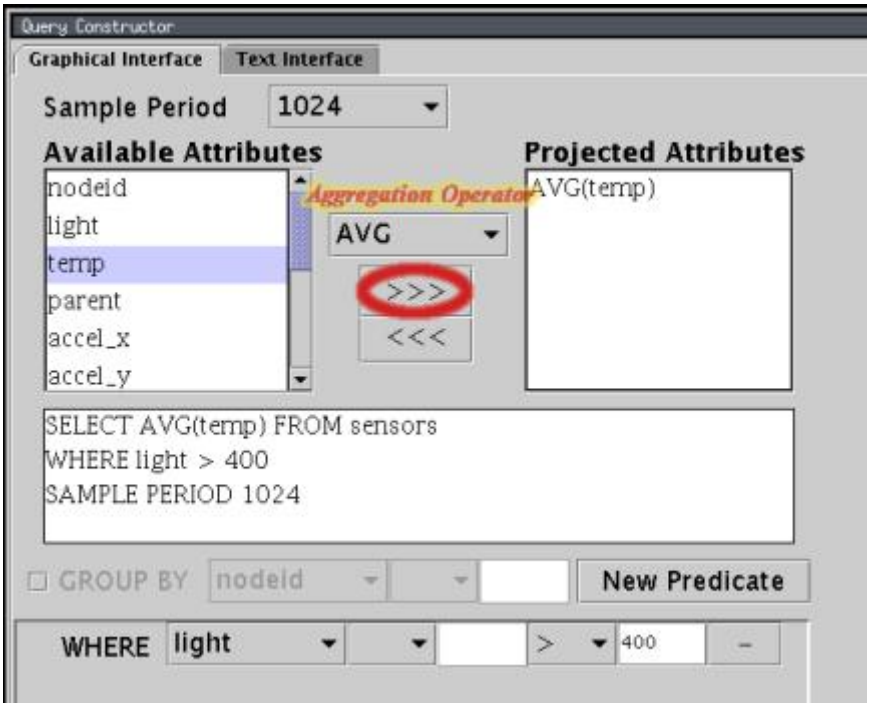
使用 WHERE 子句作用于 nodeid 字段可以对网络的一个自己进行查询。

2. 聚集谓词（Aggregation Predicates）：TinyDB 也可以对一个查询中的多个数据值进行聚集运算。例如，计算所有光强值大于 400 的微粒的平均温度值的语句为：

```
SELECT
AVG(temp) FROM
```

```
sensors WHERE
light >  400
SAMPLE PERIOD 1024
```

要得到上述语句，在聚集操作符菜单上选择“AVG”，在将 temp 字段移动到映射属性列表中：



TinyDB 通过一种有效的内置于网络的方法计算该查询语句，传感器会将其自身的数据与来自于邻居节点的数据聚集在一起然后再转发给基站。

本节演示了如何执行一个包含过滤谓词和聚集谓词的查询。下一节，将讲述如何编写一个小型的、独立的程序来运行 TinyDB 查询。

5 使用 TinyDB 的一个简单的 Java 程序

本节将介绍如何编写一个简单独立的 Java 程序以便在 TinyDB 中运行查询。该程序使用的查询语句为：SELECT light FROM sensors。下面先来看看整个程序，然后再逐部分讲解（该程序位于 tools/java/net/tinyos/tinydb/DemoApp.java）：

```
    package net.tinyos.tinydb;
    import net.tinyos.tinydb.parser.*;
    import java.util.Vector;
    import java.io.*;
    public class DemoApp implements ResultListener{
        public DemoApp() {
            try {
                TinyDBMain.initMain(); //parse the query
                q = SensorQueryer.translateQuery("SELECT light",
(byte)1);
                //inject the query, registering ourselves as a
```

```
listener for result
                System.out.println("Sending query.");
                TinyDBMain.injectQuery( q, this);
            } catch (IOException e) {
                System.out.println("Network error.");
            } catch (ParseException e) {
                System.out.println("Invalid Query.");
            }
        }

        /* ResultListener method called whenever a result arrives
*/
        public void addResult(QueryResult qr) {
            Vector v = qr.resultVector(); //print the result
            for (int i = 0;  i <  v.size(); i++) {
                System.out.print("\t" + v.elementAt(i) + "\t|");
            }
            System.out.println();
        }

        public static void main(String argv[]) {
            new DemoApp();
        }

        TinyDBQuery q;
    }
```

要运行该程序，按前面讲述的建立好微粒并确保关闭所有打开的 TinyDB 窗口，转到 tools/java/目录下输入 java.net.tinyos.tinydb.DemoApp，输出如下所示：

```
Listening for client connections on port 9000
SerialPortIO: initializing
Successfully opened COM1
client connected from localhost.localdomain (127.0.0.1)
Sending query.
        1       |       835     |
        2       |       833     |
        3       |       833     |
        4       |       833     |
        5       |       833     |
        6       |       833     |
        7       |       833     |
...
```

下面，来详细地看看这个程序的一些细节。首先，DemoApp 类实现了 ResultListener 接口。方法 The addResult(...)是该接口的唯一成员，DemoApp 注册的任何一个查询的运行结果到来时，该方法都会被调用。稍后再来看看注册是如何工作的。

构造函数 DemoApp()的第一行用于初始化 TinyDB,随后的语句分别用于解析查询语句和将查询语句注入网络：

```
    TinyDBMain.initMain();

    //parse the query
    q = SensorQueryer.translateQuery("SELECT light", (byte)1);

    //inject the query, registering ourselves as a listener for
result
    System.out.println("Sending query.");
    TinyDBMain.injectQuery( q, this);
```

调用 TinyDB.initMain()是为了读取 TinyDB 配置文件并建立网络通信。缺省情况下，它独立打开一个到串口的连接，尽管可以配置它使其通过 SerialForwarder 来共享这个连接。

接下来调用 SensorQueryer.translateQuery(..)函数，将指定的 SQL 查询语句转换成一个 TinyDBQuery 对象；其中第二个参数((byte)1)说明该查询的查询 ID 为 1；查询 ID 的作用是在查询命令注入到网络之后可以修改或取消该查询。

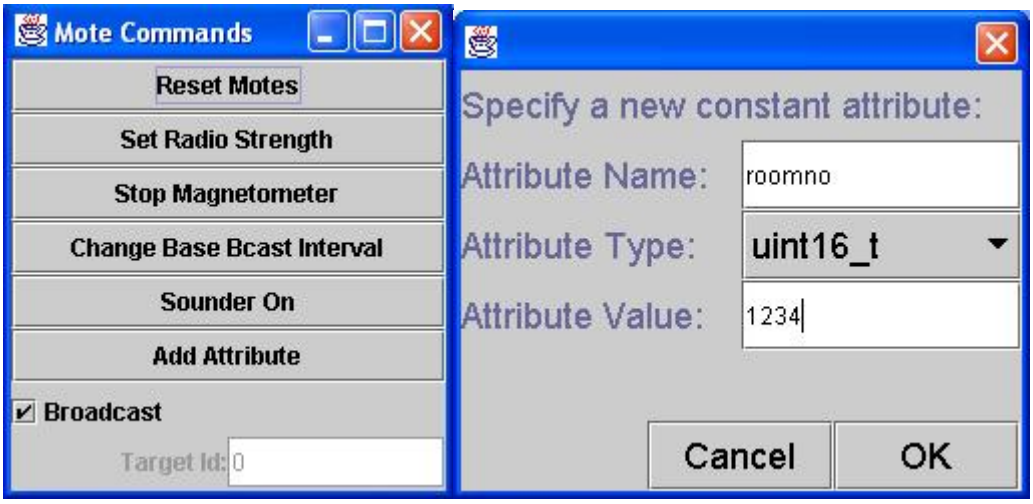
最后 TinyDBMain.injectQuery(..)用于将查询发送给网络并启动其运行。第二个参数 this 指明 DemoApp 对象将注册为一个监听器以获得查询结果。

此时，查询正在执行。每当一个查询结果到来时，addResult 方法将被调用以打印出到来的查询结果的每一个字段。

从上面例子可以看到如何使用 TinyDB GUI 以及编写与 TinyDB 微粒交互的简单应用程序。下一节，将讲述如何通过 TinyDB 查询界面扩展可被查询的 TinyDB 的新属性。

6 增加一个属性

TinyDB 具有一个内建命令，可用于增加常数属性。在微粒命令窗口（Mote Commands window）中点击“Add Attribute”按钮，将弹出一个对话框，要求用户填写属性名称，类型和常数值。若微粒接收到“Add Attribute”命令，其绿色的 LED 灯将会闪烁，表示新属性可以使用了。



如果是要为某个特定的微粒添加一个属性的话，不要选中在微粒命令窗口中的“Broadcast”复选框，再填上一个目标 ID。如果常数属性与已注册属性重名，那么该属性先前的数值将会被新数值覆盖。

要增加非常数属性，需要使用 TinySchema API 来编写 NesC 程序。最简单的学习方法是复制和修改 tos/lib/Attributes 中实现的一些内建属性。