Installing TinyOS 2.0.2

Last updated 30 July 2007

If you already have a 1.x tree or an existing 2.x tree, you are better off following the *upgrade* instructions at **upgrade-tinyos.html**. There are two ways to install TinyOS. The first way is to install a live CD that gives you a virtualized Linux with a complete TinyOS install. Note that since this is on a CD, you can't modify anything; you can, however, make a LiveUSB device to use as your TinyOS install. The second way is to install TinyOS on your host operating system. When installing on a host operating system, you can either use a debian package repository or manually install with RPMs.

One-step Install with a Live CD

Download a Linux live CD that has a TinyOS installation on it. All you need to do is download the CD image, burn it onto a CD, and install from there. This saves you all of the complexities of installation, and it's the recommended way to install TinyOS. The link above has complete instructions. The live CD is provided by the Toilers group at the Colorado School of Mines.

Two-step install on your host OS with Debian packages

If you are running a version of Linux that supports Debian packages, then you may want to use the TinyOS package repository. There is a **story on www.tinyos.net** that describes how to use it. If you do this, then you do not have to install the instructions here, except that you will need to check that your environment is set up correctly (the end of step 5 in the manual installation.

Manual installation on your host OS with RPMs

Currently, the TinyOS Core Working Group supports TinyOS on two platforms: Cygwin (Windows) and Linux. There have been some **successful efforts** to getting TinyOS environments working on Mac OSX, but OSX is not supported by the Core WG.

Installing a TinyOS enviromnent has five basic steps; Windows requires an extra step, installing Cygwin, which is a UNIX-like environment. The steps are:

1. **Installing a Java 1.5 (Java 5) JDK.** Java is the most common way of interacting with mote base stations or gateways that are plugged into a PC or laptop.

2. **Windows only. Install Cygwin.** This gives you a shell and many UNIX tools which the TinyOS environment uses, such as perl and shell scripts.
3. **Installing native compilers.** As you're compiling code for low-power microcontrollers, you need compilers that can generate the proper assembly code. If you using mica-family motes, you need the AVR toolchain; if you're using telos-family motes, you need the MSP430 toolchain.
4. **Installing the nesC compiler.** TinyOS is written in nesC, a dialect of C with support for the TinyOS concurrency model and component-based programming. The nesC compiler is platform-independent: it passes its output to the native compilers, so that it can take advantage of all of the effort put into their optimizations.
5. **Installing the TinyOS source tree.** If you want to compile and install TinyOS programs, you need the code.
6. **Installing the Graphviz visualization tool.** The TinyOS environment includes `nesdoc`, a tool that automatically generates HTML documentation from source code. Part of this process involves drawing diagrams that show the relationships between different TinyOS components. **Graphviz** is an open source tool that nesdoc uses to draw the diagrams.

Step 1: Install Java 1.5 JDK

Windows Download and install Sun's 1.5 JDK from **http://java.sun.com**.

Linux Download and install IBM's 1.5 JDK from **http://www-128.ibm.com/developerworks/java/jdk/**.

Step 2: Install Cygwin

This step is required for Windows installations only. If you are installing on Linux, skip to step 3.

We have put online the cygwin packages that we've confirmed to be compatible with TinyOS. The instructions below use those packages. You can also upgrade your cygwin environment according to the instructions at www.cygwin.com and your environment will most likely work. A large number of TinyOS users, upgrade their cygwin packages at least monthly from cygnus. However, since we can't test what packages are compatible as they become available daily, we can't confirm that today's set will work.

1. Download and install Cygwin from **www.cygwin.com**.
2. Download the confirmed-compatible cygwin packages from the tinyos web site **here** .
3. In a cygwin shell, unzip the above package into some directory. In these instructions the directory is /cygdrive/c/newcygpkgs.

```
$ cd /cygdrive/c/newcygpkgs
$ tar zxvf cygwin-1.2a.tgz
```

This unzips the packages.

6. In Windows Explorer, navigate to /cygdrive/c/newcygpkgs and click on the file setup.exe. Setup.exe is the setup program distributed by Cygnus Solutions.
7. Follow these steps when the Cygwin Setup windows appears:
8. Opt to disable the virus scanner (it will be enabled when you're finished).
9. Opt to Install from Local Directory.
10. Specify the Root directory to be where your *current* cygwin installation is. This would be the directory that directories like 'opt' and 'usr' are in. For example, mine is rooted at c:\tinyos\cygwin, so I enter that.
11. Select to Install for All Users
12. Select the Unix file type (very important!)
13. For the Local Packages Directory, specify where you unzipped the cygwin packages tarfile. For example, I would specify c:\newcygpkgs. (The setup.exe program will probably select the right default directory.)
14. The next window will allow you to select packages to install. You should see that most of the packages have an X-ed box next to them; these are the packages that are to be installed.
15. Click install.

Some notes:

- You might see a message explaining that you need to reboot because some files are in use. This most likely means that your cygwin DLL is loaded and in-use and, therefore, cannot be replaced. When you reboot, the new DLL will be loaded.
- Related to the above warnings, if you see warnings about the cygwin1.dll not being found, don't worry. All will be well once you reboot and the right DLL is loaded.

Step 3: Install native compilers

Install the appropriate version of the following (Windows or Linux, avr or msp430 or both) with the rpm command 'rpm -ivh *rpm>*'. On windows, if you get an error claiming that the rpm was build for an NT computer and you're on a windows NT computer, bypass the erroneous error by using 'rpm -ivh --ignoreos *rpmname*'. (We have xscale compiler tools online at **http://www.tinyos.net/dist-1.2.0/tools/** but they have not yet been extensively tested by a large community.)

Atmel AVR Tools

Tool	Windows/Cygwin	Linux
avr-binutils†	**avr-binutils-2.15tinyos-3.cygwin.i386.rpm**	**avr-binutils-2.15tinyos-3.i386.rpm**
avr-gcc	**avr-gcc-3.4.3-1.cygwin.i386.rpm**	**avr-gcc-3.4.3-1.i386.rpm**
avr-libc	**avr-libc-1.2.3-1.cygwin.i386.rpm**	**avr-libc-1.2.3-1.i386.rpm**
avarice	**avarice-2.4-1.cygwin.i386.rpm**	**avarice-2.4-1.i386.rpm**

insight (avr-gdb) avr-insight-6.3-1.cygwin.i386.rpm avr-insight-6.3-1.i386.rpm

†*If you receive an rpm error that indicates that you have a newer version already installed, try* `rpm -Uvh --force`

TI MSP430 Tools

Tool	Windows/Cygwin	Linux
base	msp430tools-base-0.1-20050607.cygwin.i386.rpm	msp430tools-base-0.1-20050607.i386.rpm
python tools	msp430tools-python-tools-1.0-1.cygwin.noarch.rpm	msp430tools-python-tools-1.0-1.noarch.rpm
binutils	msp430tools-binutils-2.16-20050607.cygwin.i386.rpm	msp430tools-binutils-2.16-20050607.i386.rpm
gcc	msp430tools-gcc-3.2.3-20050607.cygwin.i386.rpm	msp430tools-gcc-3.2.3-20050607.i386.rpm
libc	msp430tools-libc-20050308cvs-20050608.cygwin.i386.rpm	msp430tools-libc-20050308cvs-20050608.i386.rpm
jtag	Not yet available	msp430tools-jtag-lib-20031101cvs-20050610.i386.rpm
gdb	Not yet available	msp430tools-gdb-6.0-20050609.i386.rpm

Step 4: Install TinyOS toolchain

The TinyOS-specific tools are the NesC compiler and a set of tools developed in the tinyos-2.x/tools source code repository. They are also installed using rpms. If you using the Cygwin version recommended in these install instructions, you should install the "Recommended" Windows/Cygwin nesC RPM. If you get strange errors when you try to compile TinyOS programs, such as the error message "the procedure entry point basename could not be located in the dynamic link library cygwin1.dll", this is likely due to a Cygwin version incompatibility: try the "Other" Windows/Cygwin RPM (1.2.7a). If you are using Cygwin and installing the nesC RPM causes an error that the RPM was built for Cygwin, add the `--ignoreos` option. Finally, there are two Linux versions of tinyos-tools, depending on whether you have a 32-bit or 64-bit machine. The first is the i386 RPM and the second is the i686 RPM. If you have a 64-bit Java VM, it is important that you install the i686 RPM or otherwise the Java support may not work properly.

TinyOS-specific Tools

Tool	Recommended Windows/ Cygwin	Other Windows/Cygwin	Linux	Command
NesC	nesc-1.2.8a-1.cygwin.i386.rpm	nesc-1.2.8b-1.cygwin.i386.rpm	nesc-1.2.8a-1.i386.rpm	`rpm -Uvh` `rpm -Uvh --ignoreo`

				`s` (if Cygwin complains)
Tool	**Windows/Cygwin**	**32-bit Linux**	**64-bit Linux**	**Command**
tinyos-tools	**tinyos-tools-1.2.4-2.cygwin.i386.rpm**	**tinyos-tools-1.2.4-3.i386.rpm**	**tinyos-tools-1.2.4-3.i686.rpm**	`rpm -ivh --force` (1.x tree) `rpm -Uvh` (no 1.x tree)

Step 5: Install the TinyOS 2.x source tree

Now that the tools are installed, you need only install the tinyos 2.x source tree and then set your environment variables. Install the appropriate version of the following (Window or Linux) with the rpm command 'rpm -ivh *rpm*'. As with the previous rpms, if you get an error claiming that the rpm was build for an NT computer and you're on a windows NT computer, bypass the erroneous error by using 'rpm -ivh --ignoreos *rpmname*'.

- Install tinyos-2.x

  TinyOS 2.x

	Windows/Cygwin	**Linux**
TinyOS	**tinyos-2.0.2-2.cygwin.noarch.rpm**	**tinyos-2.0.2-2.noarch.rpm**

- Configure your environment

  Ideally, you'll put these environment variables in a shell script that will run when your shell starts, but you needn't put such a script under /etc/profile.d.

  The example settings below assume that the tinyos-2.x installation is in /opt/tinyos-2.x. Change the settings to be correct for where you've put your tinyos-2.x tree. Note that the windows CLASSPATH must be a windows-style path, not a cygwin path. You can generate a windows style path from a cygwin-style path using 'cygpath -w'. For example:

  ```
  export CLASSPATH=`cygpath -w
  $TOSROOT/support/sdk/java/tinyos.jar`
  export CLASSPATH="$CLASSPATH;."
  ```

  TinyOS 2.x

Environme nt	**Windows**	**Linux**

Variable		
TOSROOT	/opt/tinyos-2.x	same as in Cygwin
TOSDIR	$TOSROOT/tos	same as in Cygwin
CLASSPATH	C:\tinyos\cygwin\opt\tinyos-2.x\support\sdk\java\tinyos.jar;.	$TOSROOT/support/sdk/java/tinyos.jar:.
MAKERULES	$TOSROOT/support/make/Makerules	same as in Cygwin
PATH†	/opt/msp430/bin:$PATH	same as in Cygwin

†Only necessary if you're using the MSP430 platform/tools.

In addition to the above environment variables, do the following on Linux machines:

1. Change the ownership on your /opt/tinyos-2.x files: `chown -R <your uid> /opt/tinyos-2.x`
2. Change the permissions on any serial (/dev/ttyS<N>), usb (/dev/tts/usb<N>, /dev/ttyUSB<N>), or parallel (/dev/parport) devices you are going to use: `chmod 666 /dev/<devicename>`

- Finally, if you have installed TinyOS 2.0.1, there is a bug in TOSSIM (which will be fixed in 2.0.2). The bug is in file `tos/chips/atm128/sim/atm128_sim.h`. Change these lines 22 and 23 from:
  - `#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr) + 0x20)`
  - `#define _SFR_IO16(io_addr) _MMIO_WORD((io_addr) + 0x20)`

  to

```
#define _SFR_IO8(io_addr) _MMIO_BYTE((io_addr))
#define _SFR_IO16(io_addr) _MMIO_WORD((io_addr))
```

  If you do not do this, then timers will not work correctly.

Step 6: Installing Graphviz

Go to **download page** of the Graphviz project and download the appropriate RPM. You only need the basic graphviz RPM (`graphviz-`); you don't need all of the add-ons, such as -devel, -doc, -perl, etc. If you're not sure what version of Linux you're running,

```
uname -a
```

might give you some useful information. Install the rpm with `rpm -i` *rpm-name*. In the case of Windows, there is a simple install program, so you don't need to deal with RPMs.

Lesson 1: Getting Started with TinyOS and nesC

Last updated April 8 2007

Introduction

This lesson shows you how to compile a TinyOS program and install it on a mote, introduces the basic concepts and syntax of the TinyOS component model, and shows you how to generate and navigate TinyOS's source code documentation.

Compiling and Installing

As a first exercise, you'll compile and install a very simple TinyOS application called Blink. If you don't have mote hardware, you can compile it for TOSSIM, the TinyOS simulator.

You compile TinyOS applications with the program `make`. TinyOS uses a powerful and extensible make system that allows you to easily add new platforms and compilation options. The makefile system definitions are located in tinyos-2.x/support/make.

The first step is to check that your environment is set up correctly. Run the `tos-check-env` command:

```
$ tos-check-env
```

This script checks pretty much everything that the TinyOS environment needs. Most of the warnings should be somewhat self-explanatory, if you are at all accustomed to a UNIX environment. If you are having trouble with warnings, your best bet is to join and ask questions on the **tinyos-help** email list. It's almost always the case that if you've run into a problem, someone else has too. Searching the **help archives** can therefore be useful.

If your system says some command is not available, then chances are you need to install the TinyOS tools (tos-*) RPM. Please refer to your installation instructions. If you have installed the RPM, then look in /usr/bin and /usr/local/bin for tos-check-env. If you have downloaded from CVS rather than used RPMs, then you need to compile and build the tools. Go to `tinyos-2.x/tools/tinyos` and type:

```
$ configure
$ make
$ make install
```

On Linux systems, you will either need superuser abilities or access to `sudo` for the last command.

The second thing to check is that you have the TinyOS build system enabled. This involves the MAKERULES environment variable. In a shell, type

```
printenv MAKERULES
```

You should see `/opt/tinyos-2.x/support/make/Makerules`. If your TinyOS tree is installed somewhere besides the standard place, you might not see `/opt`, but rather a different initial path. If MAKERULES is not set (printenv prints nothing), you need to set it. Depending on your shell, this involves using either `export` (bash) or `setenv` (csh, tcsh). If you don't know about shell environment variables, this **tutorial** should help.

The make command to compile a TinyOS application is `make` *[platform]*, executed from the application's directory. To compile Blink, go the `apps/Blink` directory and depending on which hardware you have, type `make micaz`, `make mica2`, `make telosb`, or, for simulation, type `make micaz sim`.

You should see output such as this:

```
dark /root/src/tinyos-2.x/apps/Blink -4-> make telosb
mkdir -p build/telosb
    compiling BlinkAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -mdisable-hwmul -
Wall -Wshadow
-DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -target=telosb -
fnesc-cfile=build/telosb/app.c
-board=   BlinkAppC.nc -lm
    compiled BlinkAppC to build/telosb/main.exe
            2782 bytes in ROM
              61 bytes in RAM
msp430-objcopy --output-target=ihex
build/telosb/main.exe build/telosb/main.ihex
    writing TOS image
```

If you compile for TOSSIM, you will see a much longer message, as building TOSSIM requires several steps, such as building shared libraries and scripting support.

If you are running Cygwin and see the error message "The procedure entry point basename could not be located in the dynamic link library cygwin1.dll" then you need to install the other Cygwin nesC RPM (step 4 of the **install instructions** or step 2 of the **upgrade instructions**)

Making sure you're invoking the right version of the nesC compiler

If you see an error message along the lines of this:

```
BlinkAppC.nc:46: syntax error before `new'
make: *** [exe0] Error 1
```

Then you are invoking an older version of the nesc compiler. Check by typing `ncc --version`. You should see:

```
ncc: 1.2.1
nescc: 1.2.6
```

Followed by some information on what version of the C compiler is being used. If you see different versions than those above, your compilation problems are most probably due to the fact that make is invoking the wrong version. This can easily happen if you are upgrading from TinyOS 1.x. You might have passed the tos-check-env because you have the right compiler on your system, but for some reason make is invoking the wrong one.

`ncc` is a script that invokes the full compiler, `nescc`. It lives in `tinyos-2.x/tools/tinyos/ncc`. If you've installed from an RPM, then the RPM put the new version of `ncc` in `/usr/bin`. You can see which version make is invoking by typing `which ncc`:

```
$ which ncc
/usr/local/bin/ncc
$ /usr/local/bin/ncc --version
Unknown target mica
Known targets for TinyOS directory /opt/tinyos-2.x/tos
and the specified include directories are:
none.
```

In this case, the version of ncc is so old that it doesn't even respond to the `--version` flag. In contrast,

```
$ /usr/bin/ncc --version
ncc: 1.2.1
nescc: 1.2.5
```

The best solution to this problem is to move the old ncc to a different name (keep in around in case you need to go back to your old setup):

```
$ mv /usr/local/bin/ncc /usr/local/bin/ncc.old
```

```
$ which ncc
/usr/bin/ncc
```

You can apply the same process for `nescc`:

```
$ nescc --version
ncc: 1.1.2
$ which nescc
/usr/local/bin/nescc
$ /usr/bin/nescc --version
nescc: 1.2.5
$ mv /usr/local/bin/nescc /usr/local/bin/nescc.old
$ which nescc
/usr/bin/nescc
```

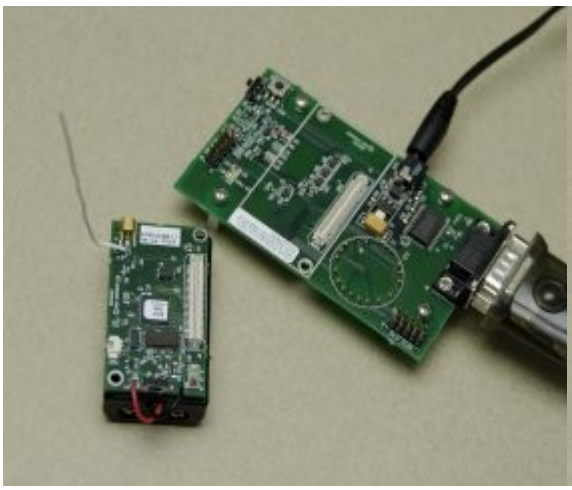
Now that we've compiled the application it's time to program the mote and run it. The next step depends on what family of mote you are programming.

- **Installing on a mica-family mote**
- **Installing on a telos-family mote**
- **Installing on a tinynode mote**
- **Installing on an eyesIFX-family mote**
- **Installing on an IntelMote2**

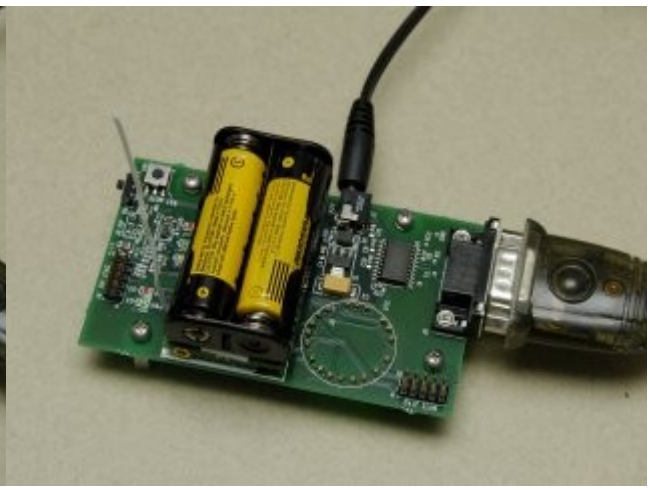
Installing on a mica-family mote (micaz, mica2, mica2dot)

This example uses a Mica2 mote and the serial-based programming board (`mib510`). Instructions on how to use other programming boards are **here**. To download your program onto the mote, place the mote board (or mote and sensor stack) into the bay on the programming board, as shown below. You can either supply a 3 or 5 volt supply to the connector on the programming board or power the node directly. The green LED (labeled PWR) on the programming board will be on when power is supplied. If you are using batteries to power the mote, be sure the mote is switched on (the power switch should be towards the connector). The ON/OFF switch on the mib510 board should normally be left in the OFF position. Only switch it to ON if you have problems programming the mote and when you are done programming, switch it back to OFF (when the switch is ON the mote cannot send data to the PC).

Plug the 9-pin connector into the serial port of a computer configured with the TinyOS tools, using a pass-through (not null-modem!) DB9 serial cable. If your computer does not have a serial port, you can easily obtain DB9-serial-to-USB cables.



Mica2 mote next to the programming board



Mica2 mote connected to the programming board

Type:

```
make mica2 reinstall mib510,serialport
```

where *serialport* is the serial port device name. Under Windows, if your serial port is COM*n*:, you must use /dev/ttyS*n-1* as the device name. On Linux, the device name is typically /dev/ttyS*n* for a regular serial port and /dev/ttyUSB*n* or /dev/usb/tts/*n* for a USB-serial cable (the name depends on the Linux distribution). Additionally, on Linux, you will typically need to make this serial port world writeable. As superuser, execute the following command:

```
chmod 666 serialport
```

If the installation is successful you should see something like the following (if you don't, try repeating the make command):

```
cp build/mica2/main.srec build/mica2/main.srec.out
    installing mica2 binary using mib510
uisp -dprog=mib510 -dserial=/dev/ttyUSB1 --
wr_fuse_h=0xd9 -dpart=ATmega128
     --wr_fuse_e=ff  --erase --upload
if=build/mica2/main.srec.out
Firmware Version: 2.1
Atmel AVR ATmega128 is found.
Uploading: flash

Fuse High Byte set to 0xd9

Fuse Extended Byte set to 0xff
rm -f build/mica2/main.exe.out
build/mica2/main.srec.out
```

Installing on telos-family mote (telosa, telosb)

Telos motes are USB devices, and can be plugged into any USB port:

Telos mote

Telos mote plugged into a USB port

Because Telos motes are USB devices, they register with your OS when you plug them in. Typing `motelist` will display which nodes are currently plugged in:

```
$ motelist
Reference  CommPort   Description
---------- ----------
----------------------------------------
UCC89MXV   COM4       Telos (Rev B 2004-09-27)
```

motelist tells you which ports have motes attached. Under Windows, it displays the mote's COM port (in this case 4), under Linux it displays just the USB serial port number (e.g., 2). Confusingly, the Windows version of the code installer (`tos-bsl`) takes the COM port number - 1 as it's argument (in this case 3); under Linux it takes the USB device name (e.g., /dev/ttyUSB2 or /dev/tts/usb/2 if motelist reports that the mote is device 2). On Linux, as with the mica programmers, you will typically need to make the USB serial port world writeable. As superuser, execute the following command:

```
chmod 666 usb-device-name
```

Now you can install the application using one of:

```
  make telosb reinstall bsl,3            # Windows
example
  make telosb reinstall bsl,/dev/ttyUSB2 # Linux
example
```

This would compile an image suitable for the telosb platform and install it with a mote ID of 2 on a mote connected to COM4 on Windows or /dev/ttyUSB2 on Linux. If you have a single mote installed, you can skip the bsl and device name/number arguments. Again, see the Getting Started Guide for your chosen platform for the exact make parameters.

You should see something like this scroll by:

```
    installing telosb binary using bsl
tos-bsl --telosb -c 16 -r -e -I -p
build/telosb/main.ihex.out
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID:
f16c)
Changing baudrate to 38400 ...
Program ...
2782 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out-2
build/telosb/main.ihex.out
```

Installing on a TinyNode mote

There are different ways to program a TinyNode mote depending on how it is connected to your computer. The most common case is to connect it to a serial port using either the standard extension board (SEB) or the MamaBoard. *(The other possible methods are to use a Mamaboard with a Digi Connect ethernet adaptor and program a node over the network, or to use a JTAG adaptor. These are not covered in this tutorial; please refer to the Tinynode documentation for further details.)*

To install an application on a TinyNode mote using the serial port, enter the following command, taking care to replace `/dev/ttyXXX` with the file device corresponding to the serial port that the tinynode is plugged into.

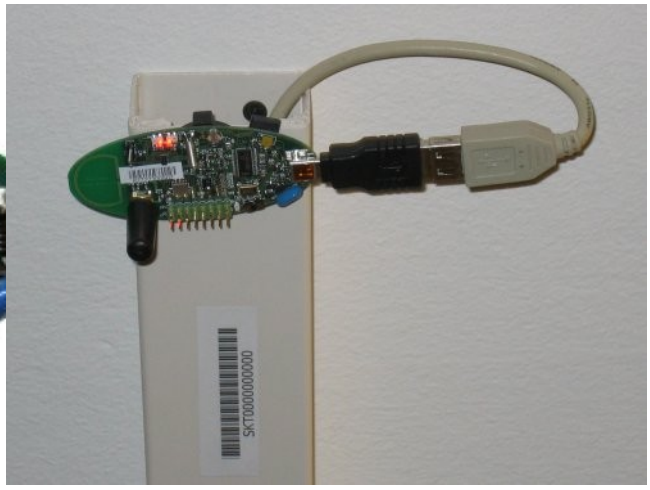
```
  make tinynode reinstall bsl,/dev/XXX
```

As with the telos and eyesIFX platforms, this command will reprogram your mote using the `tos-bsl` utility, and you will see similar output on your screen as given above for telos.

Installing on an eyesIFX-family mote

The eyesIFX motes have a mini-B USB connector, allowing easy programming and data exchange over the USB. The on-board serial-to-USB chip exports two separate serial devices: a lower-numbered one used exclusively for serial data communication, and a higher-numbered one used for programming of the microcontroller.

eyesIFXv2 mote



eyesIFXv2 mote attached to a USB cable

The actual programming is performed by the *msp430-bsl* script, conveniently invoked using the same *make* rules as for the telos motes. In the most basic form:

```
make eyesIFX install bsl
```

the install script defaults to programming using the /dev/ttyUSB1 device on Linux and COM1 on Windows, giving output similar to this:

```
    installing eyesIFXv2 binary using bsl
msp430-bsl --invert-test --invert-reset --f1x -c /dev/
ttyUSB1 -r -e -I -p build/eyesIFXv2/main.ihex.out
MSP430 Bootstrap Loader Version: 2.0
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device ID:
f16c)
Program ...
2720 bytes programmed.
Reset device ...
rm -f build/eyesIFXv2/main.exe.out
build/eyesIFXv2/main.ihex.out
```

The programming device can also be explicitly set as a parameter of the *bsl* command using shorthand or full notation:

```
  make eyesIFX install bsl,USB3
  make eyesIFX install bsl,/dev/ttyUSB3
```

The eyesIFX motes can be programmed over the provided JTAG interface with the help of the msp430-jtag script:

```
  make eyesIFX install jtag
```

producing output as in the following:

```
    installing eyesIFXv2 binary using the parallel
port jtag adapter
msp430-jtag  -Iepr build/eyesIFXv2/main.ihex.out
MSP430 parallel JTAG programmer Version: 2.0
Mass Erase...
Program...
2720 bytes programmed.
Reset device...
Reset and release device...
```

Installing on an IntelMote2

Installation options

You can now test the program by unplugging the mote from the programming board and turning on the power switch (if it's not already on). With any luck the three LEDs should be displaying a counter incrementing at 4Hz.

The `reinstall` command told the make system to install the currently compiled binary: it skips the compilation process. Type `make clean` to clean up all of the compiled binary files, then type, e.g., `make telosb install` This will recompile Blink and install it on one action.

Networking almost always requires motes to have unique identifiers. When you compile a TinyOS application, it has a default unique identifier of 1. To give a node a different identifier, you can specify it at installation. For example, if you type `make telosb install.5` or `make telosb reinstall.5`, you will install the application on a node and give it 5 as its identifier.

For more information on the build system, please see **Lesson 13**.

Components and Interfaces

Now that you've installed Blink, let's look at how it works. Blink, like all TinyOS code, is written in nesC, which is C with some additional language features for components and concurrency.

A nesC application consists of one or more *components* assembled, or *wired*, to form an application executable. Components define two scopes: one for their specification which contains the names of their *interfaces*, and a second scope for their implementation. A component *provides* and *uses* interfaces. The provided interfaces are intended to represent the functionality that the component provides to its user in its specification; the used interfaces represent the functionality the component needs to perform its job in its implementation.

Interfaces are bidirectional: they specify a set of *commands*, which are functions to be implemented by the interface's provider, and a set of *events*, which are functions to be implemented by the interface's user. For a component to call the commands in an interface, it must implement the events of that interface. A single component may use or provide multiple interfaces and multiple instances of the same interface.

The set of interfaces which a component provides together with the set of interfaces that a component uses is considered that component's *signature*.

Configurations and Modules

There are two types of components in nesC: *modules* and *configurations*. Modules provide the implementations of one or more interfaces. Configurations are used to assemble other components together, connecting interfaces used by components to interfaces provided by others. Every nesC application is described by a top-level configuration that wires together the components inside.

Blink: An Example Application

Let's look at a concrete example: **Blink** in the TinyOS tree. As you saw, this application displays a counter on the three mote LEDs. In actuality, it simply causes the LED0 to to turn on and off at .25Hz, LED1 to turn on and off at .5Hz, and LED2 to turn on and off at 1Hz. The effect is as if the three LEDs were displaying a binary count of one to seven every two seconds.

Blink is composed of two **components**: a **module**, called "BlinkC.nc", and a **configuration**, called "BlinkAppC.nc". Remember that all applications require a top-level configuration file, which is typically named after the application itself. In this case BlinkAppC.nc is the configuration for the Blink application and the source file that the nesC compiler uses to generate an executable file. BlinkC.nc, on the other hand, actually provides the *implementation* of the

Blink application. As you might guess, BlinkAppC.nc is used to wire the BlinkC.nc module to other components that the Blink application requires.

The reason for the distinction between modules and configurations is to allow a system designer to build applications out of existing implementations. For example, a designer could provide a configuration that simply wires together one or more modules, none of which she actually designed. Likewise, another developer can provide a new set of library modules that can be used in a range of applications.

Sometimes (as is the case with BlinkAppC and BlinkC) you will have a configuration and a module that go together. When this is the case, the convention used in the TinyOS source tree is:

File Name	File Type
Foo.nc	Interface
Foo.h	Header File
FooC.nc	Public Module
FooP.nc	Private Module

While you could name an application's implementation module and associated top-level configuration anything, to keep things simple we suggest that you adopt this convention in your own code. There are several other conventions used in TinyOS; **TEP 3** specifies the coding standards and best current practices.

The BlinkAppC.nc Configuration

The nesC compiler compiles a nesC application when given the file containing the top-level configuration. Typical TinyOS applications come with a standard Makefile that allows platform selection and invokes ncc with appropriate options on the application's top-level configuration.

Let's look at BlinkAppC.nc, the configuration for this application first:

apps/Blink/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;
}
```

The first thing to notice is the key word configuration, which indicates that this is a configuration file. The first two lines,

apps/Blink/BlinkAppC.nc:

```
configuration BlinkAppC {
}
```

simply state that this is a configuration called BlinkAppC. Within the empty braces here it is possible to specify uses and provides clauses, as with a module. This is important to keep in mind: a configuration can use and provide interfaces. Said another way, not all configurations are top-level applications.

The actual configuration is implemented within the pair of curly brackets following the key word implementation . The components lines specify the set of components that this configuration references. In this case those components are Main, BlinkC, LedsC, and three instances of a timer component called TimerMilliC which will be referenced as Timer0, Timer1, and Timer2(1). This is accomplished via the *as* keyword which is simply an alias(2).

As we continue reviewing the BlinkAppC application, keep in mind that the BlinkAppC component is not the same as the BlinkC component. Rather, the BlinkAppC component is composed of the BlinkC component along with MainC, LedsC and the three timers.

The remainder of the BlinkAppC configuration consists of connecting interfaces used by components to interfaces provided by others. The MainC.Boot and MainC.SoftwareInit interfaces are part of TinyOS's boot sequence and will be covered in detail in Lesson 3. Suffice it to say that these wirings enable the LEDs and Timers to be initialized.

The last four lines wire interfaces that the BlinkC component *uses* to interfaces that the TimerMilliC and LedsC components *provide*. To fully understand the semantics of these wirings, it is helpful to look at the BlinkC module's definition and implementation.

The BlinkC.nc Module

apps/Blink/BlinkC.nc:

```
module BlinkC {
  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
{
  // implementation code omitted
}
```

The first part of the module code states that this is a module called BlinkC and declares the interfaces it provides and uses. The BlinkC module **uses** three instances of the interface Timer<TMilli> using the names Timer0, Timer1 and Timer2 (the <TMilli> syntax simply supplies the generic Timer interface with the required timer precision). Lastly, the BlinkC module also uses the Leds and Boot interfaces. This means that BlinkC may call any command declared in the interfaces it uses and must also implement any events declared in those interfaces.

After reviewing the interfaces used by the BlinkC component, the semantics of the last four lines in BlinkAppC.nc should become clearer. The line BlinkC.Timer0 -> Timer0 wires the three Timer<TMilli> interface used by BlinkC to the Timer<TMilli> interface provided the three TimerMilliC component. The BlinkC.Leds -> LedsC line wires the Leds interface used by the BlinkC component to the Leds interface provided by the LedsC component.

nesC uses arrows to bind interfaces to one another. The right arrow (A->B) as "A wires to B". The left side of the arrow (A) is a user of the interface, while the right side of the arrow (B) is the provider. A full wiring is A.a->B.b, which means "interface a of component A wires to interface b of component B." Naming the interface is important when a component uses or provides multiple instances of the same interface. For example, BlinkC uses three instances of Timer: Timer0, Timer1 and Timer2. When a component only has one instance of an interface, you can elide the interface name. For example, returning to BlinkAppC:

apps/Blink/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;
}
```

The interface name Leds does not have to be included in LedsC:

```
  BlinkC.Leds -> LedsC; // Same as BlinkC.Leds ->
LedsC.Leds
```

Because BlinkC only uses one instance of the Leds interface, this line would also work:

```
  BlinkC -> LedsC.Leds; // Same as BlinkC.Leds ->
LedsC.Leds
```

As the TimerMilliC components each provide a single instance of Timer, it does not have to be included in the wirings:

```
  BlinkC.Timer0 -> Timer0;
  BlinkC.Timer1 -> Timer1;
  BlinkC.Timer2 -> Timer2;
```

However, as BlinkC has three instances of Timer, eliding the name on the user side would be a compile-time error, as the compiler would not know which instance of Timer was being wired:

```
  BlinkC -> Timer0.Timer;  // Compile error!
```

The direction of a wiring arrow is always from a user to a provider. If the provider is on the left side, you can also use a left arrow:

```
  Timer0 <- BlinkC.Timer0; // Same as BlinkC.Timer0 ->
Timer0;
```

For ease of reading, however, most wirings are left-to-right.

Visualizing a Component Graph

Carefully engineered TinyOS systems often have many layers of configurations, each of which refines the abstraction in simple way, building something robust with very little executable code. Getting to the modules underneath -- or just navigating the layers -- with a text editor can be laborious. To aid in this process, TinyOS and nesC have a documentation feature called nesdoc, which generates documentation automatically from source code. In addition to comments, nesdoc displays the structure and composition of configurations.

To generate documentation for an application, type

```
  make platform docs
```

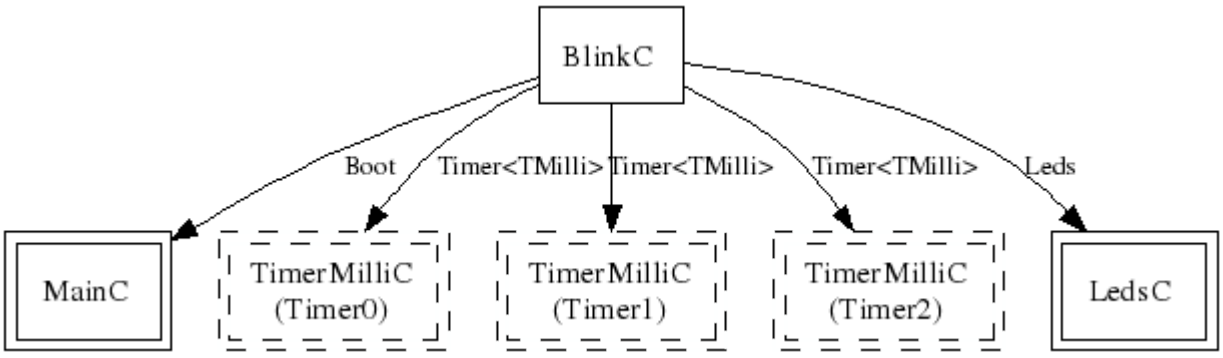
You should see a long list of interfaces and components stream by. If you see the error message

```
sh: dot: command not found
```

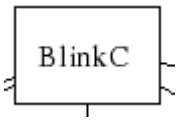
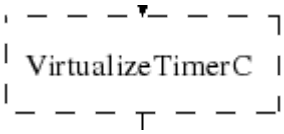
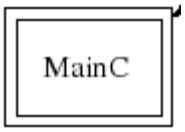
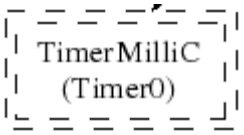
then you need to **install graphviz**, which is the program that draws the component graphs.

Once you've generated the documentation, go to tinyos-2.x/doc/nesdoc. You should see a directory for your platform: open its index.html, and you'll see a list of the components and interfaces for which you've generated documentation. For example, if you generated documentation for Blink on the telosb platform, you'll see documentation for interfaces such as Boot, Leds, and Timer, as well as some from the underlying hardware implementations, such as Msp430TimerEvent and HplMsp430GeneralIO.

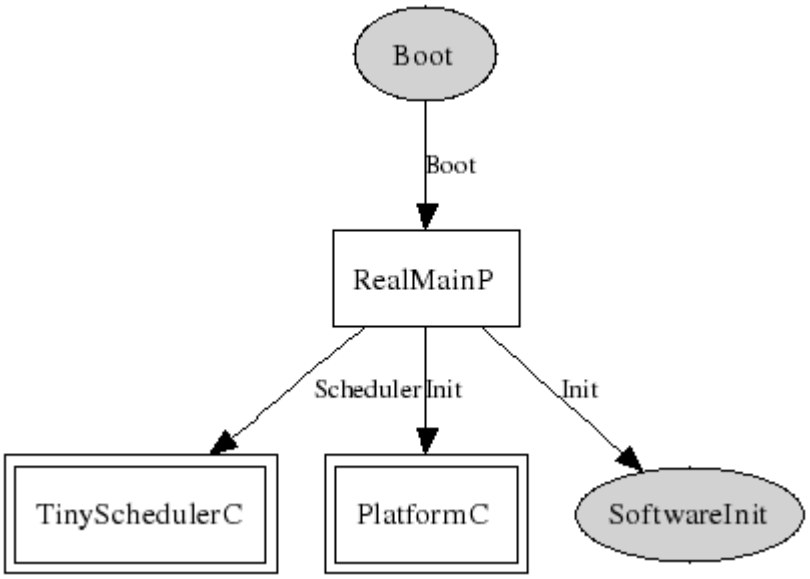
In the navigation panel on the left, components are below interfaces. Click on BlinkAppC, and you should a figure like this:



In nesdoc diagrams, a single box is a module and a double box is a configuration. Dashed border lines denote that a component is a generic:

	Singleton	**Generic**
Module		
Configuration		

Lines denote wirings, and shaded ovals denote interfaces that a component provides or uses. You can click on the components in the graph to examine their internals. Click on MainC, which shows the wirings for the boot sequence:



Shaded ovals denote wireable interfaces. Because MainC provides the Boot interface and uses the Init (as SoftwareInit) interface, it has two shaded ovals. Note the direction of the arrows: because it uses SoftwareInit, the wire goes out from RealMainP to SoftwareInit, while because it provides Boot, the wire goes from Boot into RealMainP. The details of MainC aren't too important here, and we'll be looking at it in greater depth in **lesson 3** (you can also read **TEP 107** for details), but looking at the components you can get a sense of what it does: it controls the scheduler, initializes the hardware platform, and initializes software components.

Conclusion

This lesson has introduced the concepts of the TinyOS component model: configurations, modules, interfaces and wiring. It showed how applications are built by wiring components together. The next lesson continues with Blink, looking more closely at modules, including the TinyOS concurrency model and executable code.

Related Documentation

- mica mote Getting Started Guide at **Crossbow**
- telos mote Getting Started Guide for **Moteiv**
- **nesc at sourceforge**
- **nesC reference manual**
- **TinyOS Programming Guide**
- **TEP 3: TinyOS Coding Conventions**
- **TEP 102: Timers**
- **TEP 106: Schedulers and Tasks**
- **TEP 106: TinyOS 2.x Boot Sequence**

(1) The TimerMilliC component is a *generic component* which means that, unlike non-generic components, it can be instantiated more than once. Generic components can take types and constants as arguments, though in this case TimerMilliC takes none. There are also *generic interafces*, which take type arguments only. The Timer interface provided by TimerMilliC is a generic interface; its type argument defines the timer's required precision (this prevents programmer from wiring, e.g., microsecond timer users to millisecond timer providers). A full explanation of generic components is outside this document's scope, but you can read about them in the nesc generic component documentation.

(2) **Programming Hint 10:** Use the *as* keyword liberally. From ***TinyOS Programming***

Lesson 2: Modules and the TinyOS Execution Model

Last updated June 27 2006

This lesson introduces nesC modules, commands, events, and tasks. It explains the TinyOS execution model.

Modules and State

Compiling TinyOS applications produces a single binary image that assumes it has complete control of the hardware. Therefore, a mote only runs one TinyOS image at a time. An image consists of the components needed for a single application. As most more platforms do not have hardware-based memory protection, there is no seperation between a "user" address space and a "system" address space; there is only one address space that all components share. This is why many TinyOS components try to keep their state private and avoid passing pointers: since there is no hardware protection, the best way to keep memory uncorrupted is to share it as little as possible.

Recall from lesson 1 that the set of interfaces a component uses and provides define its signature. Both kinds of components -- configurations and modules -- provide and use interfaces. The difference between the two lies in their implementation: configurations are implemented in terms of other components, which they wire, while modules are executable code. After unwrapping all of the layers of abstraction that configurations introduce, modules always lie within. Module implementations are for the most part written in C, with some extra constructions for nesC abstractions.

Modules can declare state variables. Any state a component declares is private: no other component can name it or directly access it. The only way two components can directly interact is through interfaces. Let's revisit the Blink application. Here is the Blink module BlinkC's implementation in its entirety:

apps/Blink/BlinkC.nc:

```
module BlinkC {
  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()
  {
    call Leds.led0Toggle();
  }

  event void Timer1.fired()
  {
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
}
```

BlinkC does not allocate any state. Let's change it so that its logic is a little different: rather than blink the LEDs from three different timers, we'll blink them with a single timer and keep some state to know which ones to toggle. Make a copy of the Blink application, BlinkSingle, and go into its directory.

```
$ cd tinyos-2.x/apps
```

```
$ cp -R Blink BlinkSingle
$ cd BlinkSingle
```

Open the BlinkC module in an editor. The first step is to comment out the LED toggles in Timer1 and Timer2:

```
  event void Timer1.fired()
  {
    // call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    // call Leds.led2Toggle();
  }
```

The next step is to add some state to BlinkC, a single byte. Just like in C, variables and functions must be declared before they are used, so put it at the beginning of the implementation:

```
implementation
{

  uint8_t counter = 0;

  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }
```

Rather than the standard C names of int, long, or char, TinyOS code uses more explicit types, which declare their size. In reality, these map to the basic C types, but do so differently for different platforms. TinyOS code avoids using int, for example, because it is platform-specific. For example, on mica and Telos motes, int is 16 bits, while on the IntelMote2, it is 32 bits. Additionally, TinyOS code often uses unsigned values heavily, as wrap-arounds to negative numbers can often lead to very unintended consequences. The commonly used types are:

	8 bits	16 bits	32 bits	64 bits

signed	int8_t	int16_t	int32_t	int64_t
unsigned	uint8_t	uint16_t	uint32_t	uint64_t

There is also a bool type. You can use the standard C types, but doing so might raise cross-platform issues. Also, uint32_t is often easier to write than unsigned long. Most platforms support floating point numbers (float almost always, double sometimes), although their arithmetic is in software rather than hardware.

Returning to our modified BlinkC, we've allocated a single unsigned byte, counter. When the mote boots, the counter will be initialized to zero. The next step is to make it that when Timer0 fires, it increments counter and displays the result:

```
event void Timer0.fired()
{
  counter++;
  if (counter & 0x1) {
    call Leds.led0On();
  }
  else {
    call Leds.led0Off();
  }
  if (counter & 0x2) {
    call Leds.led2On();
  }
  else {
    call Leds.led2Off();
  }
  if (counter & 0x4) {
    call Leds.led2On();
  }
  else {
    call Leds.led2Off();
  }
}
```

Another, more succinct way to do it is to use the set command:

```
event void Timer0.fired()
{
  counter++;
  call Leds.set(counter);
```

```
  }
```

Compile your program and install it on a mote. You'll see that it behaves just as before, except that now the LEDs are being driven by a single, rather than three, timers.

As only one timer is being used, this means that you don't need Timer1 and Timer2: they waste CPU resources and memory. Open BlinkC again and remove them from its signature and implementation. You should have something that looks like this:

```
module BlinkC {
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
  users interface Boot;
}
implementation
{
  uint8_t counter = 0;

  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
  }

  event void Timer0.fired()
  {
    counter++;
    call Leds.set(counter);
  }

}
```

Try to compile the application: nesC will throw an error, because the configuration BlinkAppC is wiring to interfaces on BlinkC that no longer exist (Timer1 and Timer2):

```
dark /root/src/tinyos-2.x/apps/BlinkSingle -5-> make
micaz
mkdir -p build/micaz
    compiling BlinkAppC to a micaz binary
ncc -o build/micaz/main.exe -Os -finline-limit=100000
-Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -
target=micaz -fnesc-cfile=build/micaz/app.c -
board=micasb  -fnesc-dump=wiring -fnesc-
```

```
dump='interfaces(!abstract())' -fnesc-
dump='referenced(interfacedefs, components)' -fnesc-
dumpfile=build/micaz/wiring-check.xml BlinkAppC.nc -lm
In component `BlinkAppC':
BlinkAppC.nc:54: cannot find `Timer1'
BlinkAppC.nc:55: cannot find `Timer2'
make: *** [exe0] Error 1
```

Open BlinkAppC and remove the two Timers and their wirings. Compile the application:

```
mkdir -p build/micaz
    compiling BlinkAppC to a micaz binary
ncc -o build/micaz/main.exe -Os -finline-limit=100000
-Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -
target=micaz -fnesc-cfile=build/micaz/app.c -
board=micasb  -fnesc-dump=wiring -fnesc-
dump='interfaces(!abstract())' -fnesc-
dump='referenced(interfacedefs, components)' -fnesc-
dumpfile=build/micaz/wiring-check.xml BlinkAppC.nc -lm
    compiled BlinkAppC to build/micaz/main.exe
            2428 bytes in ROM
              39 bytes in RAM
avr-objcopy --output-target=srec build/micaz/main.exe
build/micaz/main.srec
avr-objcopy --output-target=ihex build/micaz/main.exe
build/micaz/main.ihex
    writing TOS image
```

If you compare the ROM and RAM sizes with the unmodified Blink application, you should see that they are a bit smaller: TinyOS is only allocating state for a single timer, and there is event code for only one timer.

Interfaces, Commands, and Events

Go back to tinyos-2.x/apps/Blink. In lesson 1 we learned that if a component uses an interface, it can call the interface's commands and must implement handlers for its events. We also saw that the BlinkC component uses the Timer, Leds, and Boot interfaces. Let's take a look at those interfaces:

tos/interfaces/Boot.nc:

```
interface Boot {
```

```
  event void booted();
}
```

tos/interfaces/Leds.nc:

```
interface Leds {

  /**
   * Turn LED n on, off, or toggle its present state.
   */
  async command void led0On();
  async command void led0Off();
  async command void led0Toggle();

  async command void led1On();
  async command void led1Off();
  async command void led1Toggle();

  async command void led2On();
  async command void led2Off();
  async command void led2Toggle();

  /**
   * Get/Set the current LED settings as a bitmask.
Each bit corresponds to
   * whether an LED is on; bit 0 is LED 0, bit 1 is
LED 1, etc.
   */
  async command uint8_t get();
  async command void set(uint8_t val);

}
```

tos/interfaces/Timer.nc:

```
interface Timer
{
  // basic interface
  command void startPeriodic( uint32_t dt );
  command void startOneShot( uint32_t dt );
  command void stop();
  event void fired();

  // extended interface omitted (all commands)
}
```

Looking over the interfaces for `Boot`, `Leds`, and `Timer`, we can see that since `BlinkC` uses those interfaces it must implement handlers for the `Boot.booted()` event, and the

`Timer.fired()` event. The `Leds` interface signature does not include any events, so `BlinkC` need not implement any in order to call the Leds commands. Here, again, is `BlinkC`'s implementation of `Boot.booted()`:

apps/Blink/BlinkC.nc:

```
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }
```

`BlinkC` uses 3 instances of the TimerMilliC component, wired to the interfaces `Timer0`, `Timer1`, and `Timer2`. The `Boot.booted()` event handler starts each instance. The parameter to `startPeriodic()` specifies the period in milliseconds after which the timer will fire (it's millseconds because of the `<TMilli>` in the interface). Because the timer is started using the `startPeriodic()` command, the timer will be reset after firing such that the `fired()` event is triggered every n milliseconds.

Invoking an interface command requires the `call` keyword, and invoking an interface event requires the `signal` keyword. BlinkC does not provide any interfaces, so its code does not have any signal statements: in a later lesson, we'll look at the boot sequence, which signals the Boot.booted() event.

Next, look at the implementation of the `Timer.fired()`:

apps/Blink/BlinkC.nc:

```
  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }

  event void Timer1.fired()
  {
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
}
```

Because it uses three instances of the Timer interface, `BlinkC` must implement three instances of `Timer.fired()` event. When implementing or invoking an interface function, the function name is always *interface.function*. As BlinkC's three Timer instances are named `Timer0`, `Timer1`, and `Timer2`, it implements the three functions `Timer0.fired`, `Timer1.fired`, and `Timer2.fired`.

TinyOS Execution Model: Tasks

All of the code we've looked at so far is *synchronous*. It runs in a single execution context and does not have any kind of pre-emption. That is, when synchronous (sync) code starts running, it does not relinquish the CPU to other sync code until it completes. This simple mechanism allows the TinyOS scheduler to minimize its RAM consumption and keeps sync code very simple. However, it means that if one piece of sync code runs for a long time, it prevents other sync code from running, which can adversely affect system responsiveness. For example, a long-running piece of code can increase the time it takes for a mote to respond to a packet.

So far, all of the examples we've looked at have been direct function calls. System components, such as the boot sequence or timers, signal events to a component, which takes some action (perhaps calling a command) and returns. In most cases, this programming approach works well. Because sync code is non-preemptive, however, this approach does not work well for large computations. A component needs to be able to split a large computation into smaller parts, which can be executed one at a time. Also, there are times when a component needs to do something, but it's fine to do it a little later. Giving TinyOS the ability to defer the computation until later can let it deal with everything else that's waiting first.

Tasks enable components to perform general-purpose "background" processing in an application. A task is a function which a component tells TinyOS to run later, rather than now. The closest analogies in traditonal operating systems are **interrupt bottom halves** and **deferred procedure calls**.

Make a copy of the Blink application, and call it BlinkTask:

```
$ cd tinyos-2.x/apps
$ cp -R Blink BlinkTask
$ cd BlinkTask
```

Open `BlinkC.nc`. Currently, the event handler for `Timer0.fired()` is:

```
event void Timer0.fired() {
  dbg("BlinkC", "Timer 0 fired @ %s\n",
sim_time_string());
  call Leds.led0Toggle();
```

```
}
```

Let's change it so that it does a bit of work, enough that we'll be able to see how long it runs. In terms of a mote, the rate at which we can see things (about 24 Hz, or 40 ms) is slow: the micaZ and Telos can send about 20 packets in that time. So this example is really exaggerated, but it's also simple enough that you can observe it with the naked eye. Change the handler to be this:

```
event void Timer0.fired() {
  uint32_t i;
  dbg("BlinkC", "Timer 0 fired @ %s\n",
sim_time_string());
  for (i = 0; i < 400001; i++) {
    call Leds.led0Toggle();
  }
}
```

This will cause the timer to toggle 400,001 times, rather than once. Because the number is odd, it will have the end result of a single toggle, with a bit of flickering in-between. Compile and install the program. You'll see that Led 0 introduces so much latency in the Led 1 and Led 2 toggles that you never see a situation where only one is on. On TelosB motes, this long running task can cause the Timer stack to completely skip events (try setting the count to 200,001 or 100,001).

The problem is that this computation is interfering with the timer's operation. What we'd like to do is tell TinyOS to execute the computation later. We can accomplish this with a **task**.

A task is declared in your implementation module using the syntax

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

where `taskname()` is whatever symbolic name you want to assign to the task. Tasks must return `void` and may not take any arguments. To dispatch a task for (later) execution, use the syntax

```
  post taskname();
```

A component can post a task in a command, an event, or a task. Because they are the root of a call graph, a tasks can safely both call commands and signal events. We will see later that, by convention, commands do not signal events to avoid creating recursive loops across component boundaries (e.g., if command X in component 1 signals event Y in component 2, which itself calls command X in component 1). These loops would be hard for the programmer to detect (as they depend on how the application is wired) and would lead to large stack usage.

Modify BlinkC to perform the loop in a task:

```
task void computeTask() {
```

```
  uint32_t i;
  for (i = 0; i < 400001; i++) {}
}

event void Timer0.fired() {
  call Leds.led0Toggle();
  post computeTask();
}
```

Telos platforms will still struggle, but mica platforms will operate OK.

The post operation places the task on an internal **task queue** which is processed in FIFO order. When a task is executed, it runs to completion before the next task is run. Therefore, and as the above examples showed, a task should not run for long periods of time. Tasks do not preempt each other, but a task can be preempted by a hardware interrupts (which we haven't seen yet). If you need to run a series of long operations, you should dispatch a separate task for each operation, rather than using one big task. The post operation returns an error_t, whose value is either SUCCESS or FAIL. A post fails if and only if the task is already pending to run (it has been posted successfully and has not been invoked yet).(1)

For example, try this:

```
uint32_t i;

task void computeTask() {
  uint32_t start = i;
  for (;i < start + 10000 && i < 400001; i++) {}
  if (i >= 400000) {
    i = 0;
  }
  else {
    post computeTask();
  }
}
```

This code breaks the compute task up into many smaller tasks. Each invocation of computeTask runs through 10,000 iterations of the loop. If it hasn't completed all 400,001 iterations, it reposts itself. Compile this code and run it; it will run fine on both Telos and mica-family motes.

Note that using a task in this way required including another variable (i) in the component. Because computeTask() returns after 10,000 iterations, it needs somewhere to store its state for the next invocation. In this situation, i is acting as a static function variable often does in C. However,

as nesC component state is completely private, using the static keyword to limit naming scope is not as useful. This code, for example, is equivalent:

```
task void computeTask() {
  static uint32_t i;
  uint32_t start = i;
  for (;i < start + 10000 && i < 400001; i++) {}
  if (i >= 400000) {
    i = 0;
  }
  else {
    post computeTask();
  }
}
```

Internal Functions

Commands and events are the only way that a function in a component can be made callable by another component. There are situations when a component wants private functions for its own internal use. A component can define standard C functions, which other components cannot name and therefore cannot invoke directly. While these functions do not have the command or event modifier, they can freely call commands or signal events. For example, this is perfectly reasonable nesC code:

```
module BlinkC {
  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
{

  void startTimers() {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }

  event void Boot.booted()
  {
```

```
    startTimers();
  }

  event void Timer0.fired()
  {
    call Leds.led0Toggle();
  }

  event void Timer1.fired()
  {
    call Leds.led1Toggle();
  }

  event void Timer2.fired()
  {
    call Leds.led2Toggle();
  }
}
```

Internal functions act just like C functions: they don't need the `call` or `signal` keywords.

Split-Phase Operations

Because nesC interfaces are wired at compile time, callbacks (events) in TinyOS are very efficient. In most C-like languages, callbacks have to be registered at run-time with a function pointer. This can prevent the compiler from being able to optimize code across callback call paths. Since they are wired statically in nesC, the compiler knows exactly what functions are called where and can optimize heavily.

The ability to optimize across component boundaries is very important in TinyOS, because it has no blocking operations. Instead, every long-running operation is **split-phase**. In a blocking system, when a program calls a long-running operation, the call does not return until the operation is complete: the program blocks. In a split-phase system, when a program calls a long-running operation, the call returns immediately, and the called abstraction issues a callback when it completes. This approach is called split-phase because it splits invocation and completion into two separate phases of execution. Here is a simple example of the difference between the two:

Blocking

```
if (send() == SUCCESS)
{
  sendCount++;
}
```

Split-Phase

```
// start phase
send();

//completion phase
```

```
void sendDone(error_t err)
{
  if (err == SUCCESS) {
    sendCount++;
  }
}
```

Split-phase code is often a bit more verbose and complex than sequential code. But it has several advantages. First, split-phase calls do not tie up stack memory while they are executing. Second, they keep the system reponsive: there is never a situation when an application needs to take an action but all of its threads are tied up in blocking calls. Third, it tends to reduce stack utilization, as creating large variables on the stack is rarely necessary.

Split-phase interfaces enable a TinyOS component to easily start several operations at once and have them execute in parallel. Also, split-phase operations can save memory. This is because when a program calls a blocking operation, all of the state it has stored on the call stack (e.g., variables declared in functions) have to be saved. As determining the exact size of the stack is difficult, operating systems often choose a very conservative and therefore large size. Of course, if there is data that has to be kept across the call, split-phase operations still need to save it.

The command Timer.startOneShot is an example of a split-phase call. The user of the Timer inteface calls the command, which returns immediately. Some time later (specified by the argument), the component providing Timer signals Timer.fired. In a system with blocking calls, a program might use sleep():

Blocking

```
state = WAITING;
op1();
sleep(500);
op2();
state = RUNNING
```

Split-phase

```
state = WAITING;
op1();
call Timer.startOneShot(500);

event void Timer.fired() {
  op2();
  state = RUNNING;
}
```

In the next lesson, we'll look at one of the most basic split-phase operations: sending packets.

Related Documentation

- **TEP 102: Timers**
- **TEP 106: Schedulers and Tasks**

(1) The task semantics have changed significantly from tinyos-2.x. In 1.x, a task could be posted more than once and a post could fail if the task queue were full. In 2.x, a basic post will only fail if that task has already been posted and has not started execution. So a task can always run, but can only have one outstanding post at any time. If a component needs to post task several times, then the end of the task logic can repost itself as need be.

Lesson 3: Mote-mote radio communication

Last Modified: June 27 2006

This lesson introduces radio communications in TinyOS. You will become familar with TinyOS interfaces and components that support communications and you will learn how to:

- Use `message_t`, the TinyOS 2.0 message buffer.
- Send a message buffer to the radio.
- Receive a message buffer from the radio.

Introduction

TinyOS provides a number of *interfaces* to abstract the underlying communications services and a number of *components* that *provide* (implement) these interfaces. All of these interfaces and components use a common message buffer abstraction, called `message_t`, which is implemented as a nesC struct (similar to a C struct). The `message_t` abstraction replaces the TinyOS 1.x `TOS_Msg` abstraction. Unlike TinyOS 1.x, the members of `message_t` are opaque, and therefore not accessed directly. Rather, `message_t` is an *abstract data type*, whose members are read and written using accessor and mutator functions(1).

Basic Communications Interfaces

There are a number of interfaces and components that use `message_t` as the underlying data structure. Let's take a look at some of the interfaces that are in the `tos/interfaces` directory to familiarize ourselves with the general functionality of the communications system:

- `Packet` - Provides the basic accessors for the `message_t` abstract data type. This interface provides commands for clearing a message's contents, getting its payload length, and getting a pointer to its payload area.

- **`Send`** - Provides the basic *address-free* message sending interface. This interface provides commands for sending a message and canceling a pending message send. The interface provides an event to indicate whether a message was sent successfully or not. It also provides convenience functions for getting the message's maximum payload as well as a pointer to a message's payload area.
- **`Receive`** - Provides the basic message reception interface. This interface provides an event for receiving messages. It also provides, for convenience, commands for getting a message's payload length and getting a pointer to a message's payload area.
- **`PacketAcknowledgements`** - Provides a mechanism for requesting acknowledgements on a per-packet basis.
- **`RadioTimeStamping`** - Provides time stamping information for radio transmission and reception.

Active Message Interfaces

Since it is very common to have multiple services using the same radio to communicate, TinyOS provides the Active Message (AM) layer to multiplex access to the radio. The term "AM type" refers to the field used for multiplexing. AM types are similar in function to the Ethernet frame type field, IP protocol field, and the UDP port in that all of them are used to multiplex access to a communication service. AM packets also includes a destination field, which stores an "AM address" to address packets to particular motes. Additional interfaces, also located in the `tos/interfaces` directory, were introduced to support the AM services:

- **`AMPacket`** - Similar to `Packet`, provides the basic AM accessors for the `message_t` abstract data type. This interface provides commands for getting a node's AM address, an AM packet's destination, and an AM packet's type. Commands are also provides for setting an AM packet's destination and type, and checking whether the destination is the local node.
- **`AMSend`** - Similar to `Send`, provides the basic Active Message sending interface. The key difference between `AMSend` and `Send` is that `AMSend` takes a destination AM address in its `send` command.

The AM address of a node can be set at installation time, using the `make install.n` or `make reinstall.n` commands. It can be changed at runtime using the `ActiveMessageAddressC` component (see below).

Components

A number of components implement the basic communications and active message interfaces. Let's take a look at some of the components in the `/tos/system` directory. You should be familiar with these components because your code needs to specify both the *interfaces* your application *uses* as well as the *components* which *provide* (implement) those interfaces:

- **AMReceiverC** - Provides the following interfaces: Receive, Packet, and AMPacket.
- **AMSenderC** - Provides AMSend, Packet, AMPacket, and PacketAcknowledgements as Acks.
- **AMSnooperC** - Provides Receive, Packet, and AMPacket.
- **AMSnoopingReceiverC** - Provides Receive, Packet, and AMPacket.
- **ActiveMessageAddressC** - Provides commands to get and set the node's active message address. This interface is not for general use and changing the a node's active message address can break the network stack, so avoid using it unless you know what you are doing.

Naming Wrappers

Since TinyOS supports multiple platforms, each of which might have their own implementation of the radio drivers, an additional, platform-specific, naming wrapper called ActiveMessageC is used to bridge these interfaces to their underlying, platform-specific implementations. ActiveMessageC provides most of the communication interfaces presented above. Platform-specific versions of ActiveMessageC, as well the underlying implementations which may be shared by multiple platforms (e.g. Telos and MicaZ) include:

- ActiveMessageC for the **eyesIFX** platform is implemented by **Tda5250ActiveMessageC**.
- ActiveMessageC for the **intelmote2**, **micaz**, **telosa**, and **telosb** are all implemented by **CC2420ActiveMessageC**.
- ActiveMessageC for the **mica2** platform is implemented by **CC1000ActiveMessageC**.

The TinyOS 2.0 Message Buffer

TinyOS 2.0 introduces a new message buffer abstraction called message_t. If you are familiar with earlier versions of TinyOS, you need to know that message_t replaces TOS_Msg. The message_t structure is defined in **tos/types/message.h**.

```
typedef nx_struct message_t {
  nx_uint8_t header[sizeof(message_header_t)];
  nx_uint8_t data[TOSH_DATA_LENGTH];
  nx_uint8_t footer[sizeof(message_header_t)];
  nx_uint8_t metadata[sizeof(message_metadata_t)];
} message_t;
```

Note: The header, footer, and metadata fields are all opaque and must not be accessed directly. It is important to access the `message_t` fields only through `Packet`, `AMPacket`, and other such interfaces, as will be demonstrated in this tutorial. The rationale for this approach is that it allows the data (payload) to be kept at a fixed offset, avoiding a copy when a message is passed between two link layers. See Section 3 in **TEP 111** for more details.

Sending a Message over the Radio

We will now create a simple application that increments a counter, displays the counter's three least significant bits on the three LEDs, and sends a message with the counter value over the radio. Our implementation will use a single timer and a counter, in a way similar to the `BlinkSingle` example from **lesson 2**.

Reimplementing Blink

As a first step, we can reimplement `Blink` using a single timer and counter. Create a new directory in `apps` named `BlinkToRadio`:

```
$ cd tinyos-2.x/apps
$ mkdir BlinkToRadio
```

Inside this directory, create a file `BlinkToRadioC.nc`, which has this code:

```
#include <Timer.h>
#include "BlinkToRadio.h"

module BlinkToRadioC {
  uses interface Boot;
  uses interface Leds;
  uses interface Timer<TMilli> as Timer0;
}
implementation {
  uint16_t counter = 0;

  event void Boot.booted() {
    call Timer0.startPeriodic(TIMER_PERIOD_MILLI);
  }

  event void Timer0.fired() {
    counter++;
    call Leds.set(counter);
  }
}
```

Let's look at a few specific lines in this program. First, notice the C preprocessor `include` directive on the first line. This directive tells the compiler to simply replace the directive with the entire contents of `Timer.h`. The compiler looks for `Timer.h` in the *standard* places. In this case, standard means the TinyOS system directories that are located in `tos` or its subdirectories. It is possible to tell the compiler to look beyond these standard directories by using the `-I` flag in the Makefile, for example, as is common when including contributed libraries located in `contrib` directory tree.

The second line of this program is also an `include` directive, but note that it uses quotes around the filename rather than angle brackets. The quotes tell the preprocessor to look in the current directory before searching through the standard directories for the particular file. In this case, the `BlinkToRadio.h` file is located in the same directory and defines some constants that are used in this program. We will take a look at `BlinkToRadio.h` in just a bit.

Next, the call to `Leds.set` directly sets the three LEDs to the three low-order bits of the counter.

Finally, note the `call Timer0.startPeriodic(TIMER_PERIOD_MILLI)` line in the `Boot.booted` function. The value of `TIMER_PERIOD_MILLI` is defined in the `BlinkToRadio.h` header file:

```
#ifndef BLINKTORADIO_H
#define BLINKTORADIO_H

enum {
  TIMER_PERIOD_MILLI = 250
};

#endif
```

`BlinkToRadio.h` is a pretty standard header file but there are two things to note here. First, notice the use of the `ifndef`, `define`, and `endif` directives. These directives are used to ensure that the definitions in each header file is not included multiple times because the compiler would complain about multiply-defined objects. By convention, the literal used for these directives is an all-caps version of the filename with any periods converted to underscores. The other important thing to note is the use of an `enum` declaration for defining the constant `TIMER_PERIOD_MILLI`. Using `enum` for defining constants is preferred over using `define` because `enum` does not indiscriminately replace every occurrence of the `define`d literal, regardless of where it appears in the source. As a result, `enum`s provide better scoping as well.

`BlinkToRadioC.nc` provides the *implementation* logic of the program and `BlinkToRadio.h` defines constants and/or data structures. A third file is needed to *wire* the interfaces that the implementation `uses` to the actual components which `provide` these interfaces. The `BlinkToRadioAppC.nc` provides the needed wiring:

```
#include <Timer.h>
#include "BlinkToRadio.h"

configuration BlinkToRadioAppC {
}
implementation {
  components MainC;
  components LedsC;
  components BlinkToRadioC as App;
  components new TimerMilliC() as Timer0;

  App.Boot -> MainC;
  App.Leds -> LedsC;
  App.Timer0 -> Timer0;
}
```

The `BlinkToRadioAppC` should look familiar to you since it is essentially a subset of the `Blink` application/configuration from an earlier lesson.

These three files constitute all of the application code: the only other thing it needs is a Makefile. Create a file named `Makefile`. For an application as simple as this one, the Makefile is very short:

```
COMPONENT=BlinkToRadioAppC
include $(MAKERULES)
```

The first line tells the TinyOS make system that the top-level application component is BlinkToRadioAppC. The second line loads in the TinyOS build system, which has all of the rules for building and installing on different platforms.

Defining a Message Structure

Now that `Blink` has been reimplemented using a single timer and counter, we can now turn our attention to defining a message format to send data over the radio. Our message will send both the node id and the counter value over the radio. Rather than directly writing and reading the payload area of the `message_t` with this data, we will use a structure to hold them and then use structure assignment to copy the data into the message payload area. Using a structure allows

reading and writing the message payload more conveniently when your message has multiple fields or multi-byte fields (like uint16_t or uint32_t) because you can avoid reading and writing bytes from/to the payload using indices and then shifting and adding (e.g. `uint16_t x = data[0] << 8 + data[1]`). Even for a message with a single field, you should get used to using a structure because if you ever add more fields to your message or move any of the fields around, you will need to manually update all of the payload position indices if you read and write the payload at a byte level. Using structures is straightforward. To define a message structure with a `uint16_t` node id and a `uint16_t` counter in the payload, we add the following lines to `BlinkToRadio.h`, just before the `endif` directive:

```
typedef nx_struct BlinkToRadioMsg {
  nx_uint16_t nodeid;
  nx_uint16_t counter;
} BlinkToRadioMsg;
```

If this code doesn't look even vaguely familiar, you should spend a few minutes reading up on C structures. If you are familiar with C structures, this syntax should look familar but the `nx_` prefix on the keywords `struct` and `uint16_t` should stand out. The `nx_` prefix is specific to the nesC language and signifies that the `struct` and `uint16_t` are *external types*(3)(4). External types have the same representation on all platforms. The nesC compiler generates code that transparently reorders access to `nx_` data types and eliminates the need to manually address endianness and alignment (extra padding in structs present on some platforms) issues. So what is endianness? Read on...

Different processors represent numbers in different ways in their memory: some processors use a "big endian" representation which means that the most significant byte of a multi-byte (e.g. 16- or 32-bit) number is located at a lower memory address than the least significant byte, while "little endian" stores data in exactly the opposite order. A problem arises when data is serialized and sent over the network because different processors will decode the same set of bytes in different ways, depending on their "endianness." The main difficulty endianness presents is that it requires operations to rearrange byte orders to match the network protocol specification or the processor architecture -- an annoying and error-prone process. The `htons`, `htonl`, `ntohs`, and `ntohl` calls used with the sockets API are an example of platform-specific calls that convert between network and host byte orders, but you have to remember to use them. The nesC programming language takes a different approach to the problem and defines *external types* which allow the programmer to avoid dealing with byte reordering. In particular, the `nx_` prefix on a type (e.g. `nx_uint16_t`) indicates the field is to serialized in big endian format. In contrast, the `nxle_` prefix signifies that the field is serialized in little endian format.

Sending a Message

Now that we have defined a message type for our application, `BlinkToRadioMsg`, we will next see how to send the message over the radio. Before beginning, let's review the purpose of

the application. We want a timer-driven system in which every firing of the timer results in (i) incrementing a counter, (ii) displaying the three lowest bits of the counter on the LEDs, and (iii) transmitting the node's id and counter value over the radio. To implement this program, we follow a number of simple steps, as described in the next paragraph.

First, we need to identify the interfaces (and components) that provide access to the radio and allow us to manipulate the `message_t` type. Second, we must update the `module` block in the `BlinkToRadioC.nc` by adding `uses` statements for the interfaces we need. Third, we need to declare new variables and add any initialization and start/stop code that is needed by the interfaces and components. Fourth, we must add any calls to the component interfaces we need for our application. Fifth, we need to implmement any events specified in the interfaces we plan on using. Sixth, the `implementation` block of the application configuration file, `BlinkToRadioApp.c`, must be updated by adding a `components` statement for each component we use that provides one of the interfaces we chose earlier. Finally, we need to wire the the interfaces used by the application to the components which provide those interfaces.

Let's walk through the steps, one-by-one:

1. **Identify the interfaces (and components) that provide access to the radio and allow us to manipulate the `message_t` type.**

   We will use the `AMSend` interface to send packets as well as the `Packet` and `AMPacket` interfaces to access the `message_t` abstract data type. Although it is possible to wire directly to the `ActiveMessageC` component, we will instead use the `AMSenderC` component. However, we still need to start the radio using the `ActiveMessageC.SplitControl` interface.

   The reason for using `AMSenderC` is because it provides a virtualized abstraction. Earlier versions of TinyOS did not virtualize access to the radio, so it was possible for two components that were sharing the radio to interfere with each other. It was not at all uncommon for one component to discover the radio was busy because some other component, unknown to the first component, was accessing the active message layer. Radio virtualization was introduced in TinyOS 2.0 to address this interference and `AMSenderC` was written to provide this virtualization. Every user of `AMSenderC` is provided with a 1-deep queue and the queues of all users are serviced in a fair manner.

2. **Update the `module` block in the `BlinkToRadioC.nc` by adding `uses` statements for the interfaces we need:**

```
3. module BlinkToRadioC {
4.   ...
5.   uses interface Packet;
6.   uses interface AMPacket;
```

```
7.   uses interface AMSend;
8.   uses interface SplitControl as AMControl;
9. }
```

Note that `SplitControl` has been renamed to `AMControl` using the `as` keyword. nesC allows interfaces to be renamed in this way for several reasons. First, it often happens that two or more components that are needed in the same module provide the same interface. The `as` keyword allows one or more such names to be changed to distinct names so that they can each be addressed individually. Second, interfaces are sometimes renamed to something more meaningful. In our case, `SplitControl` is a general interface used for starting and stopping components, but the name `AMControl` is a mnemonic to remind us that the particular instance of `SplitControl` is used to control the `ActiveMessageC` component.

10. **Declare any new variables and add any needed initialization code.**

    First, we need to declare some new module-scope variables. We need a `message_t` to hold our data for transmission. We also need a flag to keep track of when the radio is busy sending. These declarations need to be added in the `implementation` block of `BlinkToRadioC.nc`:

    ```
    implementation {
      bool busy = FALSE;
      message_t pkt;
      ...
    }
    ```

    Next, we need to handle the initialization of the radio. The radio needs to be started when the system is booted so we must call `AMControl.start` inside `Boot.booted`. The only complication is that in our current implementation, we start a timer inside `Boot.booted` and we are planning to use this timer to send messages over the radio but the radio can't be used until it has completed starting up. The radio signals that it has completed starting through the `AMControl.startDone` event. To ensure that we do not start using the radio before it is ready, we need to postpone starting the timer until after the radio has completed starting. We can accomplish this by moving the call to start the timer, which is now inside `Boot.booted`, to `AMControl.startDone`, giving us a new `Boot.booted` with the following body:

    ```
      event void Boot.booted() {
        call AMControl.start();
      }
    ```

We also need to implement the `AMControl.startDone` and `AMControl.stopDone` event handlers, which have the following bodies:

```
  event void AMControl.startDone(error_t err) {
    if (err == SUCCESS) {
      call
Timer0.startPeriodic(TIMER_PERIOD_MILLI);
    }
    else {
      call AMControl.start();
    }
  }

  event void AMControl.stopDone(error_t err) {
  }
```

If the radio is started successfully, `AMControl.startDone` will be called with the `error_t` parameter set to a value of `SUCCESS`. If the radio starts successfully, then it is appropriate to start the timer. If, however, the radio does not start successfully, then it obviously cannot be used so we try again to start it. This process continues until the radio starts, and ensures that the node software doesn't run until the key components have started successfully. If the radio doesn't start at all, a human operator might notice that the LEDs are not blinking as they are supposed to, and might try to debug the problem.

11. **Add any program logic and calls to the used interfaces we need for our application.**

    Since we want to transmit the node's id and counter value every time the timer fires, we need to add some code to the `Timer0.fired` event handler:

```
event void Timer0.fired() {
  ...
  if (!busy) {
    BlinkToRadioMsg* btrpkt = (BlinkToRadioMsg*)
(call Packet.getPayload(&pkt, NULL));
    btrpkt->nodeid = TOS_NODE_ID;
    btrpkt->counter = counter;
    if (call AMSend.send(AM_BROADCAST_ADDR, &pkt,
sizeof(BlinkToRadioMsg)) == SUCCESS) {
      busy = TRUE;
    }
  }
}
```

This code performs several operations. First, it ensures that a message transmission is not in progress by checking the busy flag. Then it gets the packet's payload portion and casts it to a pointer to the previously declared `BlinkToRadioMsg` external type. It can now use this pointer to initialise the packet's fields, and then send the packet by calling `AMSend.send`. The packet is sent to all nodes in radio range by specifying `AM_BROADCAST_ADDR` as the destination address. Finally, the test against SUCCESS verifies that the AM layer accepted the message for transmission. If so, the busy flag is set to true. For the duration of the send attempt, the packet is owned by the radio, and user code must not access it.

Note that we could have avoided using the `Packet` interface, as it's `getPayload` command is repeated within `AMSend`.

12. **Implmement any (non-initialization) events specified in the interfaces we plan on using.**

    Looking through the `Packet`, `AMPacket`, and `AMSend` interfaces, we see that there is only one `event` we need to worry about, `AMSend.sendDone`:

```
  /**
   * Signaled in response to an accepted send
request. msg is
   * the message buffer sent, and error indicates
whether
   * the send was successful.
   *
   * @param  msg   the packet which was submitted
as a send request
   * @param  error SUCCESS if it was sent
successfully, FAIL if it was not,
   *               ECANCEL if it was cancelled
   * @see send
   * @see cancel
   */
  event void sendDone(message_t* msg, error_t
error);
```

    This event is signaled after a message transmission attempt. In addition to signaling whether the message was transmitted successfully or not, the event also returns ownership of `msg` from `AMSend` back to the component that originally called the `AMSend.send` command. Therefore `sendDone` handler needs to clear the `busy` flag to indicate that the message buffer can be reused:

```
  event void AMSend.sendDone(message_t* msg,
error_t error) {
    if (&pkt == msg) {
      busy = FALSE;
    }
  }
```

Note the check to ensure the message buffer that was signaled is the same as the local message buffer. This test is needed because if two components wire to the same `AMSend`, *both* will receive a `sendDone` event after *either* component issues a `send` command. Since a component writer has no way to enforce that her component will not be used in this manner, a defensive style of programming that verifies that the sent message is the same one that is being signaled is required.

13. **Update the `implementation` block of the application configuration file by adding a `components` statement for each component used that provides one of the interfaces chosen earlier.**

The following lines can be added just below the existing `components` declarations in the `implementation` block of `BlinkToRadioAppC.nc`:

```
implementation {
  ...
  components ActiveMessageC;
  components new AMSenderC(AM_BLINKTORADIO);
  ...
}
```

These statements indicate that two components, `ActiveMessageC` and `AMSenderC`, will provide the needed interfaces. However, note the slight difference in their syntax. `ActiveMessageC` is a singleton component that is defined once for each type of hardware platform. `AMSenderC` is a generic, parameterized component. The `new` keyword indicates that a new instance of `AMSenderC` will be created. The `AM_BLINKTORADIO` parameter indicates the AM type of the `AMSenderC`. We can extend the `enum` in the `BlinkToRadio.h` header file to incorporate the value of `AM_BLINKTORADIO`:

```
...
enum {
  AM_BLINKTORADIO = 6,
  TIMER_PERIOD_MILLI = 250
};
...
```

14. **Wire the the interfaces used by the application to the components which provide those interfaces.**

    The following lines will wire the used interfaces to the providing components. These lines should be added to the bottom of the `implementation` block of `BlinkToRadioAppC.nc`:

```
implementation {
  ...
  App.Packet -> AMSenderC;
  App.AMPacket -> AMSenderC;
  App.AMSend -> AMSenderC;
  App.AMControl -> ActiveMessageC;
}
```

Receiving a Message over the Radio

Now that we have an application that is transmitting messages, we can add some code to receive and process the messages. Let's write code that, upon receiving a message, sets the LEDs to the three least significant bits of the counter in the message. To make this application interesting, we will want to remove the line ~~`call Leds.set(counter);`~~ from the `Timer0.fired` event handler. Otherwise, both the timer events and packet receptions will update the LEDs and the resulting effect will be bizarre.

If two motes are programmed with our modified application, then each will display the other mote's counter value. If the motes go out of radio range, then the LEDs will stop changing. You can even investigate link asymmetry by trying to get one mote's LEDs to keep blinking while the other mote's LEDs stop blinking. This would indicate that the link from the non-blinking mote to blinking mote was available but that the reverse channel was no longer available.

1. **Identify the interfaces (and components) that provide access to the radio and allow us to manipulate the `message_t` type.**

   We will use the `Receive` interface to receive packets.

2. **Update the module block in the BlinkToRadioC.nc by adding uses statements for the interfaces we need:**

```
module BlinkToRadioC {
  ...
  uses interface Receive;
}
```

7. **Declare any new variables and add any needed initialization code.**

We will not require any new variables to receive and process messages from the radio.

8. **Add any program logic and calls to the used interfaces we need for our application.**

   Message reception is an event-driven process so we do not need to call cny commands on the `Receive`.

9. **Implemement any (non-initialization) events specified in the interfaces we plan on using.**

   We need to implement the `Receive.receive` event handler:

```
event message_t* Receive.receive(message_t* msg,
void* payload, uint8_t len) {
  if (len == sizeof(BlinkToRadioMsg)) {
    BlinkToRadioMsg* btrpkt =
(BlinkToRadioMsg*)payload;
    call Leds.set(btrpkt->counter);
  }
  return msg;
}
```

   The `receive` event handler performs some simple operations. First, we need to ensure that the length of the message is what is expected. Then, the message payload is cast to a structure pointer of type `BlinkToRadioMsg*` and assigned to a local variable. Then, the counter value in the message is used to set the states of the three LEDs.

   Note that we can safely manipulate the `counter` variable outside of an atomic section. The reason is that receive event executes in task context rather than interrupt context (events that have the `async` keyword can execute in interrupt context). Since the TinyOS execution model allows only one task to execute at a time, if all accesses to a variable occur in task context, then no race conditions will occur for that variable. Since all accesses to `counter` occur in task context, no critical sections are needed when accessing it.

10. **Update the implementation block of the application configuration file by adding a components statement for each component used that provides one of the interfaces chosen earlier.**

    The following lines can be added just below the existing `components` declarations in the implementation block of `BlinkToRadioAppsC.nc`:

```
implementation {
  ...
```

```
  components new AMReceiverC(AM_BLINKTORADIO);
  ...
}
```

This statement means that a new instance of `AMReceiverC` will be created. `AMReceiver` is a generic, parameterized component. The `new` keyword indicates that a new instance of `AMReceiverC` will be created. The `AM_BLINKTORADIO` parameter indicates the AM type of the `AMReceiverC` and is chosen to be the same as that used for the `AMSenderC` used earlier, which ensures that the same AM type is being used for both transmissions and receptions. `AM_BLINKTORADIO` is defined in the `BlinkToRadio.h` header file.

11. **Wire the the interfaces used by the application to the components which provide those interfaces.**

    Update the wiring by insert the following line just before the closing brace of the `implementation` block in BlinkToRadioAppC:

```
implementation {
  ...
  App.Receive -> AMReceiverC;
}
```

12. **Test your application!**

    Testing your application is easy. Get two motes. They can be mica2, micaz, telosa, telosb, or tmote. For this exercise, let's assume that the motes are telosb (if not, skip past the motelist part and program the mote using whatever the appropriate programmer parameters are for your hardware). Assuming you are using a telosb, first open a Cygwin or Linux shell and cd to the `apps/tutorials/BlinkToRadio` directory. Then, insert the first telosb mote into an available USB port on the PC and type `motelist` the at the Cygwin or Linux prompt ($). You should see exactly one mote listed. For example:

```
$ motelist
Reference  CommPort   Description
---------- ----------
----------------------------------------
UCC89MXV   COM17      Telos (Rev B 2004-09-27)
```

Now, assuming you are in the `apps/tutorials/BlinkToRadio` directory, type `make telosb install,1`. You should see a lot text scroll by that looks something like:

```
$ make telosb install,1
mkdir -p build/telosb
    compiling BlinkToRadioAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -mdisable-
hwmul -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d -
Wnesc-all -target=telosb
-fnesc-cfile=build/telosb/app.c -board=
BlinkToRadioAppC.nc -lm
    compiled BlinkToRadioAppC to
build/telosb/main.exe
            9040 bytes in ROM
             246 bytes in RAM
msp430-objcopy --output-target=ihex build/telosb/
main.exe build/telosb/main.ihex
    writing TOS image
tos-set-symbols --objcopy msp430-objcopy --
objdump msp430-objdump --target ihex
build/telosb/main.ihex
build/telosb/main.ihex.out-1 TOS_NODE_ID=1
ActiveMessageAddressC$addr=1
    found mote on COM17 (using bsl,auto)
    installing telosb binary using bsl
tos-bsl --telosb -c 16 -r -e -I -p
build/telosb/main.ihex.out-1
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device
ID: f16c)
Changing baudrate to 38400 ...
Program ...
9072 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out-1
build/telosb/main.ihex.out-1
```

Now, remove the first telosb from the USB port, insert the batteries, and set it aside. Insert the second telos into the USB port and once again type `motelist`. You should again see something like:

```
$ motelist
Reference  CommPort   Description
```

```
---------- ----------
---------------------------------------
UC9VN03I   COM14      Telos (Rev B 2004-09-27)
```

Finally, type `make telosb reinstall,2` and you should once again see something like the following scroll by:

```
$ make telosb reinstall,2
tos-set-symbols --objcopy msp430-objcopy --
objdump msp430-objdump --target ihex
build/telosb/main.ihex
build/telosb/main.ihex.out-2 TOS_NODE_ID=2
ActiveMessageAddressC$addr=2
    found mote on COM14 (using bsl,auto)
    installing telosb binary using bsl
tos-bsl --telosb -c 13 -r -e -I -p
build/telosb/main.ihex.out-2
MSP430 Bootstrap Loader Version: 1.39-telos-8
Mass Erase...
Transmit default password ...
Invoking BSL...
Transmit default password ...
Current bootstrap loader version: 1.61 (Device
ID: f16c)
Changing baudrate to 38400 ...
Program ...
9072 bytes programmed.
Reset device ...
rm -f build/telosb/main.exe.out-2
build/telosb/main.ihex.out-2
```

At this point, both motes should be blinking their LEDs. If you press the RESET button on either telosb, then the LEDs on the *other* telosb will pause on whatever was being displayed at the moment you pressed RESET. When you release the RESET the button, the paused mote will be reset and then resume counting from one.

Conclusions

This lesson has introduced radio communications TinyOS 2.x.

Related Documentation

- (1) **TEP 111: message_t**
- (2) **TEP 116: Packet Protocols**

(3)**Programming Hint 15:**Always use platform independent types when defining message formats. From Phil Levis' ***TinyOS Programming***

(4)**Programming Hint 16:**If you have to perform significant computation on a platform independent type or access it many (hundreds or more) times, then temporarily copying it to a native type can be a good idea. From Phil Levis' ***TinyOS Programming***

Lesson 4: Mote-PC serial communication and SerialForwarder

Last updated June 27 2006

To goal of this lesson is to show you how to communicate with a mote from a PC. This will allow you to collect data from the network, send commands to motes, and monitor network traffic.

This tutorial presents the Java-based infrastructure for communicating with motes. There is also a C-based infrastructure, found in support/sdk/c. Please see the documentation found there, and the `mig` and `ncg` man pages for more details.

Packet sources and TestSerial

The first step is to check that you are able to get your PC to communicate with a mote. Most motes have a serial port or similar interface. For example, the mica family can directly control a serial port: programming boards basically connect the mote's serial port pins to the actual serial port on the board. Telos motes also have a serial interface, but it talks to their USB hardware, which is similar in functionality but very different in terms of cables and connectors.

The basic abstraction for mote-PC communication is a **packet source**. A packet source is exactly that: a communication medium over which an application can receive packets from and send packets to a mote. Examples of packet sources include serial ports, TCP sockets, and the SerialForwarder tool. Most TinyOS communication tools take an optional `-comm` parameter, which allows you to specify the packet source as a string. For example:

```
$ java net.tinyos.tools.Listen -comm serial@COM1:telos
```

tells the Listen tool to use the COM1 serial port (on a Windows machine) at the correct speed for a telos mote, while

```
$ java net.tinyos.tools.Listen -comm
serial@/dev/ttyS0:micaz
```

tells Listen to use the serial port `/dev/ttyS0` (on a UNIX machine) at the correct speed for a micaz mote.

The first step to testing your serial port is to install the `apps/tests/TestSerial` application on a mote. This application sends a packet to the serial port every second, and when it receives a packet over the serial port it displays the packet's sequence number on the LEDs.

Once you have installed `TestSerial`, you need to run the corresponding Java application that communicates with it over the serial port. This is built when you build the TinyOS application. From in the application directory, type:

```
$ java TestSerial
```

If you get a message like

```
The java class is not found:  TestSerial
```

it means that you either haven't compiled the Java code (try running `make` *`platform`* again) or you don't have `.` (the current directory) in your Java `CLASSPATH`.

Because you haven't specified a packet source, TestSerial will fall back to a default, which is a SerialForwarder. Since you don't have a SerialForwarder running, TestSerial will exit, complaining that it can't connect to one. So let's specify the serial port as the source. The syntax for a serial port source is as follows:

```
   serial@<PORT>:<SPEED>
```

PORT depends on your platform and where you have plugged the mote in. For Windows/Cygwin platforms, it is COM*N*, where *N* is the port number. For Linux/UNIX machines, it is `/dev/ttyS`*N* for a built-in serial port, or one of `/dev/ttyUSB`*N* or `/dev/usb/tts/`*N* for a serial-over-USB port. Additionally as we saw in **lesson 1**, on Linux you will typically need to make this serial port world writeable. As superuser, execute the following command:

```
chmod 666 serialport
```

The SPEED can either be a numeric value, or the name of a platform. Specifying a platform name tells the serial packet source to use the default speed for the platform. Valid platforms are:

Platform	Speed (baud)

telos	115200
telosb	115200
tmote	115200
micaz	57600
mica2	57600
mica2dot	19200
eyes	115200
intelmote2	115200

The Java file `support/sdk/java/net/tinyos/packet/BaudRate.java` determines these mappings. Unlike in TinyOS 1.x, all platforms have a common serial packet format. Following the table, these two serial specfications are identical:

```
serial@COM1:micaz
serial@COM1:57600
```

If you run `TestSerial` with the proper PORT and SPEED settings, you should see output like this:

```
Sending packet 1
Received packet sequence number 4
Sending packet 2
Received packet sequence number 5
Sending packet 3
Received packet sequence number 6
Sending packet 4
Received packet sequence number 7
Received packet sequence number 8
Sending packet 5
Received packet sequence number 9
Sending packet 6
```

and the mote LEDs will blink.

Cannot find JNI error

If you try to run TestSerial and receive an error that Java cannot find TOSComm JNI support, this means the Java Native Interface (JNI) files that control the serial port haven't been correctly installed. Run the command `tos-install-jni` (on Linux, do this as the superuser). If this command does not exist, then you have either not installed the tinyos-tools RPM or it was

installed incorrectly. The `tos-` commands are typically installed in /usr/bin. If you still cannot find the script, email `tinyos-help`.

Installing `tos-install-jre` from CVS sources

If you have not installed the tools RPM and are working directly from the TinyOS CVS repository, you can manually install the tos-locate-jre script. Go to `tinyos-2.x/tools/tinyos/java`. If the directory has a `Makefile` in it, type `make` and (again, on Linux, as superuser) `make install`. If the directory does not have a `Makefile`, go to `tinyos-2.x/tools` and type:

```
$ ./Bootstrap
$ ./configure
$ ./make
$ ./make install
```

Then type `tos-install-jni`. This should install serial support in your system.

MOTECOM

If you do not pass a `-comm` parameter, then tools will check the `MOTECOM` environment variable for a packet source, and if there is no `MOTECOM`, they default to a SerialForwarder. This means that if you're always communicating with a mote over your serial port, you can just set `MOTECOM` and no longer have to specify the `-comm` parameter. For example:

```
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
```

Try setting your MOTECOM variable and running TestSerial without a `-comm` parameter.

BaseStation and net.tinyos.tools.Listen

`BaseStation` is a basic TinyOS utility application. It acts as a bridge between the serial port and radio network. When it receives a packet from the serial port, it transmits it on the radio; when it receives a packets over the radio, it transmits it to the serial port. Because TinyOS has a

toolchain for generating and sending packets to a mote over a serial port, using a BaseStation allows PC tools to communicate directly with mote networks.

Take one of the two nodes that had BlinkToRadio (from **lesson 3**) installed and install BaseStation on it. If you turn on the node that still has BlinkToRadio installed, you should see LED 1 on the BaseStation blinking. BaseStation toggles LED 0 whenever it sends a packet to the radio and LED 1 whenever it sends a packet to the serial port. It toggles LED 2 whenever it has to drop a packet: this can happen when one of the two receives packets faster than the other can send them (e.g., receiving micaZ radio packets at 256kbps but sending serial packets at 57.6kbps).

BaseStation is receiving your BlinkToRadio packets and sending them to the serial port, so if it is plugged into a PC we can view these packets. The Java tool Listen is a basic packet sniffed: it prints out the binary contents of any packet it hears. Run Listen, using either MOTECOM or a -comm parameter:

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

Listen creates a packet source and just prints out every packet it sees. Your output should look something like this:

```
00 FF FF 04 22 06 00 02 00 01
00 FF FF 04 22 06 00 02 00 02
00 FF FF 04 22 06 00 02 00 03
00 FF FF 04 22 06 00 02 00 04
00 FF FF 04 22 06 00 02 00 05
00 FF FF 04 22 06 00 02 00 06
00 FF FF 04 22 06 00 02 00 07
00 FF FF 04 22 06 00 02 00 08
00 FF FF 04 22 06 00 02 00 09
00 FF FF 04 22 06 00 02 00 0A
00 FF FF 04 22 06 00 02 00 0B
```

Listen is simply printing out the packets that are coming from the mote. Each data packet that comes out of the mote contains several fields of data. The first byte (00) indicates that this is packet is an AM packet. The next fields are the generic Active Message fields, defined in tinyos-2.x/tos/serial/Serial.h. Finally, the remaining fields are the data payload of the message, which was defined in BlinkToRadio.h as:

```
typedef nx_struct BlinkToRadioMsg {
  nx_uint16_t nodeid;
  nx_uint16_t counter;
} BlinkToRadioMsg;
```

The overall message format for the BlinkToRadioC application is therefore (ignoring the first 00 byte):

- **Destination address** (2 bytes)
- **Message length** (1 byte)
- **Group ID** (1 byte)
- **Active Message handler type** (1 byte)
- **Payload** (up to 28 bytes):
    - **source mote ID** (2 bytes)
    - **sample counter** (2 bytes)

So we can interpret the data packet as follows:

dest addr	msg len	groupID	handlerID	source addr	counter
ff ff	04	22	06	00 02	00 0B

The source address depends on what mote ID you installed your BlinkToRadio application with. The default (if you do not specify and ID) is `00 01`. Note that the data is sent by the mote in *big-endian* format; for example, `01 02` means 258 (256*1 + 2). This format is independent of the endian-ness of the processor, because the packet format is an `nx_struct`, which is a network format, that is, big-endian and byte-aligned. Using `nx_struct` (rather than a standard C `struct`) for a message payload ensures that it will work across platforms.

As you watch the packets scroll by, you should see the counter field increase as the BlinkToRadio app increments its counter.

MIG: generating packet objects

The `Listen` program is the most basic way of communicating with the mote; it just prints binary packets to the screen. Obviously it is not easy to visualize the sensor data using this program. What we'd really like is a better way of retrieving and observing data coming from the sensor network. Of course, exactly what data to display and how to visualize it can be very application specific. For this reason, TinyOS only has a few applications for visualizing simple sensor data (in the next lesson, you'll use the Oscilloscope application), but it provides support for building new visualization or logging systems.

One problem with Listen is that it just dumps binary data: a user has to be able to read the bytes and parse them into a given packet format. The TinyOS toolchain makes this process easier by providing tools for automatically generating message objects from packet descriptions. Rather than parse packet formats manually, you can use the `mig` (Message Interface Generator) tool to build a Java, Python, or C interface to the message structure. Given a sequence of bytes, the MIG-generated code will automatically parse each of the fields in the packet, and it provides a set of standard accessory and mutators for printing out received packets or generating new ones.

The mig tool takes three basic arguments: what programming language to generate code for (Java, Python, or C), which file in which to find the structure, and the name of the structure. The tool also takes standard C options, such as -I for includes and -D for defines. The TestSerial application, for

example, uses mig so that it can easily create and parse the packets over the serial port. If you go back to TestSerial and type `make clean;make`, you should see this:

```
rm -rf build *.class TestSerialMsg.java
rm -rf _TOSSIMmodule.so TOSSIM.pyc TOSSIM.py
mkdir -p build/telosb
mig java -target=telosb -I%T/lib/oski -java-
classname=TestSerialMsg TestSerial.h TestSerialMsg -o
TestSerialMsg.java
javac *.java
    compiling TestSerialAppC to a telosb binary
ncc -o build/telosb/main.exe -Os -O -mdisable-hwmul -
Wall -Wshadow -DDEF_TOS_AM_GROUP=0x66 -Wnesc-all -
DCC2420_DEF_CHANNEL=19 -target=telosb -fnesc-
cfile=build/telosb/app.c -board= -I%T/lib/oski
TestSerialAppC.nc -lm
    compiled TestSerialAppC to build/telosb/main.exe
            6300 bytes in ROM
             281 bytes in RAM
msp430-objcopy --output-target=ihex
build/telosb/main.exe build/telosb/main.ihex
    writing TOS image
```

Before building the TinyOS application, the Makefile has a rule for generating `TestSerialMsg.java`. It then compiles TestSerialMsg.java as well as TestSerial.java, and finally compiles the TinyOS application. Looking at the Makefile, we can see that it has a few more rules than the one for BlinkToRadio:

```
COMPONENT=TestSerialAppC
BUILD_EXTRA_DEPS += TestSerial.class
CLEAN_EXTRA = *.class TestSerialMsg.java

TestSerial.class: $(wildcard *.java)
TestSerialMsg.java
        javac *.java

TestSerialMsg.java:
        mig java -target=null -java-
classname=TestSerialMsg TestSerial.h TestSerialMsg -o
$@

include $(MAKERULES)
```

The BUILD_EXTRA_DEPS line tells the TinyOS make system that the TinyOS application has additional dependencies that must be satisfied before it can be built. The Makefile tells the make system that TestSerial.class, the Java application that we ran to test serial communication. The CLEAN_EXTRA line tells the make system extra things that need to be done when a user types make clean to clean up.

The BUILD_EXTRA_DEPS line tells make to compile TestSerial.class before the application; the line

```
TestSerial.class: $(wildcard *.java)
TestSerialMsg.java
	javac *.java
```

tells it that TestSerial.class depends on all of the .java files in the directory as well as TestSerialMsg.java. Once all of these dependencies are resolved, the make system will call javac *.java, which creates TestSerial.class. The final line,

```
TestSerialMsg.java:
	mig java -target=null -java-
classname=TestSerialMsg TestSerial.h TestSerialMsg -o
$@
```

tells the make system how to create TestSerialMsg.java, the Java class representing the packet sent between the mote and PC. Because TestSerialMsg.java is a dependency for TestSerial.class, make will create it if it is needed. To create TestSerialMsg.java, the Makefile invokes the mig tool. Let's step through the parameters one by one:

mig	Invoke mig
java	Build a Java class
-target=null	For the null platform
-java-classname=TestSerialMsg	Name the Java class TestSerialMsg
TestSerial.h	The structure is in TestSerial.h
TestSerialMsg	The structure is named TestSerialMsg
-o $@	Write the file to $@, which is TestSerialMsg.java

The null platform is a special platform which is convenient to use as the target when using mig. It includes all the standard system components, but with dummy do-nothing implementations. Building an application for the null platform is useless, but it allows mig to extract the layout of packets.

Let's build a Java packet object for BlinkToRadio. Open the Makefile for BlinkToRadio and add a dependency:

```
BUILD_EXTRA_DEPS=BlinkToRadioMsg.class
```

Then add a step which explains how to compile a .java to a .class:

```
BlinkToRadioMsg.class: BlinkToRadioMsg.java
	javac BlinkToRadioMsg.java
```

Note that there must be a tab before javac, and not just spaces. Finally, add the line which explains how to create BlinkToRadioMsg.java:

```
BlinkToRadioMsg.java:
	mig java -target=null -java-
classname=BlinkToRadioMsg BlinkToRadio.h
BlinkToRadioMsg -o $@
```

As with javac, there must be a tab (not spaces) before mig. Now, when you type `make` in `BlinkToRadio/`, the make system will compile BlinkToRadioMsg.class, a Java class that parses a binary packet into message fields that can be accessed through methods.

There is one more step, however. When you compiled, you probably saw this warning:

```
warning: Cannot determine AM type for BlinkToRadioMsg
	 (Looking for definition of
AM_BLINKTORADIOMSG)
```

One part of the TinyOS communication toolchain requires being able to figure out which AM types correspond to what kinds of packets. To determine this, for a packet type named X, mig looks for a constant of the form `AM_X`. The warning is because we defined our AM type as AM_BLINKTORADIO, but mig wants AM_BLINKTORADIOMSG. Modify BlinkToRadio.h so that it defines the latter. You'll also need to update BlinkToRadioAppC.nc so that the arguments to AMSenderC and AMReceiverC use it. Recompile the application, and you should see no warning. Install it on a mote.

Now that we have a Java message class, we can use it to print out the messages we see from the BaseStation. With BaseStation plugged into the serial port and BlinkToRadio running on another mote, from the BlinkToRadio directory type

```
java net.tinyos.tools.MsgReader BlinkToRadioMsg
```

Now, when the BaseStation sends a packet to the serial port, MsgReader reads it, looks at its AM type, and if it matches the AM type of one of the Java message classes passed at the command line, it prints out the packet. You should see output like this:

```
1152232617609: Message
  [nodeid=0x2]
```

```
  [counter=0x1049]

1152232617609: Message
  [nodeid=0x2]
  [counter=0x104a]

1152232617609: Message
  [nodeid=0x2]
  [counter=0x104b]

1152232617621: Message
  [nodeid=0x2]
  [counter=0x104c]
```

SerialForwarder and other packet sources

One problem with directly using the serial port is that only one PC program can interact with the mote. Additionally, it requires you to run the application on the PC which is physically connected to the mote. The SerialForwarder tool is a simple way to remove both of these limitations.

Most generally, the SerialForwarder program opens a packet source and lets many applications connect to it over a TCP/IP stream in order to use that source. For example, you can run a SerialForwarder whose packet source is the serial port; instead of connecting to the serial port directly, applications connect to the SerialForwarder, which acts as a proxy to read and write packets. Since applications connect to SerialForwarder over TCP/IP, applications can connect over the Internet.

SerialForwarder is the second kind of packet source. A SerialForwarder source has this syntax:

```
sf@HOST:PORT
```

HOST and PORT are optional: they default to localhost (the local machine) and 9002. For example,

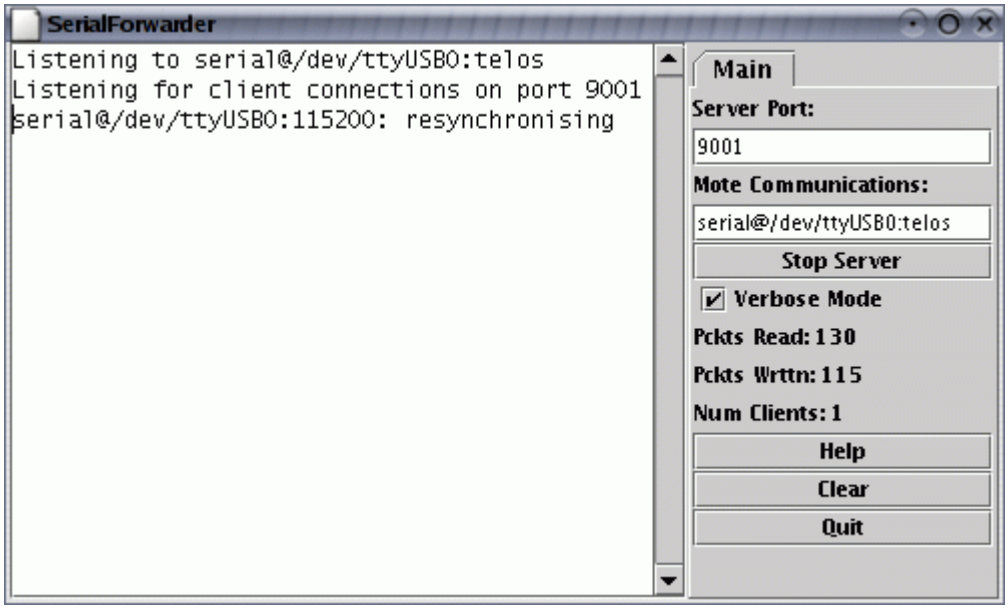
```
sf@dark.cs.berkeley.edu:1948
```

will connect to a SerialForwarder running on the computer dark.cs.berkeley.edu and port 1948.

The first step is to run a SerialForwarder; since it takes one packet source and exports it as an sf source, it takes a packet source parameter just like the other tools we've used so far: you can pass a -comm parameter, use MOTECOM, or just rely on the default. Close your MsgReader application so that it no longer uses the serial port, and run a SerialForwarder:

```
java net.tinyos.sf.SerialForwarder
```

You should see a window like this pop up:



Since SerialForwarder takes any packet source as its source, you can even string SerialForwaders along:

```
java net.tinyos.sf.SerialForwarder -port 9003 -comm
sf@localhost:9002
```

This command opens a second SerialForwarder, whose source is the first SerialForwarder. You'll see that the client count of the first one has increased to one. It's rare that you'd ever want to do this, but it demonstrates that in the message support libraries you can use a variety of packet sources.

Close the second SerialForwarder (the one listening on port 9003). Run MsgReader again, but this time tell it to connect to your SerialForwarder:

```
java net.tinyos.tools.MsgReader -comm
sf@localhost:9002 BlinkToRadioMsg
```

You will see the client count increment, and MsgReader will start printing out packets.

Packet Sources

In addition to serial ports and SerialForwarders, the TinyOS messaging library supports a third packet source, motes which are connected to an ethernet port through a Crossbow MIB 600 ethernet board. This is the full set of packet sources:

Syntax	Source
serial@PORT:SPEED	Serial ports
sf@HOST:PORT	SerialForwarder, TMote Connect
network@HOST:PORT	MIB 600

In the `network` packet source, the default MIB 600 port is 10002. The Moteiv TMote Connect appliance is a SerialForwarder packet source.

The tool side

Code for the Java messaging toolchain lives in two java packages: `net.tinyos.message` and `net.tinyos.packet`. The `packet` package contains all of the code for packet sources and their protocols: it is what reads and writes bytes. The `message` package is what turns streams of bytes into meaningful messages and provides packet source independent classes for communicating with motes.

The key class for sending and receiving packets is `MoteIF`. It has methods for registering packet listeners (callbacks when a packet arrives) and sending packets. The tools `MsgReader`, `Listen`, and `Send` are good places to start to learn how to get Java applications to communicate with motes.

There is also support for python and C.

Sending a packet to the serial port in TinyOS

Sending an AM packet to the serial port in TinyOS is very much like sending it to the radio. A component uses the AMSend interface, calls `AMSend.send`, and handles `AMSend.sendDone`. The serial stack will send it over the serial port regardless of the AM address specified.

The TinyOS serial stack follows the same programming model as the radio stack. There is a SerialActiveMessageC for turning the stack on and off (mote processors often cannot enter their lowest power state while the serial stack is on), and generic components for sending and receiving packets. As the serial stack is a dedicated link, however, it does not provide a snooping interface, and it does not filter based on the destination address of the packet. These are the serial communication components and their radio analogues:

Serial	Radio
SerialActiveMessageC	ActiveMessageC

SerialAMSenderC	AMSenderC
SerialAMReceiverC	AMReceiverC

Because serial AM communication has the same interfaces as radio AM communication, you can in most situations use them interchangably. For example, to make BlinkToRadio send packets to the serial port rather than the radio, all you have to do is change the BlinkToRadioAppC configuration:

Radio

```
  components
ActiveMessageC;
  components new
AMSenderC(AM_BLINKTORA
DIOMSG);

  BlinkToRadioC.AMSend
-> AMSenderC;

BlinkToRadioC.AMContro
l -> ActiveMessageC;
```

Serial

```
  components
SerialActiveMessageC;
  components new
SerialAMSenderC(AM_BLINKTOR
ADIOMSG);

  BlinkToRadioC.AMSend ->
SerialAMSenderC;
  BlinkToRadioC.AMControl -
> SerialActiveMessageC;
```

Now, rather than have BlinkToRadio send packets which a BaseStation recieves and forwards to the serial port, the application will send them directly to a serial port. Connect a MsgReader to test that this is happening. Note that the binary code and data size has changed significantly, as nesC has included the serial stack rather than the radio stack.

Related Documentation

- **TEP 113: Serial Communication**
- `mig` man page
- `ncg` man page
- javadoc documentation for the net.tinyos.packet and net.tinyos.message packages

< Previous Lesson | Top | Next Lesson >

CLASSPATH and Java classes

Note that the CLASSPATH variable points to `tinyos.jar`. This means that when Java looks for classes to load, it looks in tinyos.jar rather than the Java directories in `support/sdk/java`.

Therefore, if you change and recompile the Java classes, you will not see the changes, as Java will only look at the jar file. To regenerate the jar from the Java code, go to `support/sdk/java` and type `make tinyos.jar`.

Lesson 5: Sensing

Last Modified: 16 June 2006

This lesson introduces sensor data acquisition in TinyOS. It demonstrates two sensor applications: a simple application called **Sense** that periodically takes sensor readings and displays the values on the LEDs. And a more sophisticated application called **Oscilloscope** where nodes periodically broadcast their sensor readings to a basestation node. Using the Mote-PC serial communication described in the **previous lesson** the basestation forwards the sensor readings to the PC, where they are visualized with a dedicated graphical user interface.

Introduction

Sensing is an integral part of sensor network applications. In TinyOS 1.x sensing was syntactically connected with analog-to-digital converters (ADCs): TinyOS 1.x applications such as `Oscilloscope` or `Sense` used the `ADC` and `ADCControl` interfaces to collect sensor data. When new platforms appeared with sensors that were read out via the serial interface, not only did additional interfaces like `ADCError` have to be introduced, but it became clear that equating a sensor with an ADC was not always the appropriate thing to do.

Usually sensing involves two tasks: configuring a sensor (and/or the hardware module it is attached to, for example the ADC or SPI bus) and reading the sensor data. The first task is tricky, because a sensing application like, for example, `Sense` is meant to run on any TinyOS platform. How can `Sense` know the configuration details (like ADC input channel, the required reference voltage, etc.) of an attached sensor? It can't, because the configuration details of sensors will be different from platform to platform. Unless `Sense` knows about all sensors on all platforms it will be unable to perform the configuration task. However, the second task - reading the sensor data - can be solved so that the `Sense` application can collect sensor data even though it is agnostic to the platform it is running on.

In TinyOS 2.0 *platform independent* sensing applications such as `Oscilloscope`, `Sense` or `RadioSenseToLeds` do not use configuration interfaces like `ADCControl` anymore; instead they use the standard data acquisition interfaces `Read`, `ReadStream` or `ReadNow` for collecting sensor data. All configuration details are hidden from the application and this is why you can compile `Sense` and display sensor data on the

telosb or the *micaz* platform, even though the actual sensors and their connection to the rest of the system may be completely different.

This raises questions like the following:

- Since the `Sense` application component only uses standard data acquisition interfaces who is in charge of defining which sensor it samples?
- If the `Sense` application component is not configuring the sensor, then who is responsible for that?
- How can an applications like `Sense` display sensor data when they do not know the details about sensor configuration? This includes questions like "what is the value range of the sensor data" or "is a temperature reading to be interpreted in degrees Celsius or Fahrenheit"?
- Let's assume there are several sensors on a platform: what steps have to be taken to let the `Sense` or `Oscilloscope` application display data from a different sensor?

After reading this tutorial you should be able to answer these questions. Using the `Sense` and `Oscilloscope` application as an example, the following sections explain how the data acquisition interfaces are used, how the configuration procedure works and, as an example, how `Sense` can be hooked up to sensors other than the default one on the *telosb* platform.

The Sense application

`Sense` is a simple sensing demo application. It periodically samples the default sensor and displays the bottom bits of the readings on the LEDs. The `Sense` application can be found in `tinyos-2.x/apps/Sense`. Let's first look at the **`SenseAppC.nc`** configuration:

```
configuration SenseAppC
{
}
implementation {
  components SenseC, MainC, LedsC, new TimerMilliC();
  components new DemoSensorC() as Sensor;

  SenseC.Boot -> MainC;
  SenseC.Leds -> LedsC;
  SenseC.Timer -> TimerMilliC;
  SenseC.Read -> Sensor;
}
```

The `SenseAppC` configuration looks similar to the `BlinkAppC` configuration described in **lesson 1** (if you have not done so, read the sections on the Blink application in lesson 1). To understand the wiring let's look at the signature of the **`SenseC.nc`** module:

```
module SenseC
```

```
{
  uses {
    interface Boot;
    interface Leds;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
  }
}
```

Like the `BlinkC.nc` module the `SenseC.nc` module uses the interfaces `Boot`, `Leds` and `Timer<TMilli>`. Additionally, it uses the `Read<uint16_t>` interface. The sequence of actions in the `SenseC.nc` implementation is as follows: `SenseC.nc` uses the `Boot` interface to start a periodic timer after the system has been initialized. Every time the timer expires `SenseC.nc` is signalled a timer event and reads data via the `Read<uint16_t>` interface. Reading data is a split-phase operation, it is divided in a command `Read.read()` and an event `Read.readDone()`. Thus every time the timer expires `SenseC.nc` calls `Read.read()` and waits for the `Read.readDone()` event. When data is signalled in the `Read.readDone()` event `SenseC.nc` displays it on the leds: the least significant bit is displayed on LED 0 (0 = off, 1 = on), the second least significant bit is displayed on LED 1 and so on.

The **Read** interface (in `tinyos-2.x/tos/interfaces`) can be used to read a single piece of sensor data, let's look at it in detail:

```
interface Read<val_t> {
  /**
   * Initiates a read of the value.
   *
   * @return SUCCESS if a readDone() event will
eventually come back.
   */
  command error_t read();

  /**
   * Signals the completion of the read().
   *
   * @param result SUCCESS if the read() was
successful
   * @param val the value that has been read
   */
  event void readDone( error_t result, val_t val );
}
```

If you are not familiar with generic interfaces you will wonder what the meaning of `<val_t>` (in the first line) is and why the signature of `SenseC.nc` is using `Read<uint16_t>`. What you see above is a *generic interface definition*, because the `Read` interface takes a type

parameter. Generic interfaces are explained in the nesC Language Reference Manual (version 1.2 and above). For now it is enough to know that generic interfaces have at least one type parameter and two components can be wired together only if they provide/use the interface with the same types (note that the `readDone` event passes a parameter of the `<val_t>` parameter, which is a placeholder for the actual data type). This means that since `SenseC.nc` is using the `uint16_t` variant of the `Read` interface, it can only be wired to a component that provides the `Read<uint16_t>` interface and thus `SenseC.nc` expects to read 16 bit unsigned integer sensor data. If you tried to wire `SenseC.nc` to a component that provides, for example, a `Read<uint8_t>` interface you would get an error from the nesC compiler.

Recall that the wiring is defined in the `SenseAppC.nc` configuration. Let's again take a look at which component `SenseC.nc` is wired to using the `Read<uint16_t>` interface in the `SenseAppC` configuration. The interesting lines are

```
  components new DemoSensorC() as Sensor;
```

and

```
  SenseC.Read -> Sensor;
```

This means that the *generic* `DemoSensorC` component provides the `Read<uint16_t>` interface to `SenseC.nc`

It is important to understand that the `SenseC.nc` module has no way of telling which sensor it is connected to; in fact it cannot even tell whether it is getting data from a sensor at all, because it can be wired to any component that provides a `Read<uint16_t>` interface. On a platform without any built-in sensors (like *micaz*) and no attached sensorboard the `DemoSensorC` component could simply return constant values. The last sentence hints that the `DemoSensorC` component is different for every platform: therefore you will not find `DemoSensorC.nc` in the TinyOS libraries. Instead, a different `DemoSensorC.nc` has to be written for every platform, i.e. the `DemoSensorC.nc` implementation for telosb will be different than the `DemoSensorC.nc` implementation for micaz. This is the answer to the first question asked in the **introduction** section: the *platform dependent* `DemoSensorC` component defines which sensor the `Sense` or `Oscilloscope` application is sampling and every platform that wants to run sensing applications such as `Oscilloscope`, `Sense` or `RadioSenseToLeds` has to provide its own version of `DemoSensorC`. Additionally, sensor boards may come with their own version of `DemoSensorC` (e.g., the `basicsb` sensorboard for the mica-family of motes define `DemoSensorC.nc` to be that board's light sensor).

The DemoSensorC component

Let's take a closer look at the `DemoSensorC` component. Every `DemoSensorC` component has the following signature:

```
generic configuration DemoSensorC()
```

```
{
  provides interface Read<uint16_t>;
}
```

In its implementation section, however, `DemoSensorC` may differ from platform to platform. For example, on the *telosb* platform `DemoSensorC` instantiates a component called `VoltageC`, which reads data from the MCU-internal voltage sensor. Because the *micaz* doesn't have any built-in sensors its `DemoSensorC` uses system library component like `ConstantSensorC` or `SineSensorC`, which return "fake" sensor data. Thus `DemoSensorC` is a means of indirecting sensor data acquisition from a platform-specific sensor component (like `VoltageC`) to platform-independent applications like `Sense` or `Oscilloscope`. Usually the configuration of a sensor is done in the component that `DemoSensorC` instantiates.

How can `Sense` be changed to sample a sensor other than the platform's default sensor? Usually this requires changing only a single line of code in `DemoSensorC`; for example, if you wanted to replace the `VoltageC` component on *telosb* by the constant sensor component `ConstantSensorC` you could change the follwoing line in `DemoSensorC` from:

```
components new VoltageC() as DemoSensor;
```

to something like

```
components new ConstantSensorC(uint16_t, 0xbeef) as
DemoSensor;
```

What sensors are available depends on the platform. Sensor components are usually located in the respective platform subdirectory (`tinyos-2.x/tos/platforms`), in the respective sensorboard subdirectory (`tinyos-2.x/tos/sensorboards`) or, in case of microprocessor-internal sensors, in the respective chips subdirectory (`tinyos-2.x/tos/chips`). `ConstantSensorC` and `SineSensorC` can be found in `tinyos-2.x/tos/system`.

Running the Sense application

To compile the `Sense` application, go to the `apps/Sense` directory and depending on which hardware you have, type something similar to `make telosb install`. If you get errors such as the following,

```
SenseAppC.nc:50: component DemoSensorC not found
SenseAppC.nc:50: component `DemoSensorC' is not
generic
SenseAppC.nc:55: no match
```

your platform has not yet implemented the `DemoSensorC` component. For a quick solution you can copy `DemoSensorC.nc` from `tinyos-2.x/tos/platforms/micaz` to your platform directory (a good starting point on how to create sensor components is probably **TEP 101** and **TEP 114**).

If you have a mica-family mote and a "basic" (mda100) sensor board, you can get a more interesting test by compiling with

```
SENSORBOARD=basicsb make platform install
```

to run `Sense` using the mda100's light sensor.

Once you have installed the application the three least significant bits of the sensor readings are displayed on the node's LEDs (0 = off, 1 = on). It is the least significant bits, because `Sense` cannot know the precision (value range) of the returned sensor readings and, for example, the three most significant bits in a `uint16_t` sensor reading sampled through a 12-bit ADC would be meaningless (unless the value was left-shifted). If your `DemoSensorC` represents a sensor whose readings are fluctuating you may see the LEDs toggle, otherwise `Sense` is not very impressive. Let's take a look at a more interesting application: `Oscilloscope`.

The Oscilloscope application

`Oscilloscope` is an application that let's you visualize sensor readings on the PC. Every node that has `Oscilloscope` installed periodically samples the default sensor (**via `DemoSensorC`**) and broadcasts a message with 10 accumulated readings over the radio. A node running the `BaseStation` application will forward these messages to the PC using the serial communication. To run `Oscilloscope` you therefore need at least two nodes: one node attached to your PC running the `BaseStation` application (`BaseStation` can be found at `tinyos-2.x/apps/BaseStation` and was introduced in the **previous lesson**) and one or more nodes running the `Oscilloscope` application.

Let's take a look at the **`OscilloscopeAppC.nc`** configuration:

```
configuration OscilloscopeAppC
{
}
implementation
{
  components OscilloscopeC, MainC, ActiveMessageC,
LedsC,
    new TimerMilliC(), new DemoSensorC() as Sensor,
    new AMSenderC(AM_OSCILLOSCOPE), new
AMReceiverC(AM_OSCILLOSCOPE);

  OscilloscopeC.Boot -> MainC;
  OscilloscopeC.RadioControl -> ActiveMessageC;
  OscilloscopeC.AMSend -> AMSenderC;
  OscilloscopeC.Receive -> AMReceiverC;
  OscilloscopeC.Timer -> TimerMilliC;
```

```
  OscilloscopeC.Read -> Sensor;
  OscilloscopeC.Leds -> LedsC;
}
```

The actual implementation of the application is in **OscilloscopeC.nc**. This is the signature of `OscilloscopeC.nc`:

```
module OscilloscopeC
{
  uses {
    interface Boot;
    interface SplitControl as RadioControl;
    interface AMSend;
    interface Receive;
    interface Timer;
    interface Read;
    interface Leds;
  }
}
```

`Oscilloscope` is a combination of different building blocks introduced in previous parts of the tutorial. Like **Sense**, `Oscilloscope` uses `DemoSensorC` and a timer to periodically sample the default sensor of a platform. When it has gathered 10 sensor readings `OscilloscopeC` puts them into a message and broadcasts that message via the `AMSend` interface. `OscilloscopeC` uses the `Receive` interface for synchronization purposes (see below) and the `SplitControl` interface, to switch the radio on. If you want to know more about mote-mote radio communication read **lesson 3**.

Running the Oscilloscope application

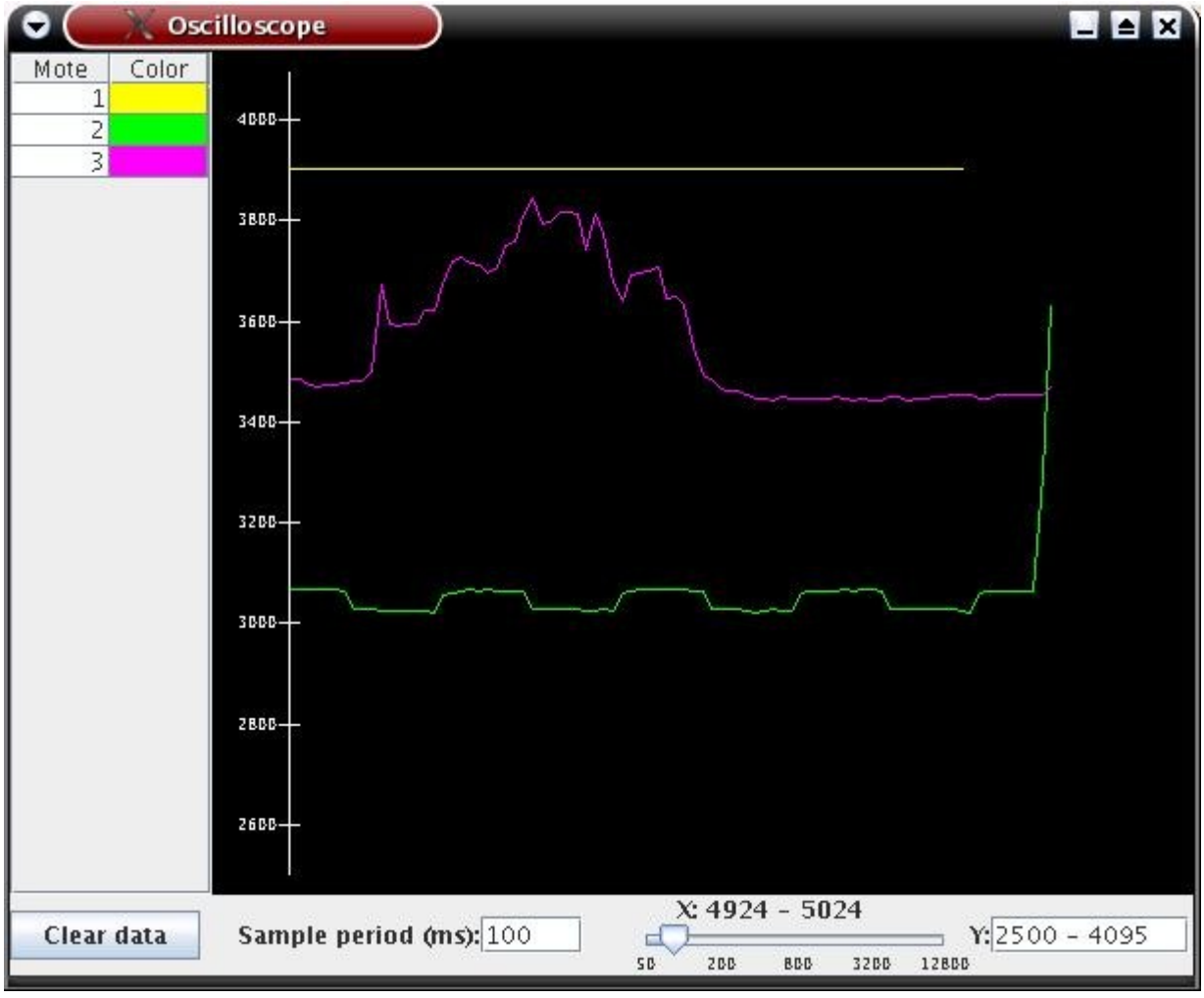
To install the `Oscilloscope` application go to `tinyos-2.x/apps/Oscilloscope` and depending on which hardware you have, type something similar to `make telosb install,1`. Note the "`,1`" after the `install` option, which assigns ID 1 to the node. Assigning IDs to nodes is helpful to differentiate them later on in the GUI, so make sure you assign different IDs to all nodes on which `Oscilloscope` is installed (e.g. install `Oscilloscope` on a second node with `make telosb install,2` and so on). A node running `Oscilloscope` will toggle its second LED for every message it has sent and it will toggle its third LED when it has received an `Oscilloscope` message from another node: incoming messages are used for sequence number synchronization to let nodes catch up when they are switched on later than the others; they are also used for changing the sample rate that defines how often sensor values are read. In case of a problem with the radio connection the first LED will toggle.

Install `BaseStation` on another node and connect it to your PC. As usual, on the `BaseStation` node you should see the second LED toggle for every message bridged from radio to serial.

Running the Java GUI

To visualize the sensor readings on your PC first go to `tinyos-2.x/apps/Oscilloscope/java` and type `make`. This creates/compiles the necessary message classes and the `Oscilloscope` Java GUI. Now start a SerialForwarder and make sure it connects to the node on which you have installed the `BaseStation` application (how this is done is explained in the **previous lesson**). In case you have problems with the Java compilation or the serial connection work through the **previous lesson**.

Once you have a SerialForwarder running you can start the GUI by typing `./run` (in `tinyos-2.x/apps/Oscilloscope/java`). You should see a window similar to the one below:



Each node is represented by a line of different color (you can change the color by clicking on it in the mote table). The x-axis is the packet counter number and the y-axis is the sensor reading. To change the sample rate edit the number in the "sample rate" input box. When you press enter, a message containing the new rate is created and broadcast via the `BaseStation` node to all

nodes in the network. You can clear all received data on the graphical display by clicking on the "clear data" button.

The `Oscilloscope` (or `Sense`) application displays the raw data as signalled by the `Read.readDone()` event. How the values are to be interpreted is out of scope of the application, but the GUI let's you adapt the visible portion of the y-axis to a plausible range (at the bottom right).

Related Documentation

- **nesC reference manual**
- **TEP 101: ADC**
- **TEP 114: SIDs: Source and Sink Independent Drivers**
- **TinyOS Programming Guide**

Lesson 6: TinyOS Boot and System Initialization

Last updated 30 October 2006

Introduction

One of the frequently asked questions regarding TinyOS is, "Where is `main()`?". In previous lessons, we deferred detailed discussion of the TinyOS boot sequence: applications handle the `Boot.booted` event and start from there. This tutorial describes the steps that occur before and after this event, showing how to properly initialize components.

Boot Sequence

The TinyOS boot sequence has four steps:

1. Scheduler initialization
2. Component initialization
3. Signal that the boot process has completed
4. Run the scheduler

The application-level boot sequence component is MainC (in tos/system). MainC provides one interface, Boot and uses one interface, Init as SoftwareInit. The boot sequence calls SoftwareInit.init() as part of step 2 and signals Boot.booted in step 3.

The default real boot sequence is in the component RealMainP. Note that its name ends in P, denoting that components should not directly wire to it. This is RealMainP's signature:

```
module RealMainP {
  provides interface Booted;
  uses {
    interface Scheduler;
    interface Init as PlatformInit;
    interface Init as SoftwareInit;
  }
}
```

MainC only provides Boot and uses SoftwareInit; RealMainP uses two additional interfaces, PlatformInit and Scheduler. MainC hides these from applications by automatically wiring them to the system's scheduler and platform initialization sequence. The difference between PlatformInit and SoftwareInit is predominantly one of hardware vs. software. PlatformInit is responsible for placing core platform services into meaningful states; for example, the PlatformInit of mica platforms calibrates their clocks.

This is the code of RealMainP:

```
implementation {
  int main() __attribute__ ((C, spontaneous)) {
    atomic {
      call Scheduler.init();
      call PlatformInit.init();
      while (call Scheduler.runNextTask());
      call SoftwareInit.init();
      while (call Scheduler.runNextTask());
    }
    __nesc_enable_interrupt();
    signal Boot.booted();
    call Scheduler.taskLoop();
    return -1;
  }
  default command error_t PlatformInit.init()
{ return SUCCESS; }
  default command error_t SoftwareInit.init()
{ return SUCCESS; }
```

```
    default event void Boot.booted() { }
  }
```

The code shows the four steps described above.

Scheduler Initialization

The first boot step is to initialize the scheduler. If the scheduler were not initialized before the components, component initialization routines would not be able to post tasks. While not all components require tasks to be posted, this gives the flexibility required for those components that do. The boot sequence runs tasks after each initialization stage in order to allow long-running operations, since they only happen once. **TEP 106** describes TinyOS schedulers in greater detail, including information on how to replace the scheduler.

Component initialization.

After RealMainP initializes the scheduler, it initializes the platform. The `Init` interface implements only the single command `init()`.

tos/interfaces/Init.nc:

```
interface Init {
  command error_t init();
}
```

The platform initialization phase is the responsability of the platform implementer. Thus, `PlatformInit` is wired to the platform-specific initialization component, `PlatformC`. No other component should be wired to `PlatformInit`. Any component that requires initialization can implement the `Init` interface and wire itself to `MainC`'s `SoftwareInit` interface:

tos/system/MainC.nc:

```
configuration MainC {
  provides interface Boot;
  uses interface Init as SoftwareInit;
}
implementation {
  components PlatformC, RealMainP, TinySchedulerC;

  RealMainP.Scheduler -> TinySchedulerC;
  RealMainP.PlatformInit -> PlatformC;
```

```
  // Export the SoftwareInit and Booted for
applications
  SoftwareInit = RealMainP.SoftwareInit;
  Boot = RealMainP;
}
```

A common issue in initialization code is dependencies between different parts of the system. These are handled in three ways in TinyOS:

- Hardware-specific initialization issues are handled directly by each platform's `PlatformC` component.
- System services (e.g., the timer, the radio) are typically written to be independently initializable. For instance, a radio that uses a timer does not setup the timer at radio initialisation time, rather it defers that action until the radio is started. In other words, initialisation is used to setup software state, and hardware state wholly owned by the service.
- When a service is split into several components, the `Init` interface for one of these components may well call `Init` (and other) interfaces of the other components forming the service, if a specific order is needed.

Signal that the boot process has completed.

Once all initialization has completed, `MainC`'s `Boot.booted()` event is signaled. Components are now free to call `start()` and other commands on any components they are using. Recall that in the `Blink` application, the timers were started from the `booted()` event. This `booted` event is TinyOS's analogue of `main` in a Unix application.

Run the scheduler loop.

Once the application has been informed that the system as booted and started needed services, TinyOS enters its core scheduling loop. The scheduler runs as long as there are tasks on the queue. As soon as it detects an empty queue, the scheduler puts the microcontroller into the lowest power state allowed by the active hardware resources. For example, only having timers running usually allows a lower power state than peripheral buses like the UART. **TEP 112** describes in detail how this process works.

The processor goes to sleep until it handles an interrupt. When an interrupt arrives, the MCU exits its sleep state and runs the interrupt handler. This causes the scheduler loop to restart. If the interrupt handler posted one or more tasks, the scheduler runs tasks until the task queue and then returns to sleep.

Boot and SoftwareInit

From the perspective of an application or high-level services, the two important interfaces in the boot sequence are those which MainC exports: Boot and SoftwareInit. Boot is typically only handled by the top-level application: it starts services like timers or the radio. SoftwareInit, in contrast, touches many difference parts of the system. If a component needs code that runs once to initialize its state or configuration, then it can wire to SoftwareInit.

Typically, service components that require intialization wire themselves to SoftwareInit rather than depend on the application writer to do so. When an application developer is writing a large, complex system, keeping track of all of the initialization routines can be difficult, and debugging when one is not being called can be very difficult. To prevent bugs and simplify application development, services typically use *auto-wiring*.

The term auto-wiring refers to when a component automatically wires its dependencies rather than export them for the application writer to resolve. In this case, rather than provide the Init interface, a service component wires its Init interface to RealMainC. For example, PoolC is a generic memory pool abstraction that allows you to declare a collection of memory objects for dynamic allocation. Underneath, its implementation (PoolP) needs to initialize its data structures. Given that this must happen for the component to operate properly, an application writer shouldn't have to worry about it. So the PoolC component instantiates a PoolP and wires it to MainC.SoftwareInit:

```
generic configuration PoolC(typedef pool_t, uint8_t
POOL_SIZE) {
  provides interface Pool;
}

implementation {
  components MainC, new PoolP(pool_t, POOL_SIZE);

  MainC.SoftwareInit -> PoolP;
  Pool = PoolP;
}
```

In practice, this means that when MainP calls SoftwareInit.init, it calls Init.init on a large number of components. In a typical large application, the initialization sequence might involve as many as thirty components. But the application developer doesn't have to worry about this: properly written components take care of it automatically.

Related Documentation

- **TEP 106: Schedulers and Tasks**
- **TEP 107: Boot Sequence**

Programming Hint 8: In the top-level configuration of a software abstraction, auto-wire Init to MainC. This removes the burden of wiring Init from the programmer, which removes unnecessary work from the boot sequence and removes the possibility of bugs from forgetting to wire. From *TinyOS Programming*

Lesson 7: Permanent Data Storage

Last Modified: April 13, 2007

This lesson introduces permanent (non-volatile) data storage in TinyOS. Permanent storage allows a node to persist data even if power is disconnected or the node is reprogrammed with a new image. You will become familar with different kinds of data storage including small objects, logs, and large objects. You will be exposed to the TinyOS interfaces and components that support permanent data storage on motes and you will learn how to:

- Divide the flash chip into volumes, which allows multiple and/or different type of data to be stored.
- Store configuration data that survives a power cycle.
- Store packets using the logging abstraction and retransmit the overheard packets after a power cycle.

Introduction

TinyOS 2.x provides three basic storage abstractions: small objects, circular logs, and large objects. TinyOS 2.x also provides *interfaces* to abstract the underlying storage services and *components* that *provide* (implement) these interfaces.

Interfaces

Let's take a look at some of the interfaces that are in the `tos/interfaces` directory and the types defined in the `tos/types` to familiarize ourselves with the general functionality of the storage system:

- `BlockRead`
- `BlockWrite`
- `Mount`
- `ConfigStorage`
- `LogRead`
- `LogWrite`
- `Storage.h`

Components

Components provide concrete implementations of the interfaces. You should be familiar with these components because your code needs to specify both the *interfaces* your application *uses* as well as the *components* which *provide* (implement) these interfaces:

- **`ConfigStorageC`**
- **`LogStorageC`**
- **`BlockStorageC`**

Implementations

The preceding components are actually *chip-specific implementations* of these abstractions. Since TinyOS supports multiple platforms, each of which might have its own implementation of the storage drivers, you may need to be aware of platform-specific constants or other details (e.g. erase size) that can couple a storage client to the underlying chip-specific implementation.

For example, the preceding links are all specific to the ST Microelectronics M25Pxx family of flash memories used in the Telos and Tmote Sky motes. You *do not* need to worry about the details of where these files reside because TinyOS's make system includes the correct drivers automatically. However, you *do* need to know what these components are called because your code must list them in a `components` declaration.

If you are curious, the following links will let you browse the implementations of the Atmel AT45DB family of flash memories used in the Mica2/MicaZ motes:

- **`ConfigStorageC`**
- **`LogStorageC`**
- **`BlockStorageC`**

Finally, the following links will let you browse the implementation for the Intel imote2 flash memory:

- **`ConfigStorageC`**
- **`LogStorageC`**
- **`BlockStorageC`**

Volumes

TinyOS 2.x divides a flash chip into one or more fixed-sized *volume*s that are specified at compile-time using an XML file. This file, called the volume table, allows the application developer to specify the name, size, and (optionally) the base address of each volume in the flash. Each volume provides a single type of storage abstraction (e.g. configuration, log, or block storage). The abstraction type defines the physical layout of data on the flash memory. A volume table might look like:

```
<volume_table>
  <volume name="CONFIGLOG" size="65536"/>
  <volume name="PACKETLOG" size="65536"/>
  <volume name="SENSORLOG" size="131072"/>
  <volume name="CAMERALOG" size="524288"/>
</volume_table>
```

The volume table for a particular application must be placed in the application's directory (where one types 'make') and must be named `volumes-CHIPNAME.xml` where CHIPNAME is replaced with the platform-specific flash chip's name. For example, the Telos mote uses the ST Microelectronics M25P family of flash memories. The drivers for these chips can be found in the `tos/chips/stm25p` directory. Therefore, a Telos-based application that uses the storage abstractions needs a file named `volumes-stm25p.xml`.

Note that the size parameter is a multiple of the erase unit for a particular flash chip. See Section 4.1 in **TEP 103** for more details.

Storing Configuration Data

This lesson shows how configuration data can be written to and read from non-volatile storage. Configuration data typically exhibit some subset of the following properties. They are **limited in size**, ranging from a fews tens to a couple hundred bytes. Their values may be **non-uniform** across nodes. Sometimes, their values are **unknown** prior to deployment in the field and sometimes their values are **hardware-specific**, rather than being tied to the software running on a node.

Because configuration data can be non-uniform across nodes or unknown *a priori*, their values may be difficult to specify at compile-time and since the data are sometimes hardware-specific, their values must survive reprogramming, suggesting that encoding these values in the program image is not the simplest approach. Storing configuration data in volatile memory is also problematic since this data would not survive a reset or power cycle.

In summary, configuration data must persist through node resets, power cycles, or reprogramming, and then be restored afterward. The ability to persist and restore configuration data in this manner is useful in many scenarios.

- **Calibration.** Calibration coefficients for sensors might be factory-configured and persisted, so they are not lost when power is removed for shipping or the node is reprogrammed post-calibration. For example, a hypothetical temperature sensor might have an offset and gain that must be calibrated, because these parameters are hardware-specific, and stored because they are needed to convert the output voltage into the more useful units of degrees Celcius. The calibration data for such a sensor might look like:
  - `typedef struct calibration_config_t {`

```
    int16_t temp_offset;
    int16_t temp_gain;
  } calibration_config_t;
```

- **Identification.** Device identification information, like IEEE-compliant MAC addresses or the TinyOS TOS_NODE_ID parameters are non-uniform across nodes although they are not hardware-specific, once they are assigned to a node, these values should be *sticky* in that they are persisted across reset, power cycle, and reprogramming operations (and not lost or reassigned to another node).

```
  typedef struct radio_config_t {
    ieee_mac_addr_t mac;
    uint16_t tos_node_id;
  } radio_config_t;
```

- **Location.** Node location data may be unknown at compile-time and only become available during deployment. An application might, for example, store node coordinates as follows and update these values in the field:

```
  typedef struct coord_config_t {
    uint16_t x;
    uint16_t y;
    uint16_t z;
  } coord_config_t;
```

- **Sensing.** Sensing and signal processing parameters like sample period, filter coefficients, and detection thresholds might be adjusted in the field. The configuration data for such an application might look like:

```
  typedef struct sense_config_t {
    uint16_t temp_sample_period_milli;
    uint16_t temp_ema_alpha_numerator;
    uint16_t temp_ema_alpha_denominator;
    uint16_t temp_high_threshold;
    uint16_t temp_low_threshold;
  } sense_config_t;
```

Now that we have discussed *why* one might use this type of storage, let's see *how* to use it. We will implement a simple demo application that illustrates how to use the `Mount` and `ConfigStorage` abstractions. A timer period will be read from flash, divided by two, and written back to flash. An LED is toggled each time the timer fires. But, before diving into code, let's discuss some high-level design considerations.

See **`tinyos-2.x/apps/tutorials/BlinkConfig/`** for the accompanying code.

Prior to its first usage, a volume does not contain any valid data. So, our code should detect the first usage of a volume and take any appropriate actions (e.g. preload it with default values). Similarly, when the data layout of the volume changes (for example, if the application requires new or different configuration variables), then application code should detect this and take

appropriate actions (e.g. migrate the old data to the new layout or erase the volume and reload the defaults). These requirements suggest that we should have a way of keeping track of the volume version. We will use a version number for this purpose (and will need to maintain a discipline of updating the version number when the data layout changes incompatibly). Our configuration struct might have the following fields for the version number and blink period:

```
typedef struct config_t {
  uint16_t version;
  uint16_t period;
} config_t;
```

1. Create a `volumes-CHIPNAME.xml` file, enter the volume table in this file, and place the file in the application directory. Note that `CHIPNAME` is the flash chip used on your target plaform. For example, `CHIPNAME` will be `stm25p` for the Telos platform and `at45db` for the MicaZ platform. Our file will have the following contents:

```
<volume_table>
  <volume name="LOGTEST" size="262144"/>
  <volume name="CONFIGTEST" size="131072"/>
</volume_table>
```

   This volume information is used by the toolchain to create an include file. The auto-generated file, however, has to be included manually. Place the following line in the configuration file which declares the ConfigStorageC component (e.g. `BlinkConfigAppC.nc`):

```
#include "StorageVolumes.h"
```

6. BlinkConfigC, the application code for this simple demo, *uses* the `Mount` and `ConfigStorage` interfaces (note that we rename `ConfigStorage` to `Config`).

```
module BlinkConfigC {
  uses {
    ...
    interface ConfigStorage as Config;
    interface Mount;
    ...
  }
}
```

15. Each interface must be wired to an *implementation* that will provide it:

```
configuration BlinkConfigAppC {
}
implementation {
  components BlinkConfigC as App;
  components new
  ConfigStorageC(VOLUME_CONFIGTEST);
```

```
  ...

  App.Config     -> ConfigStorageC.ConfigStorage;
  App.Mount      -> ConfigStorageC.Mount;
  ...
}
```

Before the flash chip can be used, it must be mounted using the two-phase mount/mountDone command. Here we show how this might be chained into the boot sequence:

```
  event void Boot.booted() {
    conf.period = DEFAULT_PERIOD;

    if (call Mount.mount() != SUCCESS) {
      // Handle failure
    }
  }
```

If the Mount.mount succeeds, then the `Mount.mountDone` event will be signaled. The following code shows how to check if the volume is valid, and if it is, how to initiate a read from the volume using the `ConfigStore.read` command. If the volume is invalid, calling `Config.commit` will make it valid (this call is also used to flush buffered data to flash much like the UNIX fsync system call is supposed to flush buffered writes to disk):

```
  event void Mount.mountDone(error_t error) {
    if (error == SUCCESS) {
      if (call Config.valid() == TRUE) {
        if (call Config.read(CONFIG_ADDR, &conf,
  sizeof(conf)) != SUCCESS) {
          // Handle failure
        }
      }
      else {
        // Invalid volume.  Commit to make valid.
        call Leds.led1On();
        if (call Config.commit() == SUCCESS) {
          call Leds.led0On();
        }
        else {
          // Handle failure
        }
      }
    }
    else{
      // Handle failure
    }
```

```
  }
```

58. If the read is successful, then a `Config.readDone` event will occur. In this case, we first check for a successful read, and if successful, we then check the version number. If the version number matches what we expected, we copy of the configuration data to a local variable, and adjust its values. If there is a version mismatch, we set the value of the configuration information to a default value. Finally, we call the the `Config.write` function:

```
  event void Config.readDone(storage_addr_t addr,
  void* buf,
    storage_len_t len, error_t err)
  __attribute__((noinline)) {

    if (err == SUCCESS) {
      memcpy(&conf, buf, len);
      if (conf.version == CONFIG_VERSION) {
        conf.period = conf.period/2;
        conf.period = conf.period > MAX_PERIOD ?
  MAX_PERIOD : conf.period;
        conf.period = conf.period < MIN_PERIOD ?
  MAX_PERIOD : conf.period;
      }
      else {
        // Version mismatch. Restore default.
        call Leds.led1On();
        conf.version = CONFIG_VERSION;
        conf.period = DEFAULT_PERIOD;
      }
      call Leds.led0On();
      call Config.write(CONFIG_ADDR, &conf,
  sizeof(conf));
    }
    else {
      // Handle failure.
    }
  }

```

83. Data is not necessarily "written" to flash when `ConfigStore.write` is called and `Config.writeDone` is signaled. To ensure data is persisted to flash, a `ConfigStore.commit` call is required:

```
  event void Config.writeDone(storage_addr_t
  addr, void *buf,
    storage_len_t len, error_t err) {
    // Verify addr and len

```

```
    if (err == SUCCESS) {
      if (call Config.commit() != SUCCESS) {
        // Handle failure
      }
    }
    else {
      // Handle failure
    }
  }
```

97. Finally, when the `Config.commitDone` event is signaled, data has been durably written to flash and will survive a node power cycle:

```
  event void Config.commitDone(error_t err) {
    call Leds.led0Off();
        call Timer0.startPeriodic(conf.period);
        if (err == SUCCESS) {
          // Handle failure
        }
     }
```

Logging Data

Reliable (atomic) logging of events and small data items is a common application requirement. Logged data should not be lost if a system crashes. Logs can be either linear (stop logging when the volume is full) or circular (overwrite the least recently written data when the volume is full).

The TinyOS LogStorage abstraction supports these requirements. The log is record based: each call to LogWrite.append (see below) creates a new record. On failure (a crash or power cycle), the log only loses whole records from the end of the log. Additionally, once a circular log wraps around, log writes only lose whole records from the beginning of the log.

A demo application called `PacketParrot` shows how to use the `LogWrite` and `LogRead` abstractions. A node writes received packets to a circular log and retransmits the logged packets (or at least the parts of the packets above the AM layer) when power is cycled.

See **`tinyos-2.x/apps/tutorials/PacketParrot/`** for the accompanying code.

The application logs packets it receives from the radio to flash. On a subsequent power cycle, the application transmits any logged packets, erases the log, and then continues to log packets again. The red LED is on when the log is being erased. The blue (yellow) LED turns on when a packet is received and turns off when a packet has been logged successfully. The blue (yellow) LED remains on when packets are being received but are not logged (because the log is being erased). The green LED flickers rapidly after a power cycle when logged packets are transmitted.

1. The first step when using the log is to decide what kind of data you want to store in the log. In this case, we will declare a struct of the type:

```
2.   typedef nx_struct logentry_t {
3.     nx_uint8_t len;
4.     message_t msg;
5.   } logentry_t;
```

6. Unlike Config storage, Log storage does not require the volume to be explicitly mounted by the application. Instead, a simple read suffices in which a buffer and the number of bytes to read are passed to `LogRead.read`:

```
7.   event void AMControl.startDone(error_t err) {
8.     if (err == SUCCESS) {
9.       if (call LogRead.read(&m_entry,
   sizeof(logentry_t)) != SUCCESS) {
10.        // Handle error
11.      }
12.    }
13.    else {
14.      call AMControl.start();
15.    }
16.  }
```

17. If the call to `LogRead.read` returns SUCCESS, then a `LogRead.readDone` event will be signaled shortly thereafter. When that happens, we check if the data that was returned is the same length as what we expected. If it is, we use the data but if not, we assume that either the log is empty or that we have lost synchronization, so the log is erased:

```
18.
19.  event void LogRead.readDone(void* buf,
   storage_len_t len, error_t err) {
20.    if ( (len == sizeof(logentry_t)) && (buf ==
   &m_entry) ) {
21.      call Send.send(&m_entry.msg, m_entry.len);
22.      call Leds.led1On();
23.    }
24.    else {
25.      if (call LogWrite.erase() != SUCCESS) {
26.        // Handle error.
27.      }
28.      call Leds.led0On();
29.    }
30.  }
31.
```

32. The `PacketParrot` application stores packets received over the radio to flash by first saving the `message_t` and its length to a `log_entry_t` struct and then calling `LogWrite.append`:

```
33.   event message_t* Receive.receive(message_t*
  msg, void* payload, uint8_t len){
34.     call Leds.led2On();
35.     if (!m_busy) {
36.       m_busy = TRUE;
37.       m_entry.len = len;
38.       m_entry.msg = *msg;
39.       if (call LogWrite.append(&m_entry,
  sizeof(logentry_t)) != SUCCESS) {
40.         m_busy = FALSE;
41.       }
42.     }
43.     return msg;
44.   }
```

45. If the `LogWrite.write` returned SUCCESS, then a short time later, a `LogWrite.appendDone` will be signaled. This event returns the details of the write including the source buffer, length of data written, whether any records were lost (if this is a circular buffer) and any error code. If no errors occurred, then the data was written to flash with atomicity, consistency, and durability guarantees (and will survive node crashes and reboots):

```
46.   event void LogWrite.appendDone(void* buf,
  storage_len_t len,
47.                                   bool
  recordsLost, error_t err) {
48.     m_busy = FALSE;
49.     call Leds.led2Off();
50.   }
51.
```

Storing Large Objects

Block storage is generally used for storing large objects that cannot easily fit in RAM. Block is a low-level system interface that requires care when using since it is essentially a write-once model of storage. Rewriting requires an erase which is time-consuming, occurs at large granularity (e.g. 256 B to 64 KB), and can only happen a limited number of times (e.g. 10,000 to 100,000 times is typical). The TinyOS network reprogramming system uses Block storage to store program images.

See **`tinyos-2.x/apps/tests/storage/Block/`** for an example of code that uses the Block storage abstraction.

Conclusions

This lesson introduced the basic storage abstractions in Tiny 2.x.

Related Documentation

- (1) **TEP 103: Permanent Data Storage**
- (2) **Lesson 1: Getting Started with TinyOS and nesC** *TinyOS Programming*

Lesson 8: Resource Arbitration and Power Management

Last updated 30 October 2006

Introduction

TinyOS distinguishes between three kinds of resource abstractions: **dedicated**, **virtualized**, and **shared**. Two fundamental questions must be asked about each type of abstraction.

1. How can a client gain access to the resource provided through this abstraction?
2. How can the power state of that resource be controlled?

Components offer resource sharing mechanisms and power mangement capabilites according to the goals and level of abstraction required by their clients.

An abstraction is dedicated if it represents a resource which a subsystem needs exclusive access to at all times. In this class of resources, no sharing policy is needed since only a single component ever requires use of the resource. Resource clients simply call commands from the interfaces provided by the resource just as they would with any other TinyOS component. Resources of this type provide either an AsyncStdControl, StdControl, or SplitControl interface for controlling their power states. The definition of each of these interfaces can be found in tinyos-2.x/tos/interfaces.

```
interface AsyncStdControl {
  async command error_t start();
  async command error_t stop();
}
interface StdControl {
  command error_t start();
  command error_t stop();
```

```
}
interface SplitControl {
  async command error_t start();
  async command void startDone(error_t error);
  async command error_t stop();
  async command void stopDone(error_t error);
}
```

Currently, the power states of all dedicated resources are controlled by one of these three interfaces. They are only allowed to enter one of two logical power states (on/off), regardless of the number of physical power states provided by the hardware on top of which their resource abstraction has been built. Which of these interfaces is provided by a particular resource depends on the timing requirements for physically powering it on or off.

Virtual abstractions hide multiple clients from each other through software virtualization. Every client of a virtualized resource interacts with it as if it were a dedicated resource, with all virtualized instances being multiplexed on top of a single underlying resource. Because the virtualization is done in software, there is no upper bound on the number of clients using the abstraction, barring memory or efficiency constraints. The power states of a virtualized resource are handled automatically, and no interface is provided to the user for explicity controlling its power state. As they are built on top of shared resources, the reason their power states can be automatically controlled will become clearer after reading the following section.

Dedicated abstractions are useful when a resource is always controlled by a single component. Virtualized abstractions are useful when clients are willing to pay a bit of overhead and sacrifice control in order to share a resource in a simple way. There are situations, however, when many clients need precise control of a resource. Clearly, they can't all have such control at the same time: some degree of multiplexing is needed.

A motivating example of a shared resource is a bus. The bus may have multiple peripherals on it, corresponding to different subsystems. For example, on the Telos platform the flash chip (storage) and the radio (network) share a bus. The storage and network stacks need exclusive access to the bus when using it, but they also need to share it with the other subsystem. In this case, virtualization is problematic, as the radio stack needs to be able to perform a series of operations in quick succession without having to reacquire the bus in each case. Having the bus be a shared resource allows the radio stack to send a series of operations to the radio atomically, without having to buffer them all up in memory beforehand (introducing memory pressure in the process).

In TinyOS, a resource **arbiter** is responsible for multiplexing between the different clients of a shared resource. It determines which client has access to the resource at which time. While a client

holds a resource, it has complete and unfettered control. Arbiters assume that clients are cooperative, only acquiring the resource when needed and holding on to it no longer than necessary. Clients explicitly release resources: there is no way for an arbiter to forcibly reclaim it.

Shared resources are essentially built on top of dedicated resources, with access to them being controlled by an arbiter component. In this way, **power managers** can be used to automatically control the power state of these resources through their AsyncStdControl, StdControl, or SplitControl interfaces. They communicate with the arbiter (through the use of a ResourceDefaultOwner interface), monitoring whether the resource is being used by any of its clients and powering it on/off accordingly. The figure below shows how an arbiter component and a power manager can be wired together to provide arbitration and automatic power management for a shared resource.

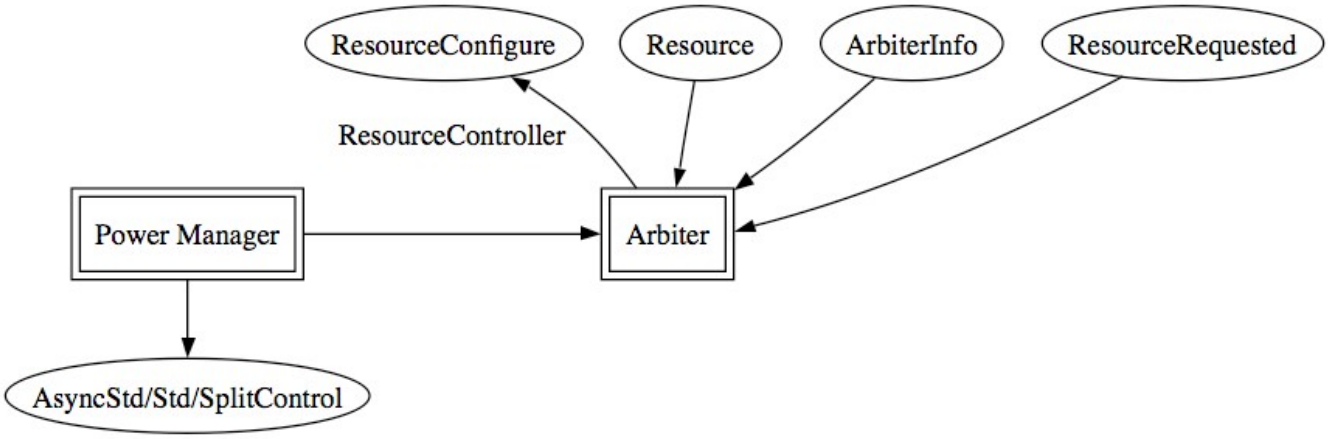


Figure 1: Arbiters and Power Managers

The arbiter component provides the Resource, ArbiterInfo, ResourceRequested, and ResourceDefaultOwner interfaces and uses the ResourceConfigure interface. The power manager doesn't provide any interfaces, but uses one of either the AsyncStdControl, StdControl, or SplitControl interfaces from the underlying resource, as well as the ResourceDefaultOwner interface provided by the arbiter. The figure below shows how these interface are then wired together with the implementation of a shared resource. Please refer to TEP 108 for more information on arbiters and TEP 115 for more information on Power Managers.

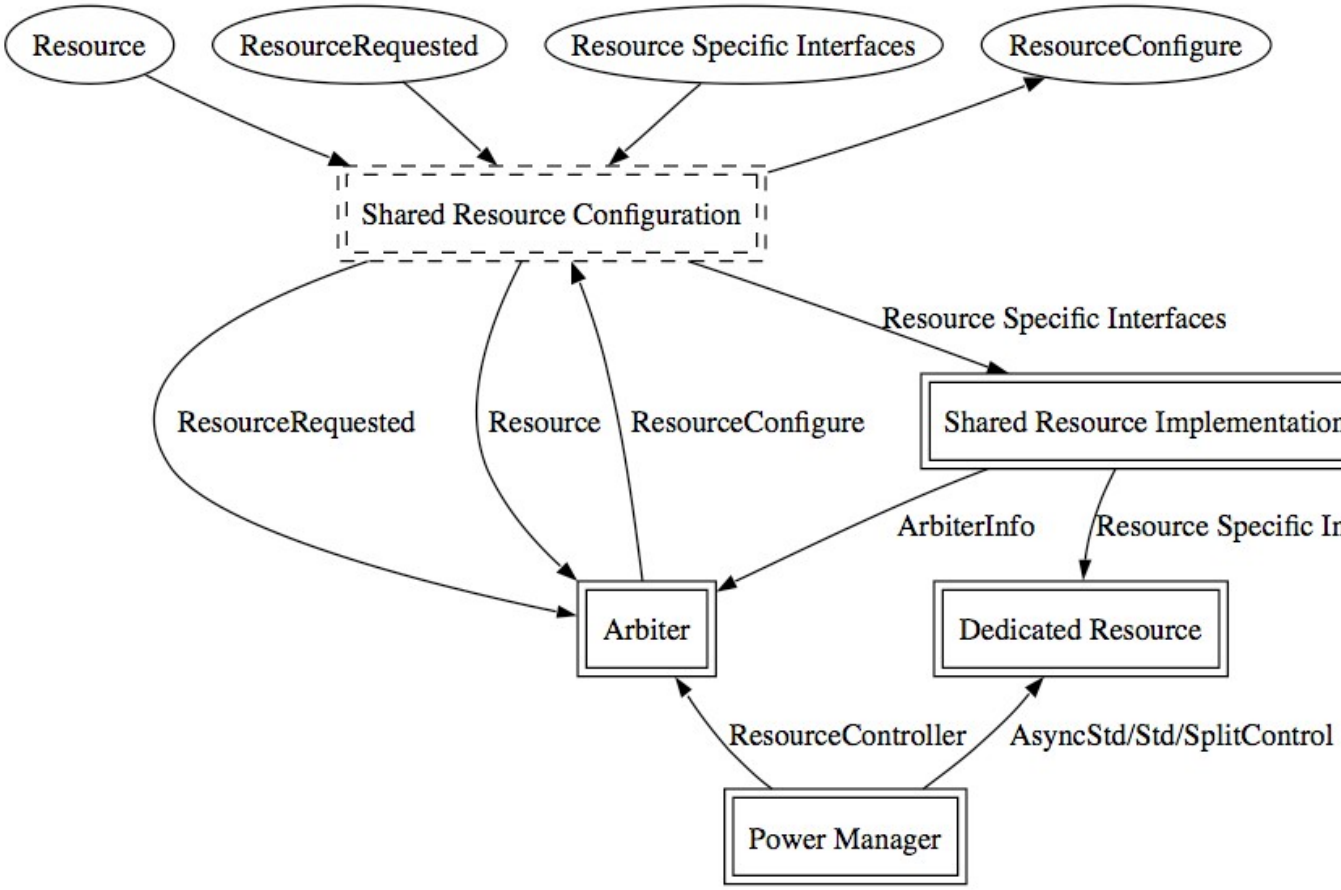


Figure 2: Shared Resource Configuration

From this figure, we see that the only interfaces exposed to a client through the shared resource abstraction are the `Resource` and `ResourceRequested` interfaces provided by the arbiter as well as any resource specific interfaces provided by the resource itself. It also uses a `ResourceConfigure` interface, expecting it to be implemented on a client by client basis depending on their requirements. A client requests access to a shared resource through the `Resource` interface and runs operations on it using whatever resource specific interfaces are provided. A client may choose to wire itself to the `ResourceRequested` interface if it wishes to hold onto a resource indefinitely and be informed whenever other clients request its use.

The rest of this tutorial is dedicated to teaching users how to use shared resources and show them how wiring is done between all components that make them up.

Specifically, this tutorial will teach users how to:

1. Wire in a shared resource for use by a client.
2. Use the `Resource` interface to gain access to a shared resource.
3. Change the arbitration policy used by a particular shared resource.
4. Wire up a power manager for use by a shared resource.

Working with Shared Resources

This section shows you how to gain access to and use shared resources in TinyOS. It walks through the process of making a request through the `Resource` interface and handling the `granted` event that is signaled back. We will connect multiple clients to a single shared resources and see how access to each of them gets arbitrated. We also show how to hold onto a resource until another client has requested it by implementing the `ResourceRequested` interface.

To begin, go to the `tinyos-2.x/apps/tutorials/SharedResourceDemo` directory and install this application on a mote. After installing the application you should see three leds flashing in sequence.

Let's take a look at the different components contained in this directory to see whats going on. Start with the top level application component: `SharedResourceDemoAppC`

```
configuration SharedResourceDemoAppC{
}
implementation {
  components MainC,LedsC, SharedResourceDemoC as App,
  new TimerMilliC() as Timer0,
  new TimerMilliC() as Timer1,
  new TimerMilliC() as Timer2;
  App -> MainC.Boot;
  App.Leds -> LedsC;
  App.Timer0 -> Timer0;
  App.Timer1 -> Timer1;
  App.Timer2 -> Timer2;

  components
  new SharedResourceC() as SharedResource0,
  new SharedResourceC() as SharedResource1,
  new SharedResourceC() as SharedResource2;
  App.Resource0 -> SharedResource0;
  App.Resource1 -> SharedResource1;
  App.Resource2 -> SharedResource2;
  App.ResourceOperations0 -> SharedResource0;
  App.ResourceOperations1 -> SharedResource1;
  App.ResourceOperations2 -> SharedResource2;
}
```

Other than the instantiation and wiring of the interfaces provided by the SharedResourceC component, this configuration is identical to the one presented in Lesson 1 for the Blink Application.

All shared resources in TinyOS are provided through a generic component similar to the SharedResourceC component. A resource client simply instantiates a new instance of this component and wires to the interfaces it provides. In this application, three instances of the SharedResourceC component are instantiated and wired to three different clients from the SharedResourceDemoC component. Each instantiation provides a Resource, ResourceOperations, and ResourceRequested interface, and uses a ResourceConfgigure interface. In this example, no wiring is done to the ResourceConfigure or ResourceRequested interface as wiring to to these interfaces is optional. The ResourceOperations interface is an **EXAMPLE** of a resource specific interface that a resource may provide to perform operations on it. Calls to commands through this interface will only succeed if the client calling them happens to have access to the resource when they are called.

Let's take a look at the SharedResourceDemoC to see how access is actually granted to a Resource.

```
module SharedResourceDemoC {
  uses {
    interface Boot;
    interface Leds;
    interface Timer as Timer0;
    interface Timer as Timer1;
    interface Timer as Timer2;

    interface Resource as Resource0;
    interface ResourceOperations as
ResourceOperations0;

    interface Resource as Resource1;
    interface ResourceOperations as
ResourceOperations1;

    interface Resource as Resource2;
    interface ResourceOperations as
ResourceOperations2;
  }
}
```

Each pair of `Resource/ResourceOperations` interfaces reperesents a different client of the shared resource used by this application. At boot time, we put in a request for the shared resource through each of these clients in the order (0,2,1).

```
event void Boot.booted() {
  call Resource0.request();
  call Resource2.request();
  call Resource1.request();
}
```

Each of these requests is serviced in the order of the arbitration policy used by the shared resource. In the case of `SharedResourceC`, a Round-Robin policy is used, so these requests are serviced in the order (0,1,2). If a first-come-first-serve policy were in use, they would we be serviced in the order the were put in, i.e. (0,2,1).

Whenever a client's request for a resource has been granted, the `Resource.granted()` event for that client gets signaled. In this application, the body of the granted event for each client simply performs an operation on the resource as provided through the `ResourceOperations` interface.

```
event void Resource0.granted() {
  call ResourceOperations0.operation();
}
event void Resource1.granted() {
  call ResourceOperations1.operation();
}
event void Resource2.granted() {
  call ResourceOperations2.operation();
}
```

Whenever one of these operations completes, a `ResourceOperations.operationDone()` event is signaled. Once this event is received by each client, a timer is started to hold onto the resource for 250 (binary) ms and an LED corresponding to that client is toggled.

```
#define HOLD_PERIOD 250

event void ResourceOperations0.operationDone(error_t
error) {
  call Timer0.startOneShot(HOLD_PERIOD);
  call Leds.led0Toggle();
}
event void ResourceOperations1.operationDone(error_t
error) {
```

```
  call Timer1.startOneShot(HOLD_PERIOD);
  call Leds.led1Toggle();
}
event void ResourceOperations2.operationDone(error_t
error) {
  call Timer2.startOneShot(HOLD_PERIOD);
  call Leds.led2Toggle();
}
```

Whenever one of these timers goes off, the client that started it releases the resource and immediately puts in a request for it again.

```
event void Timer0.fired() {
  call Resource0.release();
  call Resource0.request();
}
event void Timer1.fired() {
  call Resource1.release();
  call Resource1.request();
}
event void Timer2.fired() {
  call Resource2.release();
  call Resource2.request();
}
```

In this way, requests are continuously put in by each client, allowing the application to continuously flash the LEDs in the order in which requests are being serviced. As stated before, the SharedResourceC component services these requests in a round-robin fashion. If you would like to see the requests serviced in the order they are received (and see the LEDs flash accordingly), you can open up the SharedResourceP component in the apps/tutorials/SharedResourceDemo directory and replace the RoundRobinArbiter component with the FcfsArbiter component.

RoundRobinArbiter

```
configuration
SharedResourceP {
   provides interface
Resource[uint8_t id];
   provides interface
ResourceRequested[uint8_t
id];
   provides interface
ResourceOperations[uint8_t
id];
```

FcfsArbiter

```
configuration
SharedResourceP {
   provides interface
Resource[uint8_t id];
   provides interface
ResourceRequested[uint
8_t id];
   provides interface
ResourceOperations[uin
t8_t id];
```

```
   uses interface
ResourceConfigure[uint8_t
id];
}
implementation {
  components new
RoundRobinArbiterC(UQ_SHARE
D_RESOURCE) as Arbiter;
  ...
  ...
}
```

```
   uses interface
ResourceConfigure[uint
8_t id];
}
implementation {
  components new
FcfsArbiterC(UQ_SHARED
_RESOURCE) as Arbiter;
  ...
  ...
}
```

Looking through the rest of this component, you can see how its wiring matches the connections shown in Figure 2.

```
#define UQ_SHARED_RESOURCE   "Shared.Resource"
configuration SharedResourceP {
   provides interface Resource[uint8_t id];
   provides interface ResourceRequested[uint8_t id];
   provides interface ResourceOperations[uint8_t id];
   uses interface ResourceConfigure[uint8_t id];
}
implementation {
  components new
RoundRobinArbiterC(UQ_SHARED_RESOURCE) as Arbiter;
  components new SplitControlPowerManagerC() as
PowerManager;
  components ResourceP;
  components SharedResourceImplP;

  ResourceOperations = SharedResourceImplP;
  Resource = Arbiter;
  ResourceRequested = Arbiter;
  ResourceConfigure = Arbiter;
  SharedResourceImplP.ArbiterInfo -> Arbiter;
  PowerManager.ResourceDefaultOwner -> Arbiter;

  PowerManager.SplitControl -> ResourceP;
  SharedResourceImplP.ResourceOperations -> ResourceP;
}
```

Four different components are instantiated by this configuration:

```
components new RoundRobinArbiterC(UQ_SHARED_RESOURCE)
as Arbiter;
components new SplitControlPowerManagerC() as
PowerManager;
components ResourceP;
components SharedResourceImplP;
```

As we've already seen, the RoundRobinArbiterC component is used to provide arbitration between clients using SharedResourceC. The SplitControlPowerManagerC component is used to perform automatic power management of the resource to turn it on whenever a new client requests its use and shut it down whenever it goes idle. The ResourceP component is the implementation of a dedicated resource which provides a SplitControl interface and a ResourceOperations interface. This dedicated resource is wrapped by the SharedResourceImplP component in order to provide protected shared access to it. SharedResourceImplP wraps all the commands provided by the dedicated resource, and uses the ArbiterInfo interface to keep clients from calling them without first being granted access to the resource.

If you would like to see more examples of how to use the different arbiters and power managers provided in the default TinyOS distribution, please refer to the test applications located in tinyos-2.x/apps/tests/TestArbiter and tinyos-2.x/apps/tests/TestPowerManager. This tutorial has provided enough background information on how to use these components in order for you to sift through these applications on your own.

Conclusion

This tutorial has given an overview of how resource arbitration and mechanisms for performing power management on those resources is provided in TinyOS. It walked us through the steps necessary for:

1. Wiring in a shared resource for use by a client.
2. Using the Resource interface to gain access to a shared resource.
3. Changing the arbitration policy used by a particular shared resource.
4. Wrapping a dedicated resource and wiring in a power manager in order to create a shared resource.

While the power managers presented in this tutorial are powerful components for providing power management of shared resources, they are not the only power management mechanisms provided by TinyOS. Microcontroller power management is also preformed as outlined in TEP115. Whenever the task queue empties, the lowest power state that the microcontroller is capable of dropping to is automatically calculated and then switched to. In this way, the user is not burdened with explicity controlling these power states. The cc1000 and cc2420 radio implementations also

provide "Low Power Listening" (LPL) interfaces for controlling their duty cycles. The LPL implementation for the cc2420 can be found under tinyos-2.x/tos/chips/cc2420 and the LPL implementation for the cc1000 can be found under tinyos-2.x/tos/chips/cc1000. Take a look at **lesson 16** to see how this interface is used.

Related Documentation

- **TinyOS Programming Guide *Sections 6.2 and 7.4***
- **TEP 108: Resource Arbitration**
- **TEP 112: Microcontroller Power Management**
- **TEP 115: Power Management of Non-Virtualized Devices**

Lesson 10: Platforms

Last updated October 23 2006

Introduction

Many different hardware platforms (e.g micaZ, telos, eyesIFX) can be used with TinyOS. This lesson shows you what a platform port consists of, and how TinyOS reuses as much code as possible between different platforms. The lesson will proceed by showing how to do the port for an imaginary mote called "yamp", which has a MSP430 microcontroller and a CC2420 radio transceiver.

The **target audience** of this lesson consists of those people that want to better understand what the difference between e.g "make micaz" and "make telosb" is, and how these differences concretely map into the underlying files, definitions, etc.

Note that the material covered in this tutorial is **not** strictly necessary for regular tinyos developpers, and you can safely skip it if you have no intention of working down to the lowest level or developping new platforms.

Chips vs Platforms

Two key building blocks for any mote are the microcontroller and radio transceiver. Of course, both a microcontroller and a radio can be used on more than one platform. This is indeed the case for the MSP430 and CC2420 that our yamp mote uses (telos, eyes, and tinynode use the MSP430; telos and micaz use the CC2420).

Given this multiplicity of platforms, it would be vasly redundant if each platform developper had to rewrite the software support for each chip from scratch. While a chip may be physically wired in a different way on different platforms (e.g., a radio is connected to a different digital pins on the microcontroller), by far the largest part of the logic to deal with a chip is platform-independent.

Thus, platform-independent code to support a chip is placed in a chip-specific directory. Writing a platform port for a platform is then (essentially) a matter of **pulling in the code for each of the platform's chips, and "gluing" things together**. For example, the modules and components that support the CC2420 are in tos/chips/cc2420. This radio is used on both the telos and micaZ motes; the "gluing" is done by the modules in the tos/platforms/telosa/chips/cc2420 and tos/platforms/micaz/chips/cc2420/ directories.

Note that in general there may be more to a platform port than pulling in existing code for different chips, in particular when a new platform uses a chip that is not yet supported. Developping drivers for a new chip can be a non-trivial undertaking (especially for radios and microcontrollers); these aspects are not covered here.

Initial platform bring-up

As a first step to bring up the platform, we will stick to the bare minimum in order to compile and install the Null application on our yamp mote.

All platform-specific code goes in tos/platforms/. For example, we have tos/platforms/micaz or tos/platforms/tinynode. Our first step is to create a directory for the yamp platform:

```
$ cd tinyos-2.x/tos/platforms
$ mkdir yamp
```

The .platform file

Each platform directory (such as tos/platforms/yamp) should contain a file named ".platform", that contains basic compiler parameters information for each platform. So, create the file "tos/platforms/yamp/.platform" (note the **.** in "**.**platform" !!!), that contains the following:

```
push( @includes, qw(

%T/chips/cc2420
%T/chips/msp430
%T/chips/msp430/adc12
%T/chips/msp430/dma
%T/chips/msp430/pins
%T/chips/msp430/timer
```

```
  %T/chips/msp430/usart
  %T/chips/msp430/sensors
  %T/lib/timer
  %T/lib/serial
  %T/lib/power
  ) );

  @opts = qw(

  -gcc=msp430-gcc
  -mmcu=msp430x1611
  -fnesc-target=msp430
  -fnesc-no-debug
  -fnesc-
scheduler=TinySchedulerC,TinySchedulerC.TaskBasic,Task
Basic,TaskBasic,runTask,postTask

  );
```

This file contains perl snippets that are intepreted by the ncc compiler. The first statement simply adds some directories to the include path that is used when compiling an application for the yamp platform (the %T gets expanded to the full location of tinyos-2.x/tos, using the TOS2DIR environment variable). Note that we have included the CC2420 and MSP430 directories, as well as some libraries.

The second statement defines the @opts list, that contains various parameters passed to nesc. Please consult the nesc documentation for information on the meaning of these parameters.

The hardware.h file

Each platform directory also has a file named "hardware.h" that is included by default when compiling an application for that platform. This can define platform-specific constants, pin names, or also include other "external" header files (e.g. msp430hardware.h in our case, or atm128hardware.h for platforms using the atm128 MCU).
So, create the file "tos/platforms/yamp/hardware.h" with the following contents:

```
  #ifndef _H_hardware_h
  #define _H_hardware_h

  #include "msp430hardware.h"

  // LEDs
```

```
TOSH_ASSIGN_PIN(RED_LED, 5, 4);
TOSH_ASSIGN_PIN(GREEN_LED, 5, 5);
TOSH_ASSIGN_PIN(YELLOW_LED, 5, 6);

// UART pins
TOSH_ASSIGN_PIN(SOMI0, 3, 2);
TOSH_ASSIGN_PIN(SIMO0, 3, 1);
TOSH_ASSIGN_PIN(UCLK0, 3, 3);
TOSH_ASSIGN_PIN(UTXD0, 3, 4);
TOSH_ASSIGN_PIN(URXD0, 3, 5);
TOSH_ASSIGN_PIN(UTXD1, 3, 6);
TOSH_ASSIGN_PIN(URXD1, 3, 7);
TOSH_ASSIGN_PIN(UCLK1, 5, 3);
TOSH_ASSIGN_PIN(SOMI1, 5, 2);
TOSH_ASSIGN_PIN(SIMO1, 5, 1);


#endif // _H_hardware_h
```

This file simply pulls in msp430hardware.h from tos/chips/msp430 (the compiler will find it because we have added this directory to our search path in the .platform created previously) and defines some physical pins using macros from msp430hardware.h. For example, on our yamp mote, the red led is physically connected to the general purpose I/O (GPIO) pin 5.4.

Some other very important functions (that are defined in msp430hardware.h and so pulled in indirectly via this hardware.h) concern the disabling of interrupts for atomic sections (atomic blocks in nesc code essentially get converted into __nesc_atomic_start() and __nesc_atomic_end()). How interrupts are disabled is of course microcontroller specific; the same applies to putting the microcontroller to sleep (as is done by the scheduler when there are no more tasks to run, using the McuSleep interface). These functions must be somehow defined for each platform, typically by means of an #include'd MCU-specific file.
As an exercise, try finding the definitions of __nesc_atomic_start() *and* __nesc_atomic_end() *for the micaZ and intelmote2 platforms.*

Setting up the build environment and building the "null" app

Before pulling in existing chip drivers or writing any code, we must set up the build environment so that it is aware of and supports our platform. Once this is done, we will define the basic TinyOS module for our platform, and use the Null app (in tinyos-2.x/apps/null) in order to test that our platform is properly configured. As per it's description in its README file, Null is an empty skeleton application. It is useful to test that the build environment is functional in its most minimal

sense, i.e., you can correctly compile an application. So, let's go ahead and try to compile Null for the yamp platform:

```
  $ cd tinyos-2.x/apps/Null
  $ make yamp
/home/henridf/work/tinyos-2.x/support/make/
Makerules:166: ***

Usage:  make
        make  help

        Valid targets: all btnode3 clean eyesIFX
eyesIFXv1 eyesIFXv2 intelmote2 mica2 mica2dot micaz
null telos telosa telosb tinynode tmote
        Valid extras: docs ident_flags nescDecls
nowiring rpc sim sim-cygwin sim-fast tos_image verbose
wiring

 Welcome to the TinyOS make system!

 You must specify one of the valid targets and
possibly some combination of
 the extra options.  Many targets have custom extras
and extended help, so be
 sure to try "make  help" to learn of all the
available features.

 Global extras:

   docs    : compile additional nescdoc documentation
   tinysec : compile with TinySec secure communication

ERROR, "yamp tos-ident-flags tos_image" does not
specify a valid target.  Stop.
```

The problem is that we need to define the platform in the *TinyOS build system*, so that the make invocation above recognizes the yamp platform. The TinyOS build system is a Makefile-based set of rules and definitions that has a very rich functionality. This includes invoking necessary compilation commands as with any build system, but goes much further and includes support for other important aspects such as device reprogramming or supporting multiple platforms and targets.

A full description of the inner workings of the TinyOS build system is beyond the scope of this tutorial. For now, we will simply see how to define the yamp platform so that the "make yamp"

command does what it should. (For those that want to delve deeper, start with "tinyos-2.x/support/make/Makerules".)

Defining a make target

The TinyOS build system resides in "tinyos-2.x/support/make". The strict minimum for a platform to be recognized by the build system (i.e., for the build system to understand that "yamp" is a legal platform when we enter "make yamp") is the existence of a *platformname*.target file in the aforementioned make directory.
So, create the file "tinyos-2.x/support/make/yamp.target" with the following contents:

```
PLATFORM = yamp

$(call TOSMake_include_platform,msp)

yamp: $(BUILD_DEPS)
   @:
```

This sets the PLATFORM variable to yamp, includes the msp platform ("make/msp/msp.rules") file, and provides in the last two lines a make rule for building a yamp application using standard Makefile syntax. Now, let's go back and try to compile the Null app as before. This time we get:

```
[18:23 henridf@tinyblue: ~/work/tinyos-2.x/apps/Null]
make yamp
mkdir -p build/yamp
    compiling NullAppC to a yamp binary
ncc -o build/yamp/main.exe -Os -Wall -Wshadow -
DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -target=yamp -fnesc-
cfile=build/yamp/app.c -board=   NullAppC.nc -lm
In file included from NullAppC.nc:42:
In component `MainC':
/home/henridf/work/tinyos-2.x/tos/system/MainC.nc:50:
component PlatformC not found
/home/henridf/work/tinyos-2.x/tos/system/MainC.nc:53:
no match
make: *** [exe0] Error 1
```

So there's progress of sorts, since now we're getting a "real" compilation error as opposed to not even making it past the build system. Let's take a closer look at the output. The ncc compiler is unhappy about not finding a "PlatformC" component. The "PlatformC" component must be defined for each platform. Its role and placement in the system is described in more detail in TEP107. For now, suffice to cite from that TEP that: *A port of TinyOS to a new plaform MUST include a component PlatformC which provides one and only one instance of the Init interface.*.
Create the file "tos/platforms/yamp/PlatformP.nc" with the following contents:

```
#include "hardware.h"
```

```
module PlatformP{
  provides interface Init;
  uses interface Init as Msp430ClockInit;
  uses interface Init as LedsInit;
}
implementation {
  command error_t Init.init() {
    call Msp430ClockInit.init();
    call LedsInit.init();
    return SUCCESS;
  }

  default command error_t LedsInit.init() { return
SUCCESS; }

}
```

, and create the file "tos/platforms/yamp/PlatformC.nc" as:

```
#include "hardware.h"

configuration PlatformC
{
  provides interface Init;
}
implementation
{
  components PlatformP
    , Msp430ClockC
    ;

  Init = PlatformP;
  PlatformP.Msp430ClockInit -> Msp430ClockC.Init;
}
```

Now, compilation of the Null application finally works for the yamp platform:

```
[19:47 henridf@tinyblue: ~/work/tinyos-2.x/apps/Null]
make yamp
mkdir -p build/yamp
    compiling NullAppC to a yamp binary
ncc -o build/yamp/main.exe -Os -Wall -Wshadow -
DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -target=yamp -fnesc-
cfile=build/yamp/app.c -board=   NullAppC.nc -lm
    compiled NullAppC to build/yamp/main.exe
            1216 bytes in ROM
               6 bytes in RAM
```

```
msp430-objcopy --output-target=ihex
build/yamp/main.exe build/yamp/main.ihex
    writing TOS image
```

Getting Blink to work

With the previous steps, we now have the basic foundation to start working on our yamp platform: the basic platform definitions are in place, and the build system recognizes and correctly acts upon the "make yamp" target. We haven't yet actually *programmed* our mote yet.

The next step in the bring-up of a platform, that we will cover in this part, is to program a node with an application and verify that it actually works. We'll do this with Blink, because it is simple, and easy to verify that it works. As a bonus, we'll also have validated basic timer functionality once we see those Leds turning on and off.

As a first step, let's go to the Blink application directory and try to compile the application:

```
[19:06 henridf@tinyblue: ~/work/tinyos-2.x/apps/Blink]
make yamp
mkdir -p build/yamp
    compiling BlinkAppC to a yamp binary
ncc -o build/yamp/main.exe -Os -Wall -Wshadow -
DDEF_TOS_AM_GROUP=0x7d -Wnesc-all -target=yamp -fnesc-
cfile=build/yamp/app.c -board=   BlinkAppC.nc -lm
In file included from BlinkAppC.nc:45:
In component `LedsC':
/home/henridf/work/tinyos-2.x/tos/system/LedsC.nc:38:
component PlatformLedsC not found
/home/henridf/work/tinyos-2.x/tos/system/LedsC.nc:42:
cannot find `Init'
/home/henridf/work/tinyos-2.x/tos/system/LedsC.nc:43:
cannot find `Led0'
/home/henridf/work/tinyos-2.x/tos/system/LedsC.nc:44:
cannot find `Led1'
/home/henridf/work/tinyos-2.x/tos/system/LedsC.nc:45:
cannot find `Led2'
make: *** [exe0] Error 1
```

The compiler cannot find the component "PlatformLedsC" that is referred to in the file tos/system/LedsC.nc. As the name indicates, "PlatformLedsC" is a platform-specific component, and thus we will need to define this component for the yamp platform.

Why should there be such a platform-specific component for accessing Leds? This is because at the lowest level, i.e in hardware, Leds are implemented differently on different platforms. Typically, a Led is connected to a microcontroller I/O pin, and the Led is turned on/off by setting the appropriate output 0/1 on that pin. This is in fact the model used on all current TinyOS platforms. But even in this simple and uniform model, (and disregarding the fact that future platforms may use different hardware implementations and not connect each Led directly to an I/O pin), we have that the lowest-level to turn on/off a Led must be defined on a per-platform basis.

Now, consider Leds from another perspective, namely that of the Leds.nc interface (defined in tos/interfaces/Leds.nc). In this interface, we have commands such as `get();` in principle such a command does not need to be platform-dependent: the code that maintains the current state of a Led and returns it via the get() call does not need to be re-written each time a Led is connected to a different pin (of course re-writing get() for each platform would not be much overhead given its simplicity; this argument clearly becomes far stronger in more complex situations involving entire chips rather than individual GPIOs).

The key notion that the above example is simply that there is a boundary above which software components are platform-independent, and below which components are specifically written with one hardware platform in mind. This is at heart a very simple concept; its complete instantiation in TinyOS is of course richer than the above example, and is the topic of TEP2 (Hardware Abstraction Architecture).

Let's now return to the task at hand, which is to make Blink work on our platform. If we take a closer look at the file "tos/system/LedsC.nc", we see that it contains a configuration that wires the module LedsP with modules Init, Leds0, Leds1, and Leds2, all of which are to be provided by the (still missing) PlatformLedsC. Taking a closer look at "tos/system/LedsP.nc", we see that the Leds0,1,2 modules used by LedsP are GeneralIO interfaces. The GeneralIO interface (see "tos/interfaces/GeneralIO.nc" and TEP 117) simply encapsulates a digital I/O pin and provides functions to control its direction and get/set its input/output values.

So, we need to create a PlatformLedsC configuration that shall provide three GeneralIO interfaces and an Init. This is done by creating the file "tos/platforms/yamp/PlatformLedsC.nc" with the following contents:

```
#include "hardware.h"

configuration PlatformLedsC {
  provides interface GeneralIO as Led0;
  provides interface GeneralIO as Led1;
  provides interface GeneralIO as Led2;
  uses interface Init;
}
implementation
{
```

```
  components
      HplMsp430GeneralIOC as GeneralIOC
    , new Msp430GpioC() as Led0Impl
    , new Msp430GpioC() as Led1Impl
    , new Msp430GpioC() as Led2Impl
    ;
  components PlatformP;

  Init = PlatformP.LedsInit;

  Led0 = Led0Impl;
  Led0Impl -> GeneralIOC.Port54;

  Led1 = Led1Impl;
  Led1Impl -> GeneralIOC.Port55;

  Led2 = Led2Impl;
  Led2Impl -> GeneralIOC.Port56;

}
```

We refer the reader to the TinyOS Programming Guide cited below for more explanations on how the above confiuration uses generics (ie with the "new" keyword). For the purpose of this lesson, the key point is that we are wiring to ports 5.4, 5.5, and 5.6 -- we shall suppose that these are the MSP430 microcontroller pins to which the three Leds are connected on the yamp platform.

With the above file in place, we can now compile Blink for the yamp platform. How do we test that the application actually works? We have thus far presented yamp as an imaginary platform, but it turns out that the above application should work on any platform with the MSP430x1611 microcontroller and where the Leds are connected to microcontroller ports 5.4-5.6. Not entirely coincidentally, these are exactly the Led wirings used by the telos/tmote platforms. So those readers that have a telos/tmote at hand can test this application on it. (Testing on a tinynode or eyes mote is also easy and only requires changing the pin wirings in PlatformLedsC to follow those of that platform; running this application on mica-family motes will require more changes since they use a different microcontroller).

Now, enter the following command (where you have suitably replaced /dev/ttyXXX by the appropriate serial port),

```
  make yamp install bsl,/dev/ttyXXX
```

and you will see the Leds of your impersonated (by a telos) yamp node Blinking!

Conclusion

This lesson has introduced the notion of per-platform support in TinyOS using as a guiding example the development of a platform port to an imaginary "yamp" platform. We have seen how introducing support for a new platform requires touching not only nesc code but also adding some rules to the build system. This tutorial also touched upon the notions of Hardware Abstraction Architecture (HAA) that is central to the clean and modular support of different platforms and chips in TinyOS.

The steps outlined here did not cover what is the hardest part of a platform port: developping the components to drive a radio transceiver or MCU (which are necessary if the platform uses chips that are not currently supported in TinyOS). Developping such drivers is an advanced topic that is beyond the scope of the tutorials; for those curious to gain some insight we recommend perusing the code for a chip (e.g the CC2420 radio in tos/chips/cc2420 or the xe1205 radio in tos/chips/xe1205) armed with that chips datasheet and the platform schematics.

Related Documentation

- **nesc at sourceforge**
- **nesC reference manual**
- **TinyOS Programming Guide**
- **TEP 2: Hardware Abstraction Architecture**
- **TEP 106: Schedulers and Tasks**
- **TEP 107: TinyOS 2.x Boot Sequence**
- **TEP 117: Low-level I/O**

Lesson 11: Simulation with TOSSIM

Last Modified: 20 April 2007

This lesson introduces the TOSSIM simulator. You will become familiar with how to compile TOSSIM and use some of its functionality. You will learn how to:

- Compile TOSSIM.
- Configure a simulation in Python and C++.
- Inspect variables.
- Inject packets.

Note: This tutorial is for TOSSIM in TinyOS 2.0.1. If you are using TinyOS 2.0.0, it has a slightly different A HREF="lesson11-200.html">tutorial. The principal difference between the two is how you specify noise in RF simulation.

Introduction

TOSSIM simulates entire TinyOS applications. It works by replacing components with simulation implementations. The level at which components are replaced is very flexible: for example, there is a simulation implementation of millisecond timers that replaces `HilTimerMilliC`, while there is also an implementation for atmega128 platforms that replaces the HPL components of the hardware clocks. The former is general and can be used for any platform, but lacks the fidelity of capturing an actual chips behavior, as the latter does. Similarly, TOSSIM can replace a packet-level communication component for packet-level simulation, or replace a low-level radio chip component for a more precise simulation of the code execution. TOSSIM is a discrete event simulator. When it runs, it pulls events of the event queue (sorted by time) and executes them. Depending on the level of simulation, simulation events can represent hardware interrupts or high-level system events (such as packet reception). Additionally, tasks are simulation events, so that posting a task causes it to run a short time (e.g., a few microseconds) in the future. TOSSIM is a library: you must write a program that configures a simulation and runs it. TOSSIM supports two programming interfaces, Python and C++. Python allows you to interact with a running simulation dynamically, like a powerful debugger. However, as the interpretation can be a performance bottleneck when obtaining results, TOSSIM also has a C++ interface. Usually, transforming code from one to the other is very simple. TOSSIM currently does not support gathering power measurements.

Compiling TOSSIM

TOSSIM is a TinyOS library. Its core code lives in **`tos/lib/tossim`**. Every TinyOS source directory has an optional `sim` subdirectory, which contains simulation implementations of that package. For example, **`tos/chips/atm128/timer/sim`** contains TOSSIM implementations of some of the Atmega128 timer abstractions.

To compile TOSSIM, you pass the `sim` option to make:

```
$ cd apps/Blink
$ make micaz sim
```

Currently, the only platform TOSSIM supports is the micaz. You should see output similar to this:

```
mkdir -p build/micaz
  placing object files in build/micaz
  writing XML schema to app.xml
  compiling BlinkAppC to object file sim.o
ncc -c -fPIC -o build/micaz/sim.o -g -O0 -
tossim -fnesc-nido-tosnodes=1000 -fnesc-simulate -
fnesc-nido-motenumber=sim_node\(\)   -finline-
limit=100000 -Wall -Wshadow -DDEF_TOS_AM_GROUP=0x7d -
Wnesc-all -target=micaz -fnesc-cfile=build/micaz/app.c
```

```
-board=micasb  -Wno-nesc-data-race BlinkAppC.nc   -
fnesc-dump=components -fnesc-dump=variables -fnesc-
dump=constants -fnesc-dump=typedefs -fnesc-
dump=interfacedefs -fnesc-dump=tags -fnesc-
dumpfile=app.xml
           compiling Python support into pytossim.o
and tossim.o
         g++ -c -shared -fPIC -o
build/micaz/pytossim.o -g -O0  /home/pal/src/tinyos-
2.x/tos/lib/tossim/tossim_wrap.cxx
-I/usr/include/python2.3
-I/home/pal/src/tinyos-2.x/tos/lib/tossim -
DHAVE_CONFIG_H
         g++ -c -shared -fPIC -o build/micaz/tossim.o
-g -O0
/home/pal/src/tinyos-2.x/tos/lib/tossim/tossim.c
-I/usr/include/python2.3
-I/home/pal/src/tinyos-2.x/tos/lib/tossim
           linking into shared object
./_TOSSIMmodule.so
         g++ -shared build/micaz/pytossim.o
build/micaz/sim.o build/micaz/tossim.o -lstdc++ -o
_TOSSIMmodule.so
         copying Python script interface TOSSIM.py
from lib/tossim to local directory
```

Depending on what OS you are using and what packages are installed, TOSSIM may not properly compile on the first try. **Appendix A** addresses some of the common causes and gives possible solutions.

Compiling TOSSIM has five basic steps. Let's go through them one by one.

Writing an XML schema

```
         writing XML schema to app.xml
```

The first thing the TOSSIM build process does is use nesc-dump to produce an XML document that describes the application. Among other things, this document descibes the name and type of every variable.

Compiling the TinyOS Application

Besides introducing all of these new compilation steps, the `sim` option changes the include paths of the application. If the application has a series of includes

```
          -Ia -Ib -Ic
```

Then the sim option transforms the list to

```
          -Ia/sim -Ib/sim -Ic/sim -I%T/lib/tossim -Ia
-Ib -Ic
```

This means that any system-specific simulation implementations will be used first, followed by generic TOSSIM implementations, followed by standard implementations. The `sim` option also passes a bunch of arguments to the compiler, so it knows to compile for simulation.

The product of this step is an object file, `sim.o`, which lives in the platform's build directory. This object file has a set of C functions which configure the simulation and control execution.

Compiling the Programming Interface

```
            compiling Python support into pytossim.o
and tossim.o
          g++ -c -shared -fPIC -o
build/micaz/pytossim.o -g -O0 \

/home/pal/src/tinyos-2.x/tos/lib/tossim/tossim_wrap.cx
x \
          -I/usr/include/python2.3
-I/home/pal/src/tinyos-2.x/tos/lib/tossim \
          -DHAVE_CONFIG_H
          g++ -c -shared -fPIC -o build/micaz/tossim.o
-g -O0 \

/home/pal/src/tinyos-2.x/tos/lib/tossim/tossim.c \
          -I/usr/include/python2.3
-I/home/pal/src/tinyos-2.x/tos/lib/tossim
```

The next step compiles the support for the C++ and Python programming interfaces. The Python interface is actually built on top of the C++ interface. Calling a Python object calls a C++ object, which then calls TOSSIM through the C interface. `tossim.o` contains the C++ code, while `pytossim.o` contains the Python support. These files have to be compiled separately because C++ doesn't understand nesC, and nesC doesn't understand C++.

Building the shared object

```
            linking into shared object
./_TOSSIMmodule.so
          g++ -shared build/micaz/pytossim.o
build/micaz/sim.o build/micaz/tossim.o -lstdc++ -o
_TOSSIMmodule.so
```

The next to last step is to build a shared library that contains the TOSSIM code, the C++ support, and the Python support.

Copying Python Support

```
            copying Python script interface TOSSIM.py
from lib/tossim to local directory
```

Finally, there is the Python code that calls into the shared object. This code exists in `lib/tossim`, and the make process copies it into the local directory.

Running TOSSIM with Python

Go into the `RadioCountToLeds` application and build TOSSIM:

```
$ cd tinyos-2.x/apps/RadioCountToLeds
$ make micaz sim
```

We'll start with running a simulation in Python. You can either write a script and just tell Python to run it, or you can use Python interactively. We'll start with the latter. Fire up your Python interpreter:

```
$ python
```

You should see a prompt like this:

```
Python 2.3.4 (#1, Nov  4 2004, 14:13:38)
[GCC 3.4.2 20041017 (Red Hat 3.4.2-6.fc3)] on linux2
Type "help", "copyright", "credits" or "license" for
more information.
>>>
```

The first thing we need to do is import TOSSIM and create a TOSSIM object. Type

```
>>> from TOSSIM import *
>>> t = Tossim([])
```

The square brackets are an optional argument that lets you access variables in the simulation. We'll get to how to use that later. In this case, we're telling TOSSIM that there are no variables that we want to look at. The way you run a TOSSIM simulation is with the `runNextEvent` function. For example:

```
>>> t.runNextEvent()
0
```

When you tell TOSSIM to run the next event, it returns 0. This means that there was no next event to run. The reason is simple: we haven't told any nodes to boot. This snippet of code will tell mote 32 to boot at time 45654 and run its first event (booting):

```
>>> m = t.getNode(32);
>>> m.bootAtTime(45654);
>>> t.runNextEvent()
1
```

Segmentation faults: If trying to do this causes TOSSIM to throw a segmentation violation (segfault), then chances are you are using a version of gcc that does not work well with the dynamic linking that TOSSIM is doing. In particular, it has been verified to work properly with 4.0.2 and 3.6, but some people have encountered problems with gcc 4.1.1.

Instead of using raw simulation ticks, you can also use the call `ticksPerSecond()`. However, you want to be careful to add some random bits into this number: having every node perfectly synchronized and only different in phase in terms of seconds can lead to strange results.

```
>>> m = t.getNode(32);
>>> m.bootAtTime(4 * t.ticksPerSecond() + 242119);
```

```
>>> t.runNextEVent()
1
```

Now, `runNextEvent` returns 1, because there was an event to run. But we have no way of knowing whether the node has booted or not. We can find this out in one of two ways. The first is that we can just ask it:

```
>>> m.isOn()
1
>>> m.turnOff()
>>> m.isOn()
0
>>> m.bootAtTime(560000)
>>> t.runNextEvent()
0
>>> t.runNextEvent()
1
```

Note that the first `runNextEvent` returned 0. This is because when we turned the mote off, there was still an event in the queue, for its next timer tick. However, since the mote was off when the event was handled in that call, `runNextEvent` returned 0. The second call to `runNextEvent` returned 1 for the second boot event, at time 560000.

A Tossim object has several useful functions. In Python, you can generally see the signature of an object with the `dir` function. E.g.:

```
>>> t = Tossim([])
>>> dir(t)
['__class__', '__del__', '__delattr__', '__dict__',
'__doc__', '__getattr__',
'__getattribute__', '__hash__', '__init__',
'__module__', '__new__',
'__reduce__', '__reduce_ex__', '__repr__',
'__setattr__', '__str__',
'__swig_getmethods__', '__swig_setmethods__',
'__weakref__', 'addChannel',
'currentNode', 'getNode', 'init', 'mac', 'newPacket',
'radio', 'removeChannel',
'runNextEvent', 'setCurrentNode', 'setTime', 'this',
'thisown', 'ticksPerSecond', 'time', 'timeStr']
```

The most common utility functions are:

- **`currentNode()`**: returns the ID of the current node.
- **`getNode(id)`**: returns an object representing a specific mote
- **`runNextEvent()`**: run a simulation event
- **`time()`**: return the current time in simulation ticks as a large integer
- **`timeStr()`**: return a string representation of the current time
- **`init()`**: initialize TOSSIM
- **`mac()`**: return the object representing the media access layer
- **`radio()`**: return the object representing the radio model
- **`addChannel(ch, output)`**: add *output* as an output to channel *ch*
- **`removeChannel(ch, output)`**: remove *output* as an output to channel *ch*
- **`ticksPerSecond()`**: return how many simulation ticks there are in a simulated second

The next section discusses the last two.

Debugging Statements

The second approach to know whether a node is on is to tell it to print something out when it boots. TOSSIM has a debugging output system, called `dbg`. There are four `dbg` calls:

- `dbg`: print a debugging statement preceded by the node ID.
- `dbg_clear`: print a debugging statement which is not preceded by the node ID. This allows you to easily print out complex data types, such as packets, without interspersing node IDs through the output.
- `dbgerror`: print an error statement preceded by the node ID
- `dbgerror_clear`: print an error statement which is not preceded by the node ID

Go into `RadioCountToLedsC` and modify the `Boot.booted` event to print out a debug message when it boots, such as this:

```
event void Boot.booted() {
  call Leds.led0On();
  dbg("Boot", "Application booted.\n");
  call AMControl.start();
}
```

Calls to the debugging calls take two or more parameters. The first parameter ("Boot" in the above example) defines the output *channel*. An output channel is simply a string. The second and

subsequent parameters are the message to output. They are identical to a printf statement. For example `RadioCountToLedsC` has this call:

```
event message_t* Receive.receive(message_t* bufPtr,
void* payload, uint8_t len) {
  dbg("RadioCountToLedsC", "Received packet of length
%hhu.\n", len);
  ...
}
```

which prints out the length of received packet as an 8-bit unsigned value (%hhu).

Once you have added the debugging statement to the event, recompile the application with `make micaz sim` and start up your Python interpreter. Load the TOSSIM module and schedule a mote to boot as before:

```
>>> from TOSSIM import *
>>> t = Tossim([])
>>> m = t.getNode(32);
>>> m.bootAtTime(45654);
```

This time, however, we want to see the debugging message that the mote has booted. TOSSIM's debugging output can be configured on a per-channel basis. So, for example, you can tell TOSSIM to send the "Boot" channel to standard output, but another channel, say "AM", to a file. By default, a channel has no destination, and so messages to it are discarded.

In this case, we want to send the Boot channel to standard output. To do this, we need to import the `sys` Python package, which lets us refer to standard out. We can then tell TOSSIM to send Boot messages to this destination:

```
>>> import sys
>>> t.addChannel("Boot", sys.stdout);
1
```

The return value shows that the channel was added successfully. Run the first simulation event, and the mote boots:

```
>>> t.runNextEvent()
DEBUG (32): Application booted.
1
```

The only difference between debug and error functions is the string output at the beginning of a message. Debug statements print `DEBUG (n)`, while error statements print `ERROR (n)`.

A debugging statement can have multiple output channels. Each channel name is delimited by commas:

```
event void Boot.booted() {
  call Leds.led0On();
  dbg("Boot,RadioCountToLedsC", "Application booted.\
n");
  call AMControl.start();
}
```

If a statement has multiple channels and those channels share outputs, then TOSSIM only prints the message once. For example, if both the Boot channel and RadioCountToLedsC channel were connected to standard out, TOSSIM will only print one message. For example, this series of debug statements

```
event void Boot.booted() {
  call Leds.led0On();
  dbg("Boot,RadioCountToLedsC", "Application booted.\
n");
  dbg("RadioCountToLedsC", "Application booted again.\
n");
  dbg("Boot", "Application booted a third time.\n");
  call AMControl.start();
}
```

when configured so

```
>>> import sys
>>> t.addChannel("Boot", sys.stdout)
>>> t.addChannel("RadioCountToLedsC", sys.stdout)
```

will print out this:

```
DEBUG (32): Application booted.
DEBUG (32): Application booted again.
DEBUG (32): Application booted a third time.
```

A channel can have multiple outputs. For example, this script will tell TOSSIM to write `RadioCountToLedsC` messages to standard output, but to write `Boot` messages to both standard output and a file named `log.txt`:

```
>>> import sys
>>> f = open("log.txt", "w")
>>> t.addChannel("Boot", f)
>>> t.addChannel("Boot", sys.stdout)
>>> t.addChannel("RadioCountToLedsC", sys.stdout)
```

Configuring a Network

When you start TOSSIM, no node can communicate with any other. In order to be able to simulate network behavior, you have to specify a *network topology*. Internally, TOSSIM is structured so that you can easily change the underlying radio simulation, but that's beyond the scope of this tutorial. The default TOSSIM radio model is signal-strength based. You provide a graph to the simulator that describes the propagation strengths. You also specify noise floor, and receiver sensitivity. There are some very early results that describe current sensor platforms (e.g., the mica2) in these terms. Because all of this is through a scripting interface, rather than provide a specific radio model, TOSSIM tries to provide a few low-level primitives that can express a wide range of radios and behavior.

You control the radio simulation through a Python Radio object:

```
>>> from TOSSIM import *
>>> t = Tossim([])
>>> r = t.radio()
>>> dir(r)
['__class__', '__del__', '__delattr__', '__dict__',
'__doc__',
'__getattr__', '__getattribute__', '__hash__',
'__init__',
'__module__', '__new__', '__reduce__',
'__reduce_ex__',
'__repr__', '__setattr__', '__str__',
'__swig_getmethods__',
'__swig_setmethods__', '__weakref__', 'add',
'connected',
'gain', 'remove', 'setNoise', 'this', 'thisown',
]
```

The first set of methods (with the double underscores) are ones that you usually don't call. The important ones are at the end. They are:

- **`add(src, dest, gain)`**: Add a link from *src* to *dest* with *gain*. When *src* transmits, *dest* will receive a packet attenuated by the *gain* value.
- **`connected(src, dest)`**: Return whether there is a link from *src* to *dest*.
- **`gain(src, dest)`**: Return the gain value of the link from *src* to *dest*.
- **`threshold()`**: Return the CCA threshold.

- **`setThreshold(val)`**: Set the CCA threshold value in dBm. The default is -72dbM.

The default values for TOSSIM's radio model are based on the CC2420 radio, used in the micaZ, telos family, and imote2. It uses an SNR curve derived from experimental data collected using two micaZ nodes, RF shielding, and a variable attenuator.

In addition to the radio propagation above, TOSSIM also simulates the RF noise and interference a node hears, both from other nodes as well as outside sources. It uses the Closest Pattern Matching (CPM) algorithm. CPM takes a noise trace as input and generates a statistical model from it. This model can capture bursts of interference and other correlated phenomena, such that it greatly improves the quality of the RF simulation. It is not perfect (there are several things it does not handle, such as correlated interference at nodes that are close to one another), but it is much better than traditional, independent packet loss models. For more details, please refer to the paper "Improving Wireless Simulation through Noise Modeling," by Lee et al.

To configure CPM, you need to feed it a noise trace. You accomplish this by calling `addNoiseTraceReading` on a Mote object. Once you have fed the entire noise trace, you must call `createNoiseModel` on the node. The directory `tos/lib/tossim/noise` contains sample noise traces, which are a series of noise readings, one per line. For example, these are the first 10 lines of meyer-heavy.txt, which is a noise trace taken from Meyer Library at Stanford University:

```
-39
-98
-98
-98
-99
-98
-94
-98
-98
-98
```

As you can see, the hardware noise floor is around -98 dBm, but there are spikes of interference around -86dBm and -87dBm.

This piece of code will give a node a noise model from a noise trace file. It works for nodes 0-7: you can change the range appropriately:

```
noise = open("meyer-heavy.txt", "r")
lines = noise.readlines()
for line in lines:
    str = line.strip()
```

```
    if (str != ""):
        val = int(str)
        for i in range(0, 7):
            t.getNode(i).addNoiseTraceReading(val)

for i in range(0, 7):
    t.getNode(i).createNoiseModel()
```

CPM can use a good deal of RAM: using the entire meyer-heavy trace as input has a cost of approximately 10MB per node. You can reduce this overhead by using a shorter trace; this will of course reduce simulation fidelity. The trace must be at least 100 entries long, or CPM will not work as it does not have enough data to generate a statistical model.

You can also use

The Radio object only deals with physical layer propagation. The MAC object deals with the data link layer, packet lengths, and radio bandwidth. The default TOSSIM MAC object is for a CSMA protocol. You get a reference to the MAC object by calling `mac()` on a Tossim object:

```
>>> mac = t.mac()
```

The default MAC object has a large number of functions, for controlling backoff behavior, packet preamble length, radio bandwidth, etc. All time values are specified in terms of radio symbols, and you can configure the number of symbols per second and bits per symbol. By default, the MAC object is configured to act like the standard TinyOS 2.0 CC2420 stack: it has 4 bits per symbol and 64k symbols per second, for 256kbps. This is a subset of the MAC functions that could be useful for changing backoff behavior. Every accessor function has a corresponding set function that takes an integer as a parameter. E.g., there's `int initHigh()` and `void setInitHigh(int val)`. The default value for each parameter is shown italicized in parentheses.

- **initHigh**: The upper bound of the initial backoff range. *(400)*
- **initLow**: The lower bound of the initial backoff range. *(20)*
- **high**: The upper bound of the backoff range. This is multiplied by the exponent base to the nth power, where n is the number of previous backoffs. So if the node had its initial backoff, then the upper bound is high * base, while if it is after the second backoff then the upper bound is high * base * base. *(160)*
- **low**: The lower bound of the backoff range. This is multiplied by the exponent base to the nth power, where n is the number of previous backoffs. So if the node had its initial backoff, then the upper bound is low * base, while if it is after the second backoff then the upper bound is low * base * base. *(20)*
- **symbolsPerSec**: The number of symbols per second that the radio can transmit. *(65536)*
- **bitsPerSymbol**: The number of bits per radio symbol. Multiplying this by the symbols per second gives the radio bandwidth. *(4)*

- **preambleLength**: How long a packet preamble is. This is added to the duration of transmission for every packet. *(12)*
- **exponentBase**: The base of the exponent used to calculate backoff. Setting it to 2 provides binary exponential backoff. *(0)*.
- **maxIterations**: The maximum number of times the radio will back off before signaling failure, zero signifies forever. *(0)*.
- **minFreeSamples**: The number of times the radio must detect a clear channel before it will transmit. This is important for protocols like 802.15.4, whose synchonrous acknowledgments requires that this be greater than 1 (you could have sampled in the dead time when the radios are changing between RX and TX mode). *(2)*
- **rxtxDelay**: The time it takes to change the radio from RX to TX mode (or vice versa). *(32)*
- **ackTime**: The time it takes to transmit a synchonrous acknowledgment, not including the requisite RX/TX transition.*(34)*

Any and all of these configuration constants can be changed at compile time with #define directives. Look at `tos/lib/tossim/sim_csma.h`.

Because the radio connectivity graph can be scripted, you can easily store topologies in files and then load the file. Alternatively, you can store a topology as a script. For example, this script will load a file which specifies each link in the graph as a line with three values, the source, the destination, and the gain, e.g.:

```
1  2 -54.0
```

means that when 1 transmits 2 hears it at -54 dBm. Create a file `topo.txt` that looks like this:

```
1  2 -54.0
2  1 -55.0
1  3 -60.0
3  1 -60.0
2  3 -64.0
3  2 -64.0
```

This script will read such a file:

```
>>> f = open("topo.txt", "r")
>>> lines = f.readlines()
>>> for line in lines:
...   s = line.split()
...   if (len(s) > 0):
...     print " ", s[0], " ", s[1], " ", s[2];
...     r.add(int(s[0]), int(s[1]), float(s[2]))
```

Now, when a node transmits a packet, other nodes will hear it. This is a complete script for simulating packet transmission with RadioCountToLedsC. Save it as a file `test.py`:

```
from TOSSIM import *
import sys

t = Tossim([])
r = t.radio()
f = open("topo.txt", "r")

lines = f.readlines()
for line in lines:
  s = line.split()
  if (len(s) > 0):
    print " ", s[0], " ", s[1], " ", s[2];
    r.add(int(s[0]), int(s[1]), float(s[2]))

t.addChannel("RadioCountToLedsC", sys.stdout)
t.addChannel("Boot", sys.stdout)

noise = open("meyer-heavy.txt", "r")
lines = noise.readlines()
for line in lines:
  str = line.strip()
  if (str != ""):
    val = int(str)
    for i in range(1, 4):
      t.getNode(i).addNoiseTraceReading(val)

for i in range(1, 4):
  print "Creating noise model for ",i;
  t.getNode(i).createNoiseModel()

t.getNode(1).bootAtTime(100001);
t.getNode(2).bootAtTime(800008);
t.getNode(3).bootAtTime(1800009);

for i in range(0, 100):
  t.runNextEvent()
```

Run it by typing `python test.py`. You should see output that looks like this:

```
1  2 -54.0
2  1 -55.0
1  3 -60.0
3  1 -60.0
2  3 -64.0
3  2 -64.0
DEBUG (1): Application booted.
DEBUG (1): Application booted again.
DEBUG (1): Application booted a third time.
DEBUG (2): Application booted.
DEBUG (2): Application booted again.
DEBUG (2): Application booted a third time.
DEBUG (3): Application booted.
DEBUG (3): Application booted again.
DEBUG (3): Application booted a third time.
DEBUG (1): RadioCountToLedsC: timer fired, counter is
1.
DEBUG (1): RadioCountToLedsC: packet sent.
DEBUG (2): RadioCountToLedsC: timer fired, counter is
1.
DEBUG (2): RadioCountToLedsC: packet sent.
DEBUG (3): RadioCountToLedsC: timer fired, counter is
1.
DEBUG (3): RadioCountToLedsC: packet sent.
DEBUG (1): Received packet of length 2.
DEBUG (3): Received packet of length 2.
DEBUG (2): Received packet of length 2.
DEBUG (1): RadioCountToLedsC: timer fired, counter is
2.
DEBUG (1): RadioCountToLedsC: packet sent.
DEBUG (2): RadioCountToLedsC: timer fired, counter is
2.
DEBUG (2): RadioCountToLedsC: packet sent.
DEBUG (3): RadioCountToLedsC: timer fired, counter is
2.
DEBUG (3): RadioCountToLedsC: packet sent.
DEBUG (1): Received packet of length 2.
```

If you set node's clear channel assessment to be at -110dBm, then nodes will never transmit, as noise and interference never drop this low. You'll see something like this:

```
1  2 -54.0
2  1 -55.0
```

```
1  3 -60.0
3  1 -60.0
2  3 -64.0
3  2 -64.0
DEBUG (1): Application booted.
DEBUG (1): Application booted again.
DEBUG (1): Application booted a third time.
DEBUG (2): Application booted.
DEBUG (2): Application booted again.
DEBUG (2): Application booted a third time.
DEBUG (3): Application booted.
DEBUG (3): Application booted again.
DEBUG (3): Application booted a third time.
DEBUG (1): RadioCountToLedsC: timer fired, counter is
1.
DEBUG (1): RadioCountToLedsC: packet sent.
DEBUG (2): RadioCountToLedsC: timer fired, counter is
1.
DEBUG (2): RadioCountToLedsC: packet sent.
DEBUG (3): RadioCountToLedsC: timer fired, counter is
1.
DEBUG (3): RadioCountToLedsC: packet sent.
DEBUG (1): RadioCountToLedsC: timer fired, counter is
2.
DEBUG (2): RadioCountToLedsC: timer fired, counter is
2.
DEBUG (3): RadioCountToLedsC: timer fired, counter is
2.
```

Because the nodes backoff perpetually, they never transmit the packet and so subsequent attempts to send fail. Although it only takes a few simulation events to reach the first timer firings, it takes many simulation events (approximately 4000) to reach the second timer firings. This is because the nodes have MAC backoff events. If you want to simulate in terms of time, rather than events, you can always do something like this, which simulates 5 seconds from the first node boot:

```
t.runNextEvent();
time = t.time()
while (time + 50000000000 > t.time()):
  t.runNextEvent()
```

TOSSIM allows you to specify a network topology in terms of gain. However, this raises the problem of coming up with a topology. There are two approaches you can take. The first is to take

data from a real world network and input this into TOSSIM. The second is to generate it from applying a theoretical propagation model to a physical layout. The standard file format is

```
gain src dest g
```

where each statement is on a separate line. The *gain* statement defines a propagation gain *g* when *src* transmits to *dest*. This is a snippet of python code that will parse this file format:

```
f = open("15-15-tight-mica2-grid.txt", "r")

lines = f.readlines()
for line in lines:
  s = line.split()
  if (len(s) > 0):
    if (s[0] == "gain"):
      r.add(int(s[1]), int(s[2]), float(s[3]))
```

TOSSIM has a tool for the second option of generating a network topology using a theoretical propagation model. The tool is written in Java and is `net.tinyos.sim.PropagationModel`. The tool takes a single command line parameter, the name of a configuration file, e.g.:

```
java net.tinyos.sim.PropagationModel config.txt
```

The format of a configuration file is beyond the scope of this document: the tool has its own **documentation**. TOSSIM has a few sample configuration files generated from the tool in `tos/lib/tossim/topologies`. Note that the tool uses random numbers, these configuration files can generate multiple different network topologies. Network topology files generated from the tool follow the same format as `15-15-tight-mica2-grid.txt`. If you have topologies measured from real networks, we would love to include them in the TOSSIM distribution.

Variables

TOSSIM allows you to inspect variables in a running TinyOS program. Currently, you can only inspect basic types. For example, you can't look at fields of structs, but you can look at state variables.

When you compile TOSSIM, the make system generates a large XML file that contains a lot of information about the TinyOS program, including every component variable and its type. If you want to examine the state of your program, then you need to give TOSSIM this information so it can parse all of the variables properly. You do this by instantiating a Python object that parses the

XML file to extract all of the relevant information. You have to import the Python support package for TOSSIM to do this. First, set your PYTHONPATH environment variable to point to `tinyos-2.x/support/sdk/python`. This tells Python where to find the TOSSIM packages. Then, in an interpreter type this:

```
from tinyos.tossim.TossimApp import *

n = NescApp()
```

Instantiating a `NescApp` can take quite a while: Python has to parse through megabytes of XML. So be patient (you only have to do it once). NescApp has two optional arguments. The first is the name of the application being loaded. The second is the XML file to load. The default for the latter is `app.xml`, which is the name of the file that the make system generates. The default for the former is "Unknown App." So this code behaves identically to that above:

```
from tinyos.tossim.TossimApp import *

n = NescApp("Unknown App", "app.xml")
```

You fetch a list of variables from a NescApp object by calling the function `variables` on the field `variables`:

```
vars = n.variables.variables()
```

To enable variable inspection, you pass this list to a Tossim object when you instantiate it:

```
t = Tossim(vars)
```

The TOSSIM object now knows the names, sizes, and types of all of the variables in the TinyOS application. This information allows the TOSSIM support code to take C variables and properly tranform them into Python variables. This currently only works for simple types: if a component declares a structure, you can't access its fields. But let's say we want to read the counter in `RadioCountToLedsC`. Since each mote in the network has its own instance of the variable, we need to fetch it from a specific mote:

```
m = t.getNode(0)
v = m.getVariable("RadioCountToLedsC.counter")
```

The name of a variable is usually *C.V*, where *C* is the component name and *V* is the variable. In the case of generic components, the name is *C.N.V*, where *N* is an integer that describes which

instance. Unfortunately, there is currently no easy way to know what *N* is from nesC source, so you have to root through `app.c` in order to know.

Once you have a variable object (`v` in the above code), you can fetch its value with the `getData()` function:

```
counter = v.getData()
```

Because `getData()` transforms the underlying C type into a Python type, you can then use its return value in Python expressions. For example, this script will start a simulation of five nodes and run it until node 0's counter reaches 10:

```
from sys import *
from random import *
from TOSSIM import *
from tinyos.tossim.TossimApp import *

n = NescApp()
t = Tossim(n.variables.variables())
r = t.radio()

f = open("topo.txt", "r")
lines = f.readlines()
for line in lines:
  s = line.split()
  if (len(s) > 0):
    if (s[0] == "gain"):
      r.add(int(s[1]), int(s[2]), float(s[3]))

noise = open("meyer-heavy.txt", "r")
lines = noise.readlines()
for line in lines:
  str = line.strip()
  if (str != ""):
    val = int(str)
    for i in range(0, 4):
      t.getNode(i).addNoiseTraceReading(val)

for i in range (0, 4):
  t.getNode(i).createNoiseModel()
  t.getNode(i).bootAtTime(i * 2351217 + 23542399)

m = t.getNode(0)
```

```
v = m.getVariable("RadioCountToLedsC.counter")



while (v.getData() < 10):
  t.runNextEvent()

print "Counter variable at node 0 reached 10."
```

The TOSSIM **examples** subdirectory also has an example script, named `variables.py`.

Injecting Packets

TOSSIM allows you to dynamically inject packets into a network. Packets can be scheduled to arrive at any time. If a packet is scheduled to arrive in the past, then it arrives immediately. Injected packets circumvent the radio stack: it is possible for a node to receive an injected packet while it is in the midst of receiving a packet from another node over its radio.

TinyOS 2.0 has support for building Python packet objects. Just like the standard Java toolchain, you can build a packet class based on a C structure. The packet class gives you a full set of packet field mutators and accessors. If an application has a packet format, you can generate a packet class for it, instantiate packet objects, set their fields, and have nodes receive them.

The `RadioCountToLeds` application Makefile has an example of how to do this. First, it adds the Python class as a dependency for building the application. Whenever you compile the app, if the Python class doesn't exist, make will build it for you:

```
BUILD_EXTRA_DEPS = RadioCountMsg.py
RadioCountMsg.class
```

The Makefile also tells make how to generate RadioCountMsg.py:

```
RadioCountMsg.py: RadioCountToLeds.h
        mig python -target=$(PLATFORM) $(CFLAGS) -
python-classname=RadioCountMsg RadioCountToLeds.h
RadioCountMsg -o $@
```

The rule says to generate RadioCountMsg.py by calling mig with the python parameter. The Makefile also has rules on how to build Java class, but that's not important for TOSSIM. Since we've been using RadioCountToLeds so far, the Python class should be there already.

RadioCountMsg.py defines a packet format, but this packet is contained in the data payload of another format. If a node is sending a `RadioCountMsg` over AM, then the `RadioCountMsg` structure is put into the AM payload, and might look something like this:

If it is sending it over a routing protocol. the packet is put in the routing payload, and might look something like this:

If you want to send a `RadioCountMsg` to a node, then you need to decide how to deliver it. In the simple AM case, you place the `RadioCountMsg` structure in a basic AM packet. In the routing case, you put it in a routing packet, which you then put inside an AM packet. We'll only deal with the simple AM case here.

To get an AM packet which you can inject into TOSSIM, you call the `newPacket` function on a Tossim object. The returned object has the standard expected AM fields: *destination*, *length*, *type*, and *data*, as well as *strength*.

To include support for a packet format, you must import it. For example, to include `RadioCountMsg`, you have to import it:

```
from RadioCountMsg import *
```

This snippet of code, for example, creates a `RadioCountMsg`, sets its counter to 7, creates an AM packet, stores the `RadioCountMsg` in the AM packet, and configures the AM packet so it will be received properly (destination and type):

```
from RadioCountMsg import *

msg = RadioCountMsg()
msg.set_counter(7);
pkt = t.newPacket();
pkt.setData(msg.data)
pkt.setType(msg.get_amType())
pkt.setDestination(0)
```

The variable `pkt` is now an Active Message of the AM type of `RadioCountMsg` with a destination of 0 that contains a `RadioCountMsg` with a counter of 7. You can deliver this packet to a node with the `deliver` function. The `deliver` function takes two parameters, the destination node and the time to deliver:

```
pkt.deliver(0, t.time() + 3)
```

This call delivers pkt to node 0 at the current simulation time plus 3 ticks (e.g., 3ns). There is also a `deliverNow`, which has no time parameter. Note that if the destination of `pkt` had been set to 1, then the TinyOS application would not receive the packet, as it was delivered to node 0.

Taken all together, the following script starts a simulation, configures the topology based on topo.txt, and delivers a packet to node 0. It can also be found as `packets.py` in the TOSSIM **examples** subdirectory.

```
import sys
from TOSSIM import *
from RadioCountMsg import *

t = Tossim([])
m = t.mac();
r = t.radio();

t.addChannel("RadioCountToLedsC", sys.stdout);
t.addChannel("LedsC", sys.stdout);

for i in range(0, 2):
  m = t.getNode(i);
  m.bootAtTime((31 + t.ticksPerSecond() / 10) * i +
1);

f = open("topo.txt", "r")
lines = f.readlines()
for line in lines:
  s = line.split()
  if (len(s) > 0):
    if (s[0] == "gain"):
      r.add(int(s[1]), int(s[2]), float(s[3]))

noise = open("meyer-heavy.txt", "r")
lines = noise.readlines()
for line in lines:
  str = line.strip()
  if (str != ""):
    val = int(str)
    for i in range(0, 4):
      t.getNode(i).addNoiseTraceReading(val)
```

```
for i in range (0, 4):
  t.getNode(i).createNoiseModel()

for i in range(0, 60):
  t.runNextEvent();

msg = RadioCountMsg()
msg.set_counter(7);
pkt = t.newPacket();
pkt.setData(msg.data)
pkt.setType(msg.get_amType())
pkt.setDestination(0)

print "Delivering " + str(msg) + " to 0 at " +
str(t.time() + 3);
pkt.deliver(0, t.time() + 3)


for i in range(0, 20):
  t.runNextEvent();
```

C++

Python is very useful because it is succinct, easy to write, and can be used interactively. Interpretation, however, has a significant cost: a Python/C transition on every event is a significant cost (around 100%, so it runs at half the speed). Additionally, it's often useful to step through code with a standard debugger. TOSSIM also has support for C++, so that it can be useful in these circumstances. Because many of the Python interfaces are merely wrappers around C++ objects, much of the scripting stays the same. The two major exceptions are inspecting variables and injecting packets.

In a C++ TOSSIM, there is no variable inspection. While it is possible to request memory regions and cast them to the expected structures, currently there is no good and simple way to do so. The Python support goes through several steps in order to convert variables into Python types, and this gets in the way of C++. However, as the purpose of C++ is usually to run high performance simulations (in which inspecting variables is a big cost) or debugging (when you have a debugger), this generally isn't a big problem.

There is a C++ `Packet` class, which the Python version is a simple wrapper around. In order to inject packets in C++, however, you must build C support for a packet type and manually build the packet. There currently is no support in mig with which to generate C/C++ packet structures, and since most packets are nx_struct types, they cannot be parsed by C/C++. Furthermore, as

many of the fields are nx types, they are big endian, while x86 processors are little endian. Still, if you want to deliver a packet through C++, you can do so.

Usually, the C++ and Python versions of a program look pretty similar. For example (note that this program will use a lot of RAM and take a long time to start due to its noise models):

Python

```
import TOSSIM
import sys
import random

from RadioCountMsg import *

t = TOSSIM.Tossim([])
r = t.radio();

for i in range(0, 999):
  m = t.getNode(i);
  m.bootAtTime(5000003 * i + 1);

  for j in range (0, 2):
    if (j != i):
      r.add(i, j, -50.0);

  # Create random noise stream
  for j in range (0, 500):

m.addNoiseTraceReading(int(random.rando
m() * 20) - 70);
  m.createNoiseModel()

for i in range(0, 1000000):
  t.runNextEvent();
```

C++

```
#include <tossim.h>
#include <stdlib.h>


int main() {
  Tossim* t = new Tossim(
  Radio* r = t->radio();

  for (int i = 0; i < 999
    Mote* m = t->getNode(
    m->bootAtTime(500000
    for (int j = 0; j < 2
      if (i != j) {
        r->add(i, j, -50
      }
    }
    for (int j = 0; i < 5
       m->addNoiseTraceRe
(drand48() * 20) - 70);
    }
    m->createNoiseModel()
  }


  for (int i = 0; i < 100
    t->runNextEvent();
  }
}
```

To compile a C++ TOSSIM, you have to compile the top-level driver program (e.g, the one shown above) and link it against TOSSIM. Usually the easiest way to do this is to link it against the TOSSIM objects rather than the shared library. Often, it's useful to have a separate Makefile to do this with. E.g., `Makefile.Driver`:

```
all:
```

```
        make micaz sim
        g++ -g -c -o Driver.o Driver.c
-I../../tos/lib/tossim/
        g++ -o Driver Driver.o build/micaz/tossim.o
build/micaz/sim.o build/micaz/c-support.o
```

Using gdb

Since Driver is a C++ program, you can use gdb on it to step through your TinyOS code, inspect variables, set breakpoints, and do everything else you can normally do. Unfortunately, as gdb is designed for C and not nesC, the component model of nesC means that a single command can have multiple providers; referring to a specific command requires specifying the component, interface, and command. For example, to break on entry to the redOff command of the Leds interface of LedsC, one must type:

```
$ gdb Driver
GNU gdb Red Hat Linux (6.0post-0.20040223.19rh)
Copyright 2004 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General
Public License, and you are
welcome to change it and/or distribute copies of it
under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show
warranty" for details.
This GDB was configured as "i386-redhat-linux-
gnu"...Using host libthread_db library
"/lib/tls/libthread_db.so.1".

(gdb) break *LedsP$Leds$led0Toggle
Breakpoint 1 at 0x804f184: file LedsP.nc, line 73.
```

nesC translates component names to C names using $. $ is a legal but almost-never-used character in some versions of C, so nesC prohibits it and uses it internally. The leading * is necessary so dbg can parse the $s. With the above breakpoint set, gdb will break whenever a mote toggles led0.

Variables have similar names. For example, to inspect the packet of RadioCountToLedsC in the RadioCountToLeds application,

```
(gdb) print RadioCountToLedsC$packet
$1 = {{header = {{data = ""}, {data = ""}, {data =
""}, {data = ""}, {
```

```
        data = ""}}, data = {{data = ""} }, footer =
{{
        data = ""}, {data = ""}}, metadata = {{data =
""}, {data = ""}, {
        data = ""}, {data = ""}, {data = ""}}} }
```

For those who know gdb very well, you'll recognize this as a print of an array, rather than a single variable: there are more than 1000 instances of the message_t struct. This is because TOSSIM simulates many motes; rather than there being a single RadioCountToLedsC$packet, there is one for every node. To print the packet of a specific node, you have to index into the array. This, for example, will print the variable for node 6:

```
(gdb) print RadioCountToLedsC$packet[6]
$2 = {header = {{data = ""}, {data = ""}, {data = ""},
{data = ""}, {
      data = ""}}, data = {{data = ""} }, footer = {{
      data = ""}, {data = ""}}, metadata = {{data =
""}, {data = ""}, {
      data = ""}, {data = ""}, {data = ""}}}
```

If you want to print out the variable for the node TOSSIM is currently simulating, you can do this:

```
(gdb) print RadioCountToLedsC$counter[sim_node()]
$4 = 0
```

You can also set watchpoints (although, as to be expected, they are *slow*:

```
(gdb) watch CpmModelC$receiving[23]
Hardware watchpoint 2: CpmModelC$receiving[23]
```

This variable happens to be an internal variable in the packet-level network simulation, which keeps track of whether the radio thinks it is receiving a packet. So setting the above watchpoint will cause gdb to break whenever node 23 starts receiving a packet or returns to searching for packet preambles.

Generic components add another wrinkle. Since they use a code-copying approach, each instance of a generic has its own separate functions and variables (this is mostly due to the fact that you can pass types to them). Take, for example, `AMQueueImplP`, which is used in both the radio AM stack and the serial AM stack. If you use gdb on an application that uses both serial and radio communication and try to break on its Send.send, you'll see an error:

```
(gdb) break *AMQueueImplP$Send$send
No symbol "AMQueueImplP$Send$send" in current context.
```

nesC gives each generic a unique number. So if you have an application in which there is a single copy of AMQueueImplP, its name will actually be AMQueueImplP$0. For example, in RadioCountToLeds, this will work:

```
(gdb) break *AMQueueImplP$0$Send$send
Breakpoint 5 at 0x8051b29: file AMQueueImplP.nc, line
79.
```

If you have multiple instances of a generic in a program, there is unfortunately no easy way to figure out each one's name besides looking at the source code or stepping into them. E.g., if you application uses serial and radio communication, knowing which stack has AMQueueImpl$0 and which has AMQueueImplP$1 requires either stepping through their send operation or looking at their `app.c` files.

Conclusions

This lesson introduced the basics of the TOSSIM simulator. It showed you how to configure a network, how to run a simulation, how to inspect variables, how to inject packets, and how to compile with C++.

< Previous Lesson | Top | Next Lesson >

Appendix A: Troubleshooting TOSSIM compilation

TOSSIM is a C/C++ shared library with an optional Python translation layer. Almost all of the problems encountered in compiling TOSSIM are due to C linking issues. If you don't know what a linker is (or have never linked a C program), then chances are the rest of this appendix is going to be cryptic and incomprehensible. You're best off starting with learning about **linkers**, **why they are needed**, and how you **use the gcc/g++ compilers** to link code.

Generally, when compiling TOSSIM using `make micaz sim`, one of four things can go wrong:

1. You are using Cygwin but the `sim` compilation option can't figure this out.
2. You do not have the needed Python support installed.
3. You have Python support installed, but the make system can't find it.
4. You have Python support installed, but it turns out to be incompatible with TOSSIM.
5. You have a variant of gcc/g++ installed that expects slightly different compilation options than the normal installation.

We'll visit each in turn.

You are using Cygwin but the sim compilation option can't figure this out

It turns out that the Cygwin and Linux versions of gcc/g++ have different command-line flags and require different options to compile TOSSIM properly. For example, telling the Linux compiler to build a library requires -fPIC while the Cygwin is -fpic. If you're using Cygwin and you see the output look like this:

```
  ncc -c -shared -fPIC -o build/micaz/sim.o ...
```

rather than

```
  ncc -c -DUSE_DL_IMPORT -fpic -o
build/micaz/sim.o ...
```

then you have encountered this problem. The problem occurs because Cygwin installations do not have a consistent naming scheme, and so it's difficult for the compilation toolchain to always figure out whether it's under Linux or Cygwin.

Symptom: You're running cygwin but you see the -fPIC rather than -fpic option being passed to the compiler.

Solution: Explicitly set the OSTYPE environment variable to be cygwin either in your .bashrc or when you compile. For example, in bash:

```
$ OSTYPE=cygwin make micaz sim
```

or in tcsh

```
$ setenv OSTYPE cygwin
$ make micaz sim
```

Note that often this problem occurs in addition to other ones, due to using a nonstandard Cygwin installation. So you might have more problems to track down.

You do not have the needed Python support installed

If when you compile you see lots of errors such as "undefined reference to" or "Python.h: No such file or directory" then this might be your problem. It is a subcase of the more general problem of TOSSIM not being able to find needed libraries and files.

Compiling Python scripting support requires that you have certain Python development libraries installed. First, check that you have Python installed:

```
$ python -V
Python 2.4.2
```

In the above example, the system has Python 2.4.2. If you see "command not found" then you do not have Python installed. You'll need to track down an RPM and install it. TOSSIM has been tested with Python versions 2.3 and 2.4. You can install other versions, but there's no assurance things will work.

In addition to the Python interpreter itself, you need the libraries and files for Python development. This is essentially a set of header files and shared libraries. If you have the `locate` command, you can type `locate libpython`, or if you don't, you can look in `/lib`, `/usr/lib` and `/usr/local/lib`. You're looking for a file with a name such as `libpython2.4.so` and a file named `Python.h`. If you can't find these files, then you need to install a `python-devel` package.

Symptom: Compilation can't find critical files such as the Python interpreter, `Python.h` or a Python shared library, and searching your filesystem shows that you don't have them.

Solution: Installed the needed files from Python and/or Python development RPMS.

If you have all of the needed files, but are still getting errors such as "undefined reference" or "Python.h: No such file or directory", then you have the next problem: they're on your filesystem, but TOSSIM can't find them.

You have Python support installed, but the make system can't find it

You've found libpython and Python.h, but when TOSSIM compiles it says that it can't find one or both of them. If it can't find Python.h then compilation will fail pretty early, as g++ won't be able to compile the Python glue code. If it can't find the python library, then compilation will fail at linking, and you'll see errors along the lines of "undefined reference to __Py...". You need to point the make system at the right place.

Open up `support/make/sim.extra`. If the make system can't find Python.h, then chances are it isn't in one of the standard places (e.g., /usr/include). You need to tell the make system to look in the directory where Python.h is with a `-I` option. At the top of sim.extra, under the PFLAGS entry, add

```
CFLAGS += -I/path
```

where /path is the path of the directory where Python.h lives. For example, if it is in /opt/python/include, then add CFLAGS += -I/opt/python/include.

If the make system can't find the python library for linking (causing "undefined reference") error messages, then you need to make sure the make system can find it. The sim.extra file uses two variables to find the library: PYDIR and PYTHON_VERSION. It looks for a file named libpython$(PYTHON_VERSION).so. So if you have Python 2.4 installed, make sure that PYTHON_VERSION is 2.4 (be sure to use no spaces!) and if 2.3, make sure it is 2.3.

Usually the Python library is found in /usr/lib. If it isn't there, then you will need to modify the PLATFORM_LIB_FLAGS variable. The -L flag tells gcc in what directories to look for libraries. So if libpython2.4.so is in /opt/python/lib, then add -L/opt/python/lib to the PLATFORM_LIB_FLAGS. Note that there are three different versions of this variable, depending on what OS you're using. Be sure to modify the correct one (or be paranoid and modify all three).

Symptom: You've verified that you have the needed Python files and libraries, but compilation is still saying that it can't link to them ("undefined reference") or can't find them ("cannot find -lpython2.4").

Solution: Change the sim.extra file to point to the correct directories using -L and -I flags.

You have Python support installed, but it turns out to be incompatible with TOSSIM.

Symptom: You get a "This python version requires to use swig with the -classic option" error message.

Solution: Install SWIG and regenerate Python support with the sing-generate script in tos/lib/tossim, or install a different version of Python.

You have a variant of gcc/g++ installed that expects slightly different compilation options than the normal installation.

Symptom: g++ complains that it cannot find main() when you are compiling the shared library ("undefined reference to `_WinMain@16'").

Solution: There are two possible solutions. The first is to include a dummy main(), as described in this **tinyos-help posting.** The second is to add the -shared option, as described in this **web page**.

Hopefully, these solutions worked and you can get back to **compiling**, If not, then you should email tinyos-help.

Lesson 12: Network Protocols

Last updated Nov 3 2006

This lesson introduces the two basic network primitives of Tinyos-2: Dissemination and Collection.

Dissemination

The goal of a dissemination protocol is to reliably deliver a piece of data to every node in the network. It allows administrators to reconfigure, query, and reprogram a network. Reliability is important because it makes the operation robust to temporary disconnections or high packet loss. Dissemination is fully explained in TEP 118.

In TinyOS 2.x, dissemination provides two interfaces: DisseminationValue and DisseminationUpdate. Let's take a look at these two interfaces:

tos/lib/net/DisseminationUpdate.nc:

```
interface DisseminationUpdate<t> {
  command void change(t* newVal);
}
```

tos/lib/net/DisseminationValue.nc:

```
interface DisseminationValue<t> {
  command const t* get();
  event void changed();
}
```

DisseminationUpdate is used by producers. The command ***DisseminationUpdate.change()*** should be called each time the producer wants to disseminate a new value, passing this new value as a parameter.
DisseminationValue is for consumers. The event ***DisseminationValue.changed()*** is signalled each time the disseminated value is changed (the producer has called ***change***), and the command ***get*** allows to obtain this new value.

We build now a simple application (EasyDissemination) where one node (the producer) periodically disseminates the value of a counter to rest of the nodes in the network (consumers).

As a first step, create a new directory in `apps` named `EasyDissemination`:

```
$ cd tinyos-2.x/apps
$ mkdir EasyDissemination
```

Inside this directory, create a file `EasyDisseminationC.nc`, which has this code:

```
#include <Timer.h>

module EasyDisseminationC {
  uses interface Boot;
  uses interface DisseminationValue<uint16_t> as
Value;
  uses interface DisseminationUpdate<uint16_t> as
Update;
  uses interface Leds;
  uses interface Timer<TMilli>;
}

implementation {

  uint16_t counter;

  task void ShowCounter() {
    if (counter & 0x1)
      call Leds.led0On();
    else
      call Leds.led0Off();
    if (counter & 0x2)
      call Leds.led1On();
    else
      call Leds.led1Off();
    if (counter & 0x4)
      call Leds.led2On();
    else
      call Leds.led2Off();
  }

  event void Boot.booted() {
    counter = 0;
    if ( TOS_NODE_ID  == 1 )
      call Timer.startPeriodic(2000);
  }

  event void Timer.fired() {
```

```
    counter = counter + 1;
    // show counter in leds
    post ShowCounter();
    // disseminate counter value
    call Update.change(&counter);
  }

  event void Value.changed() {
    const uint16_t* newVal = call Value.get();
    // show new counter in leds
    counter = *newVal;
    post ShowCounter();
  }
}
```

We assume that the base station is the node with ID = 1. First note that the base station will periodically (every 2 seconds) increment a 3-bit counter, display the counter using its three leds, and disseminate it through the network. This is done using the change command provided in the DisseminationUpdate interface:

```
    call Update.change(&counter);
```

Second, note that when a node receives a change notification, it updates its counter value and shows it on the leds:

```
  event void Value.changed() {
    const uint16_t* newVal = call Value.get();
    // show new counter in leds
    counter = *newVal;
    post ShowCounter();
  }
```

The `EasyDisseminationAppC.nc` provides the needed wiring:

```
configuration EasyDisseminationAppC {}
implementation {
  components EasyDisseminationC;

  components MainC;
  EasyDisseminationC.Boot -> MainC;

  components new DisseminatorC(uint16_t, 0x1234) as
Diss16C;
  EasyDisseminationC.Value -> Diss16C;
  EasyDisseminationC.Update -> Diss16C;

  components LedsC;
  EasyDisseminationC.Leds -> LedsC;

  components new TimerMilliC();
```

```
  EasyDisseminationC.Timer -> TimerMilliC;
}
```

Note that both Dissemination interfaces we use are provided by the module DisseminatorC. This module provides the Dissemination service:

tos/lib/net/Dissemination/DisseminationC.nc:

```
 generic configuration DisseminatorC(typedef t,
uint16_t key) {
  provides interface DisseminationValue<t>;
  provides interface DisseminationUpdate<t>;
}
```

Note that we need to specify to the Disseminartor module a type t and a key. In our case, the value we want to disseminate is just an unsigned two-byte counter. The key allows to have different instances of DisseminatroC.

To compile this program we use create the following Makefile:

```
COMPONENT=EasyDisseminationAppC
CFLAGS += -I$(TOSDIR)/lib/net

include $(MAKERULES)
```

Now install this program into several nodes (make sure you have one base station, that is, one node whose ID is 1) and see how the counter displayed in the base station is "disseminated" to all the nodes belonging to the network. You will also notice that dissemination works across resets, i.e., if you reset a node it will rapidly re-'synchronize' and display the correct value after it reboots.

For more information, read TEP118 [Dissemination].

Collection

Collection is the complementary operation to disseminating and it consists in "collecting" the data generated in the network into a base stations. The general approach used is to build one or more collection *trees*, each of which is rooted at a base station. When a node has data which needs to be collected, it sends the data up the tree, and it forwards collection data that other nodes send to it.

We build now a simple application (EasyCollection) where nodes periodically send information to a base station which collects all the data.
As a first step, create a new directory in `apps` named `EasyCollection`:

```
$ cd tinyos-2.x/apps
$ mkdir EasyCollection
```

Inside this directory, create a file `EasyCollectionC.nc`, which has the following code:

```
#include <Timer.h>

module EasyCollectionC {
  uses interface Boot;
  uses interface SplitControl as RadioControl;
  uses interface StdControl as RoutingControl;
  uses interface Send;
  uses interface Leds;
  uses interface Timer<TMilli>;
  uses interface RootControl;
  uses interface Receive;
}
implementation {
  message_t packet;
  bool sendBusy = FALSE;

  typedef nx_struct EasyCollectionMsg {
    nx_uint16_t data;
  } EasyCollectionMsg;

  event void Boot.booted() {
    call RadioControl.start();
  }

  event void RadioControl.startDone(error_t err) {
    if (err != SUCCESS)
      call RadioControl.start();
    else {
      call RoutingControl.start();
      if (TOS_NODE_ID == 1)
   call RootControl.setRoot();
      else
   call Timer.startPeriodic(2000);
    }
  }

  event void RadioControl.stopDone(error_t err) {}

  void sendMessage() {
    EasyCollectionMsg* msg = (EasyCollectionMsg*)call
Send.getPayload(&packet);
    msg->data = 0xAAAA;
```

```
    if (call Send.send(&packet,
sizeof(EasyCollectionMsg)) != SUCCESS)
      call Leds.led0On();
    else
      sendBusy = TRUE;
  }
  event void Timer.fired() {
    call Leds.led2Toggle();
    if (!sendBusy)
      sendMessage();
  }

  event void Send.sendDone(message_t* m, error_t err)
{
    if (err != SUCCESS)
      call Leds.led0On();
    sendBusy = FALSE;
  }

  event message_t*
  Receive.receive(message_t* msg, void* payload,
uint8_t len) {
    call Leds.led1Toggle();
    return msg;
  }
}
```

Lets take a look at this program. First note that all nodes turn on the radio into the Boot sequence:

```
  event void Boot.booted() {
    call RadioControl.start();
  }
```

Once we are sure that the radio is on, we start the routing sub-system (that is, to generate the collection *tree*):

```
call RoutingControl.start();
```

Next we need need to specify the root of the collection tree, that is, the node that will receive all the data packets. For this, we use the interface RootControl:

tos/lib/net/RootControl.nc

```
interface RootControl {
    command error_t setRoot();
    command error_t unsetRoot();
```

```
    command bool isRoot();
}
```

This interface controls whether the current node is a root of the tree. Using the setRoot() command and assuming that the base station ID is 1, we select the root of the collection *tree* as follows:

```
if (TOS_NODE_ID == 1)
    call RootControl.setRoot();
else
    call Timer.startPeriodic(2000);
```

The remaining nodes in the network periodically generate some data and send it to the base station. To send and receive data we use two interfaces that will be wired to the collection tree. That is, when we call the send command, the data packet will be sent through the collection tree. Similarly, the receive event will be only called in the root of the tree, that is, in the base station. When the base station receives a "collected" packet it just toggle a led. Now we will see how to wire these interfaces .

The `EasyCollectionAppC.nc` provides the needed wiring:

```
configuration EasyCollectionAppC {}
implementation {
  components EasyCollectionC, MainC, LedsC,
ActiveMessageC;
  components CollectionC as Collector;
  components new CollectionSenderC(0xee);
  components new TimerMilliC();

  EasyCollectionC.Boot -> MainC;
  EasyCollectionC.RadioControl -> ActiveMessageC;
  EasyCollectionC.RoutingControl -> Collector;
  EasyCollectionC.Leds -> LedsC;
  EasyCollectionC.Timer -> TimerMilliC;
  EasyCollectionC.Send -> CollectionSenderC;
  EasyCollectionC.RootControl -> Collector;
  EasyCollectionC.Receive -> Collector.Receive[0xee];
}
```

Most of the collection interfaces (RoutingControl, RootControl and Receive) are provided by the CollectionC module. The send interface is provided by CollectionSenderC which is a virtualized collection sender abstraction module.

This is an extract of the signature of the CollectionC module and CollectionSenderC:
tos/lib/net/ctp/CollectionC.nc

```
configuration CollectionC {
  provides {
    interface StdControl;
    interface Send[uint8_t client];
    interface Receive[collection_id_t id];
    interface Receive as Snoop[collection_id_t];
    interface Intercept[collection_id_t id];

    interface Packet;
    interface CollectionPacket;
    interface CtpPacket;

    interface CtpInfo;
    interface CtpCongestion;
    interface RootControl;
  }
```

tos/lib/net/ctp/CollectionSenderC:

```
  generic configuration
CollectionSenderC(collection_id_t collectid) {
  provides {
    interface Send;
    interface Packet;
  }
```

Note that the sender and receive interfaces requires a collection_id_t to differentiate different possible collections trees.
Note also that the CollectionC module provides some other interfaces in addition to the ones used in this example. As we explained previously, the CollectionC module generates a collection tree that will be using for the routing. These interfaces can be used get information or modify this routing tree. For instance, if we want to obtain information about this tree we use the CtpInfo interface (see tos/lib/net/ctp/CtpInfo.nc) and if we want to indicate/query if any node/sink is congested we use the CtpCongestion interface (see tos/lib/net/ctp/CtpCongestion.nc)

Finally, to compile this program we create the following Makefile:

```
COMPONENT=EasyCollectionAppC
CFLAGS += -I$(TOSDIR)/lib/net \
          -I$(TOSDIR)/lib/net/le \
          -I$(TOSDIR)/lib/net/ctp
include $(MAKERULES)
```

Now install this program into several nodes (make sure you have one base station, that is, one node whose ID is 1) and see how all the packets generated in the nodes are collected in the base

station.

For more information, read TEP119 [Collection].

To experiment further

If you want to experiment with a more complex application take a look at apps/test/TestNetwork/ which combines dissemination and collection into a single application.

Related Documentation

- **TEP 118: Dissemination**
- **TEP 119: Collection**

Lesson 13: TinyOS Toolchain

Last updated October 29 2006

This lesson describes the details of the TinyOS toolchain, including the build system, how to create your own Makefile, and how to find out more information on the various tools included with TinyOS.

TinyOS Build System

As you saw in **Lesson 1**, TinyOS applications are built using a somewhat unconventional application of the *make* tool. For instance, in the `apps/Blink` directory,

```
$ make mica2
```

compiles Blink for the mica2 platform,

```
$ make mica2 install
```

compiles and installs (using the default parallel port programmer) Blink for the mica2, and

```
$ make mica2 reinstall mib510,/dev/ttyS0
```

installs the previously compiled mica2 version of Blink using the MIB510 serial port programmer connected to serial port /dev/ttyS0.

As these examples show, the TinyOS build system is controlled by passing arguments to make that specify the target platform, the desired action, and various options. These arguments can be categorised as follows:

- Target platform: one of the supported TinyOS platforms, e.g., **mica2**, **telosb**, **tinynode**. A target platform is always required, except when using the **clean** action.

- Action: the action to perform. By default, the action is to compile the application in the current directory, but you can also specify:
  - **help**: display a help message for the target platform.
  - **install,*N***: compile and install. The *N* argument is optional and specifies the mote id (default 1).
  - **reinstall,*N***: install only (fails if the application wasn't previously compiled). *N* is as for **install**.
  - **clean**: remove compiled application for all platforms.
  - **sim**: compile for the simulation environment for the specified platform (see Lesson 11 for details).

  Example: to compile for simulation for the micaz:

  ```
  $ make micaz sim
  ```
- Compilation option: you can change the way compilation proceeds by specifying:
  - **debug**: compile for debugging. This enables debugging, and turns off optimisations (e.g., inlining) that make debugging difficult.
  - **debugopt**: compile for debugging, but leave optimisations enabled. This can be necessary if compiling with **debug** gives code that is too slow, or if the bug only shows up when optimisation is enabled.
  - **verbose**: enable a lot of extra output, showing all commands executed by *make* and the details of the nesC compilation including the full path of all files loaded. This can be helpful in tracking down problems (e.g., when the wrong version of a component is loaded).
  - **wiring**, **nowiring**: enable or disable the use of the nescc-wiring to check the wiring annotations in a nesC program. See the nescc-wiring man page for more details.

  Example: to do a verbose compilation with debugging on the telosb:

  ```
  $ make debug verbose telosb
  ```

  Additionally, you can pass additional compilation options by setting the CFLAGS environment variable when you invoke make. For instance, to compile `apps/RadioCountoToLeds` for a mica2 with a 900MHz radio set to ~916.5MHz, you would do:

  ```
  $ env CFLAGS="-DCC1K_DEF_FREQ=916534800" make mica2
  ```

  Note that this will not work with applications whose Makefile defines CFLAGS (but this practice is discouraged, see the section on **writing Makefiles** below).

- Installation option: some platforms have multiple programmers, and some programmers require options (e.g., to specify which serial port to use). The programmer is specified by including its name amongst the *make* arguments. Known programmers include **bsl** for msp430-based platforms and **avrisp** (STK500), **dapa** (MIB500 and earlier), **mib510** (MIB510) and **eprb** (MIB600) for mica family motes.

  Arguments to the programmer are specified with a comma after the programmer name, e.g.,

  ```
  $ make mica2dot reinstall mib510,/dev/ttyUSB1
  $ make telosb reinstall bsl,/dev/ttyUSB1
  ```

  to specify that the programmer is connected to serial port /dev/ttyUSB1.

  More details on the programmers and their options can be found in your mote documentation.

Customising the Build System

You may find that you are often specifying the same options, e.g., that your mib510 programmer is always connected to /dev/ttyS1 or that you want to use channel 12 of the CC2420 radio rather than the default TinyOS 2 channel (26). To do this, put the following lines

```
MIB510 ?= /dev/ttyS1
PFLAGS = -DCC2420_DEF_CHANNEL=12
```

in a file called `Makelocal` in the `support/make` directory. If you now compile in `apps/RadioCountToLeds`, you will see:

```
$ make micaz install mib510
    compiling RadioCountToLedsAppC to a micaz binary
ncc -o build/micaz/main.exe -Os -
DCC2420_DEF_CHANNEL=12 ... RadioCountToLedsAppC.nc -lm
    compiled RadioCountToLedsAppC to
build/micaz/main.exe
    ...
    installing micaz binary using mib510
uisp -dprog=mib510 -dserial=/dev/ttyS1 ...
```

The definition of `PFLAGS` passes an option to the nesC compiler telling it to define the C preprocessor symbol `CC2420_DEF_CHANNEL` to 12. The CC2420 radio stack checks the value of this symbol when setting its default channel.

The definition of `MIB510` sets the value of the argument to the **mib510** installation option, i.e.,

```
$ make micaz install mib510
```

is now equivalent to

```
$ make micaz install mib510,/dev/ttyS1
```

Note that the assignment to MIB510 was written using the `?=` operator. If you just use regular assignment (`=`), then the value in `Makelocal` will override any value you specify on the command line (which is probably not what you want...).

`Makelocal` can contain definitions for any *make* variables used by the build system. Unless you understand the details of how this works, we recommend you restrict yourselves to defining:

- `PFLAGS`: extra options to pass to the nesC compiler. Most often used to define preprocessor symbols as seen above.
- *X*: set the argument for *make* argument *x*, e.g., `MIB510` as seen above. You can, e.g., set the default mote id to 12 by adding `INSTALL ?= 12` and `REINSTALL ?= 12` to `Makelocal`.

Some useful preprocessor symbols that you can define with `PFLAGS` include:

- DEFINED_TOS_AM_ADDRESS: the motes group id (default is 0x22).
- CC2420_DEF_CHANNEL: CC2420 channel (default is 26).
- CC1K_DEF_FREQ: CC1000 frequency (default is 434.845MHz).
- TOSH_DATA_LENGTH: radio packet payload length (default 28).

Application Makefiles

To use the build system with your application, you must create a makefile (a file called `Makefile`) which contains at the minimum:

```
COMPONENT=TopLevelComponent
include $(MAKERULES)
```

where *TopLevelComponent* is the name of the top-level component of your application.

TinyOS applications commonly also need to specify some options to the nesC compiler, and build some extra files alongside the TinyOS application. We will see examples of both, by looking at, and making a small change to, the `apps/RadioCountToLeds` application.

The RadioCountToLeds Makefile uses `mig` (see **Lesson 4**) to build files describing the layout of its messages, for use with python and Java tools:

```
COMPONENT=RadioCountToLedsAppC
BUILD_EXTRA_DEPS = RadioCountMsg.py
RadioCountMsg.class

RadioCountMsg.py: RadioCountToLeds.h
```

```
	mig python -target=$(PLATFORM) $(CFLAGS) -python-
classname=RadioCountMsg RadioCountToLeds.h
RadioCountMsg -o $@

RadioCountMsg.class: RadioCountMsg.java
	javac RadioCountMsg.java

RadioCountMsg.java: RadioCountToLeds.h
	mig java -target=$(PLATFORM) $(CFLAGS) -java-
classname=RadioCountMsg RadioCountToLeds.h
RadioCountMsg -o $@

include $(MAKERULES)
```

The first and last line of this Makefile are the basic lines present in all TinyOS Makefiles; the line in bold defining BUILD_EXTRA_DEPS specifies some additional *make* targets to build alongside the main TinyOS application (if you are not familiar with make, this may be a good time to read a make tutorial, e.g., **this one**).

When you compile RadioCountToLeds for the first time, you will see that the two extra targets, `RadioCountMsg.py` and `RadioCountMsg.class`, are automatically created:

```
$ make mica2
mkdir -p build/mica2
mig python -target=mica2  -python-
classname=RadioCountMsg RadioCountToLeds.h
RadioCountMsg -o RadioCountMsg.py
mig java -target=mica2  -java-classname=RadioCountMsg
RadioCountToLeds.h RadioCountMsg -o RadioCountMsg.java
javac RadioCountMsg.java
    compiling RadioCountToLedsAppC to a mica2 binary
    ...
```

As this Makefile is written, these generated files are not deleted when you execute `make clean`. Fix this by adding the following line:

```
CLEAN_EXTRA = $(BUILD_EXTRA_DEPS) RadioCountMsg.java
```

to `apps/RadioCountToLeds/Makefile`. This defines the CLEAN_EXTRA make variable to be the same as BUILD_EXTRA_DEPS, with RadioCountMsg.java added to the end. The build system's **clean** target deletes all files in CLEAN_EXTRA:

```
$ make clean
rm -rf build RadioCountMsg.py RadioCountMsg.class
RadioCountMsg.java
rm -rf _TOSSIMmodule.so TOSSIM.pyc TOSSIM.py
```

Finally, to see how to pass options to the nesC compiler, we will change RadioCountToLeds's source code to set the message sending period based on the preprocessor symbol `SEND_PERIOD`. Change the line in `RadioCountToLedsC.nc` that reads

```
call MilliTimer.startPeriodic(1000);
```

to

```
call MilliTimer.startPeriodic(SEND_PERIOD);
```

and add the following line to RadioCountToLeds's Makefile:

```
CFLAGS += -DSEND_PERIOD=2000
```

Note the use of `+=` when defining CFLAGS: this allows the user to also pass options to nesC when invoking make as we saw above (`env CFLAGS=x make ...`).

Now compiling RadioCountToLeds gives:

```
$ make mica2
    ...
    compiling RadioCountToLedsAppC to a mica2 binary
ncc -o build/mica2/main.exe ... -DSEND_PERIOD=2000 ...
RadioCountToLedsAppC.nc -lm
    compiled RadioCountToLedsAppC to
build/mica2/main.exe
    ...
```

TinyOS Tools

The TinyOS build system is designed to make it easier to write Makefiles for applications that support multiple platforms, programmers, etc in a uniform way. However, it's use is not compulsory, and all the tools it is built on can be used in your own build system (e.g., your own Makefile or simple build script). Below we show how to build and install the RadioCountToLeds application for a micaz with the mib510 programmer using just a few commands.

First, we compile RadioCountToLedsAppC.nc (the main component of the application) using the nesC compiler, ncc:

```
$ ncc -target=micaz -o rcl.exe -Os -finline-
limit=100000 -Wnesc-all -Wall RadioCountToLedsAppC.nc
```

This generates an executable file, `rcl.exe`. Next, we want to install this program on a mote with mote id 15. First, we create a new executable, `rcl.exe-15`, where the variables storing the mote's identity are changed to 15, using the `tos-set-symbols` command:

```
$ tos-set-symbols rcl.exe rcl.exe-15 TOS_NODE_ID=15
ActiveMessageAddressC\$addr=15
```

Finally, we install this executable on the micaz using `uisp`, to a mib510 programmer connected to port /dev/ttyUSB1:

```
$ uisp -dpart=ATmega128 -dprog=mib510
-dserial=/dev/ttyUSB1 --erase --upload if=rcl.exe-15
Firmware Version: 2.1
Atmel AVR ATmega128 is found.
Uploading: flash
```

If you wish to follow this route, note two things: first, you can find out what commands the build system is executing by passing the `-n` option to make, which tells it to print rather than execute commands:

```
$ make -n micaz install.15 mib510
mkdir -p build/micaz
echo "    compiling RadioCountToLedsAppC to a micaz
binary"
ncc -o build/micaz/main.exe -Os -finline-limit=100000
-Wall -Wshadow -Wnesc-all -target=micaz -fnesc-
cfile=build/micaz/app.c -board=micasb  -fnesc-
dump=wiring -fnesc-dump='interfaces(!abstract())' -
fnesc-dump='referenced(interfacedefs, components)' -
fnesc-dumpfile=build/micaz/wiring-check.xml
RadioCountToLedsAppC.nc -lm
nescc-wiring build/micaz/wiring-check.xml
...
```

Second, all the commands invoked by the build system should have man pages describing their behaviour and options. For instance, try the following commands:

```
$ man tos-set-symbols
$ man ncc
$ man nescc
```

Related Documentation

- mica mote Getting Started Guide at **Crossbow**
- telos mote Getting Started Guide for **Moteiv**
- **Lesson 1** introduced the build system.
- **Lesson 10** describes how to add a new platform to the build system.
- GNU make man page.
- man pages for the nesC compiler (man ncc, man nescc) and the various TinyOS tools.

Lesson 14: Building a simple but full-featured application

Lesson 14 goes through the process of building a simple anti-theft application using many of the features and services of TinyOS 2. Lesson 14 is found in the tinyos-2.x/apps/AntiTheft directory. The powerpoint slides found there (also available in pdf form) go over the basic principles of TinyOS, and show how to build the accompanying AntiTheft application. Please start by reading the README.txt file in the AntiTheft directory.

To run the AntiTheft demo you will need mica2 or micaz motes, and some mts310 sensor boards (you can also use mts300 boards, though you will lose the movement detection functionality). If you do not have this hardware, the slides and AntiTheft code should still provide a good overview of TinyOS 2.

Lesson 15: The TinyOS `printf` Library

Last updated August 19th, 2007

This lesson demonstrates how to use the `printf` library located in `tos/lib/printf` to debug TinyOS applications by printing messages over the serial port.

Overview

Anyone familiar with TinyOS knows that debugging applications has traditionally been a very arduous, if not stressful process. While simulators like TOSSIM can be used to help verify the logical correctness of a program, unforseen problems inevitably arise once that program is deployed on real hardware. Debugging such a program typically involves flashing the three available LEDs in some intricate sequence or resorting to line by line analysis of a running program through the use of a JTAG.

It is common practice when developing desktop applications to print output to the terminal screen for debugging purposes. While tools such as `gdb` provide means of stepping though a program line by line, often times developers simply want to quickly print something to the screen to verify that the value of a variable has been set correctly, or determine that some sequence of events is being run in the proper order. It would be absurd to suggest that they only be allowed three bits of information in order to do so.

The TinyOS `printf` library provides this terminal printing functionality to TinyOS applications through motes connected to a pc via their serial interface. Messages are printed by calling `printf` commands using a familiar syntax borrowed from the C programming language. In order to use this functionality, developers simply need to include a single component

in their top level configuration file (`PrintfC`), and include a `"printf.h"` header file in any components that actually call `printf()`.

Currently, the `printf` library is only supported on msp430 and atmega128x based platforms (e.g. mica2, micaZ, telos, eyesIFX). In the future we hope to add support for other platforms as well.

The TinyOS `printf` Library

This section provides a basic overview of the TinyOS `printf` library, including the components that make it up and the interfaces they provide. In the following section we walk you through the process of actually using these components to print messages from a mote to your pc. If you dont care how `printf` works and only want to know how to use it, feel free to skip ahead to the next section.

The entire `printf` library consists of only 4 files located in the `tos/lib/printf` directory: one module, one configuration, one interface file, and one header file.

- **PrintfC.nc** -- Configuration file providing printf functionality to TinyOS applications
- **PrintfP.nc** -- Module implementing the printf functionality
- **PrintfFlush.nc** -- Interface for flushing printf messages over the serial port to a pc
- **printf.h** -- Header file specifying the printf message format and size of the flush buffer

The `PrintfC` configuration is the only component an application needs to wire in order to use the functionality provided by the TinyOS `printf` library. Below is the component graph of the `PrintfC` configuration:

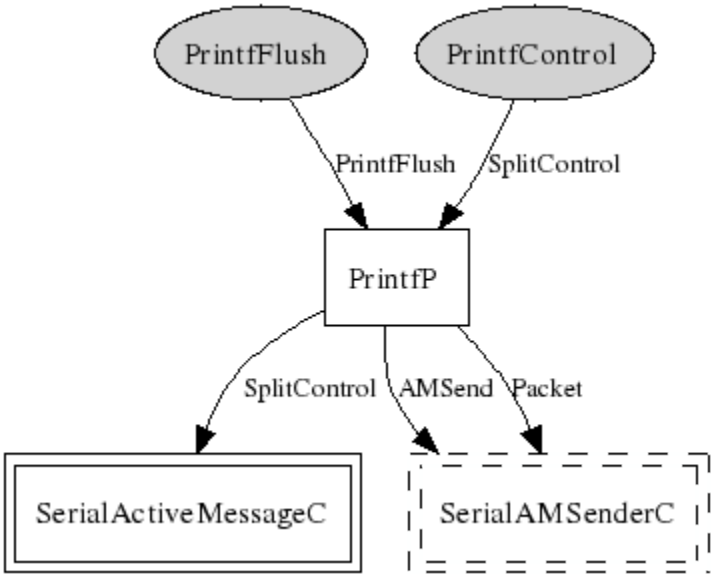


Figure 1: The component graph of the PrintfC configuration.

Conceptually, the operation of the TinyOS `printf` library is very simple. Developers supply strings to `printf()` commands in a distributed fashion throughout any of the components that make up a complete TinyOS application. These strings are buffered in a central location inside the `PrintfP` component and flushed out to a PC in the form of TinyOS SerialMessages upon calling the `flush()` command of the `PrintfFlush` interface.

By encapsulating the strings produced by calls to `printf()` inside standard TinyOS SerialMessages, applications that use the serial stack for other purposes can share the use of the serial port. Alternate implementations were considered in which `printf` would have had exclusive access to the serial port, and explicit flushing would not have been necessary. In the end, we felt it was better to give developers the freedom to decide exactly when messages should be printed, as well as allow them to send multiple types of SerialMessages in a single application.

Currently, only a single buffer is used to store the strings supplied to calls to `printf` before flushing them. This means that while the buffer is being flushed, any calls to `printf` will fail. In the future, we plan to implement a double buffered approach so that strings can continue to be buffered at the same time they are being printed.

There are also plans to provide a means of flushing messages out to a PC without requiring developers to make an explicit `flush()` call. This would allow developers to simply wire in the `PrintfC` component without having to make any calls to any interfaces it provides. In fact, the `PrintfC` component would not need to provide any interfaces at all. It would start itself up and then run in a loop, periodically flushing the contents of the `printf` buffer. Such functionality is useful in applications that do not really care when messages are printed or how long a delay the process of printing introduces to other sections of code. Explicit flushing would still be recommended in applications where the sections of code under examinatation are very timing sensitive (e.g. inside the CC2420 radio stack).

Using the TinyOS `printf` Library

To help guide the process of using the `printf` library, a `TestPrintf` application has been created. At present, this application is not included in the official TinyOS distribution (<= 2.0.2). If you are using TinyOS from a cvs checkout, you will find it located under `apps/tutorials/Printf`. Otherwise, you can obtain it from cvs by running the following set of commands from a terminal window:

```
cd $TOSROOT/apps/tutorials
cvs
-d:pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsro
ot/tinyos login
```

```
cvs -z3 -
d:pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsroo
t/tinyos co -P -d Printf
tinyos-2.x/apps/tutorials/Printf
```

Just hit enter when prompted for a CVS password. You do not need to enter one.

If you are not using cvs, you will also have to apply the patch found **here** in order to allow the `printf` library to compile correctly for atmega128x based platforms (i.e. mica2, micaz):

```
    cp tinyos-2.0-printf.patch $TOSROOT/..
    cd $TOSROOT/..
    patch -p0 < tinyos-2.0-printf.patch
```

Note that you may have to use 'sudo' when applying the patch if you run into permission problems.

The `TestPrintf` application demonstrates everything necessary to use the `printf` library. Go ahead and open the `TestPrintfAppC` configuration to see how the various interfaces provided by the `PrintfC` component have been wired in. You will want to do something similar in your own applications.

```
configuration TestPrintfAppC{
}
implementation {
  components MainC, TestPrintfC, LedsC;
  components PrintfC;

  TestPrintfC.Boot -> MainC;
  TestPrintfC.Leds -> LedsC;
  TestPrintfC.PrintfControl -> PrintfC;
  TestPrintfC.PrintfFlush -> PrintfC;
}
```

First, the `PrintfControl` interface has been wired in to enable turning on and off the service providing `printf` functionality. Turning on the `Printf` service implicity turns on the serial port for sending messages. Second, the `PrintfFlush` interface has been wired in to allow the application to control when `printf` messages should be flushed out over the serial line. In this application, all `printf()` commands are called directly within the `TestPrintfC` component. In general, `printf()` commands can be called from any component as long as they have included the `"printf.h"` header file.

Before examining the `TestPrintfC` component, first install the application on a mote and see what kind of output it produces. Note that the instructions here are only valid for installation on a telosb mote on a linux based TinyOS distribution. For installation on other systems or for other mote platforms, please refer to **lesson 1** for detailed instructions.

To install the application on the mote, run the following set of commands.

```
cd $TOSROOT\apps\tests\TestPrintf
make telosb install bsl,/dev/ttyUSBXXX
```

You will notice during the installation process that a pair of java files are compiled along with the TinyOS application. The first java file, `PrintfMsg.java`, is generated by `mig` to encapsulate a TinyOS `printf` message received over the serial line (for more information on mig and how it generates these files, please refer to the section entitled "MIG: generating packet objects" in **lesson 4**). The second file, `PrintfClient.java` is used to read `printf` messages received from a mote and print them to your screen.

To see the output generated by `TestPrintf` you need to start the `PrintfClient` by running the following command:

```
cd $TOSROOT\apps\tests\TestPrintf
java PrintfClient -comm serial@/dev/ttyUSBXXX:telosb
```

After resetting the mote, the following output should be printed to your screen:

```
Hi I am writing to you from my TinyOS application!!
Here is a uint8: 123
Here is a uint16: 12345
Here is a uint32: 1234567890
I am now iterating: 0
I am now iterating: 1
I am now iterating: 2
I am now iterating: 3
I am now iterating: 4
This is a really short string...
I am generating this string to have just less than 250
characters since that is the limit of the size I put
on my
maximum buffer when I instantiated the PrintfC
component.
Only part of this line should get printed bec
```

Note that the 'tty' device (i.e. COM port) specified when starting the PrintfClient MUST be the one used for communicating with a mote over the serial line. On telos and mica motes this is the same port that the mote is programmed from. Other motes, such as eyesIFX, have one port dedicated to programming and another for communication. Just make sure you use the correct one.

If for some reason you do not receive the output shown above, please refer to **lesson 4** to verify you have done everything necessary to allow serial communication between your pc and the mote. Remember that when using the MIB510 programming board that the switch on the very front of the board must be set to the OFF position in order to send messages from the mote to the pc.

Go ahead and open up `TestPrintfC` to see how this output is being generated.

Upon receiving the booted event, the `Printf` service is started via a call to `PrintfControl.start()`

```
event void Boot.booted() {
  call PrintfControl.start();
}
```

Once the `Printf` service has been started, a `PrintfControl.startDone()` event is generated. In the body of this event the first four lines of output are generated by making successive calls to `printf` and then flushing the buffer they are stored in.

```
event void PrintfControl.startDone(error_t error) {
  printf("Hi I am writing to you from my TinyOS
application!!\n");
  printf("Here is a uint8: %u\n", dummyVar1);
  printf("Here is a uint16: %u\n", dummyVar2);
  printf("Here is a uint32: %ld\n", dummyVar3);
  call PrintfFlush.flush();
}
```

Once these first four lines have been flushed out, the `PrintfFlush.flushDone()` event is signaled. The body of this event first prints the next 5 lines in a loop, followed by the last five lines. Finally, once all lines have been printed, the `Printf` service is stopped via a call to `PrintfControl.stop()`.

```
event void PrintfFlush.flushDone(error_t error) {
  if(counter < NUM_TIMES_TO_PRINT) {
    printf("I am now iterating: %d\n", counter);
    call PrintfFlush.flush();
  }
```

```
  else if(counter == NUM_TIMES_TO_PRINT) {
    printf("This is a really short string...\n");
    printf("I am generating this string to have just
less ...
    printf("Only part of this line should get printed
bec ...
    call PrintfFlush.flush();
  }
  else call PrintfControl.stop();
  counter++;
}
```

Notice that the last line of output is cut short before being fully printed. If you actually read the line printed above it you can see why. The buffer used to store TinyOS `printf` messages before they are flushed is by default limited to 250 bytes. If you try and print more characters than this before flushing, then only the first 250 characters will actually be printed. This buffer size is configurable, however, by specifying the proper CFLAGS option in your Makefile.

```
CFLAGS += -DPRINTF_BUFFER_SIZE=XXX
```

Once the the `Printf` service has been stopped, the `PrintfControl.stopDone()` event is signaled and Led 2 is turned on to signify that the application has terminated.

```
  event void PrintfControl.stopDone(error_t error) {
    counter = 0;
    call Leds.led2Toggle();
    printf("This should not be printed...");
    call PrintfFlush.flush();
  }
```

Notice that the call to `printf()` inside the body of the `PrintfControl.stopDone()` event never produces any output. This is because the `Printf` service has been stopped before this command is called.

Conclusion

A few points are worthy of note before jumping in and writing your own applications that use the functionality provided by the `printf` library.

1. In order to use the `printf` library, the `tos/lib/printf` directory must be in your include path. The easiest way to include it is by adding the following line directly within the Makefile of your top level application:

   ```
   CFLAGS += -I$(TOSDIR)/lib/printf
   ```

   Remember that changing the `printf` buffer size is done similarly:

   ```
   CFLAGS += -DPRINTF_BUFFER_SIZE=XXX
   ```

2. You MUST be sure to #include `"printf.h"` header file in every component in which you would like to call the `printf()` command. Failure to do so will result in obscure error messages making it difficult to identify the problem.

Hopefully you now have everything you need to get going with the TinyOS `printf` library. All questions (or comments) about the use of this library should be directed to **tinyos-help** mailing list.

Enjoy!!

Lesson 16: Writing Low Power Sensing Applications

Last updated August 31st, 2007

This lesson demonstrates how to write low power sensing applications in TinyOS. At any given moment, the power consumption of a wireless sensor node is a function of its microcontroller power state, whether the radio, flash, and sensor peripherals are on, and what operations active peripherals are performing. This tutorial shows you how to best utilize the features provided by TinyOS to keep the power consumption of applications that use these devices to a minumum.

Overview

Energy management is a critical concern in wireless sensor networks. Without it, the lifetime of a wireless sensor node is limited to just a few short weeks or even days, depending on the type of application it is running and the size of batteries it is using. With proper energy management, the same node, running the same application, can be made to last for months or years on the same set of batteries.

Ideally, one would want all energy management to be handled transparently by the operating system, relieving application developers from having to deal with this burden themselves. While attempts have been made in the past to provide such capabilities, most operating systems today

still just provide the necessary primitives for performing energy management -- the logic of when and how to do so is left completely up to the application.

TinyOS provides a novel method of performing energy management directly within the OS. The method it uses is as efficient as it is elegant. Developers no longer have to struggle with code that explicitly manages the power state for any of the peripheral devices that it uses. To get a better idea of the complexity involved in doing something as simple as periodically taking a set of sensor readings and logging them to flash in the most energy efficient manner possible, take a look at the pseudo code found below.

```
Every Sample Period:
  Turn on SPI bus
  Turn on flash chip
  Turn on voltage reference
  Turn on I2C bus
  Log prior readings
  Start humidity sample
  Wait 5ms for log
  Turn off flash chip
  Turn off SPI bus
  Wait 12ms for vref
  Turn on ADC
  Start total solar sample
  Wait 2ms for total solar
  Start photo active sample
  Wait 2ms for photo active
  Turn off ADC
  Turn off vref
  Wait 34ms for humidity
  Start temperature sample
  Wait 220ms for temperature
  Turn off I2C bus
```

With the methods provided by TinyOS, hand-tuned application code that looks like that can be transformed into this:

```
Every Sample Period:
  Log prior readings
  Sample photo active
  Sample total solar
  Sample temperature
```

```
  Sample humidity
```

The pseudo code shown above is for concurrently sampling all of the the onboard sensors found on the latest revision of the tmote sky sensor nodes. Experiments have shown that using the methods provided by TinyOS, this application comes within 1.6% of the energy efficiency of the hand-tuned implementation -- even at sampling rates as fast as 1 sample per second. For more information on these experiments and the method TinyOS uses to actually manage energy so efficiently, please refer to the paper found **here.**

The rest of this tutorial is dedicated to teaching you how to write applications that allow TinyOS to manage energy for you in the most efficient manner possible. As a rule, TinyOS can manage energy more efficiently when I/O requests are submitted by an application in parallel. Submitting them serially may result in energy wasted by unnecessarily turning devices on and off more frequently than required.

Peripheral Energy Management

Compare the following two code snippets:

Parallel

```
event void Boot.booted()
{
  call
Timer.startPeriodic(SAMPL
E_PERIOD);
}

event void Timer.fired()
{
  call
LogWrite.append(current_e
ntry,
sizeof(log_entry_t));

  current_entry_num = !
current_entry_num;
  current_entry =
&(entry[current_entry_num
]);
  current_entry-
>entry_num = +
+log_entry_num;
```

Serial

```
event void Boot.booted()
{
  call
Timer.startPeriodic(SAMPL
E_PERIOD);
}

event void Timer.fired()
{
  call Humidity.read();
}

event void
Humidity.readDone(error_t
result, uint16_t val) {
  if(result == SUCCESS) {
    entry->rad = val;
  }
  else current_entry->rad
= 0xFFFF;
  call
Temperature.read();
```

```
  call Humidity.read();
  call
Temperature.read();
  call Photo.read();
  call Radiation.read();
}

event void
Humidity.readDone(error_t
result, uint16_t val) {
  if(result == SUCCESS) {
   current_entry->hum =
val;
  }
  else current_entry->hum
= 0xFFFF;
}

event void
Temperature.readDone(erro
r_t result, uint16_t val)
{
  if(result == SUCCESS) {
    current_entry->temp =
val;
  }
  else current_entry-
>temp = 0xFFFF;
}

event void
Photo.readDone(error_t
result, uint16_t val) {
  if(result == SUCCESS) {
    current_entry->photo
= val;
  }
  else current_entry-
>photo = 0xFFFF;
 }

event void
}

event void
Temperature.readDone(erro
r_t result, uint16_t val)
{
  if(result == SUCCESS) {
    entry->rad = val;
  }
  else current_entry->rad
= 0xFFFF;
  call Photo.read();
}

event void
Photo.readDone(error_t
result, uint16_t val) {
  if(result == SUCCESS) {
    entry->rad = val;
  }
  else current_entry->rad
= 0xFFFF;
  call Radiation.read();
}

event void
Radiation.readDone(error_
t result, uint16_t val) {
  if(result == SUCCESS) {
    entry->rad = val;
  }
  else current_entry->rad
= 0xFFFF;
  call
LogWrite.append(entry,
sizeof(log_entry_t));
}

event void
LogWrite.appendDone(void*
buf,

storage_len_t len,
```

```
Radiation.readDone(error_
t result, uint16_t val) {
  if(result == SUCCESS) {
    current_entry->rad =
val;
  }
  else current_entry->rad
= 0xFFFF;
}

event void
LogWrite.appendDone(void*
buf,

storage_len_t len,

bool recordsLost,

error_t error) {
  if (error == SUCCESS)
    call
Leds.led2Toggle();
}
```

```
bool recordsLost,

error_t error) {
  if (error == SUCCESS)
    call
Leds.led2Toggle();
}
```

In the parallel case, logging to flash and requesting samples from each sensor is all done within the body of the `Timer.fired()` event. In the serial case, a chain of events is triggered by first calling `Humidity.read()` in `Timer.fired()`, sampling each subsequent sensor in the body of the previous `readDone()` event, and ending with all sensor readings being logged to flash.

By logging to flash and sampling all sensors within the body of a single event, the OS has the opportunity to schedule each operation as it sees fit. Performing each operation after the completion of the previous one gives the OS no such opportunity. The only downside of the parallel approach is that the application must manually manage a double buffer so that the values written to flash are not overwritten before they are logged. To save the most energy, however, the parallel version should always be used. Keep in mind that in both cases, the developer must also make sure that the sampling interval is longer than the time it takes to gather all sensor readings. If it is not, data corruption will inevitably occur.

Radio Power Management

By default, TinyOS provides low power radio operation through a technique known as *Low-Power Listening*. In low-power listening, a node turns on its radio just long enough to detect a

carrier on the channel. If it detects a carrier, then it keeps the radio on long enough to detect a packet. Because the LPL check period is much longer than a packet, a transmitter must send its first packet enough times for a receiver to have a chance to hear it. The transmitter stops sending once it receives a link-layer acknowledgment or a timeout of twice the check period. When a node receives a packet, it stays awake long enough to receive a second packet. Therefore, a packet burst amortizes the wakeup cost of the first packet over the follow-up packets. It is therefore more energy efficient to send packets in bursts when using low-power listening than sending individual packets at some fixed constant rate. Keep this in mind when developing applications that require low power operation.

Controlling the operation of low-power listening in TinyOS is provided through the use of the `LowPowerListening` interface. Currently, this interface is only supported for the cc1000 and cc2420 radios. In the future we hope to support other radio platforms as well.

```
interface LowPowerListening {
  /**
   * Set this this node's radio sleep interval, in
milliseconds.
   * Once every interval, the node will sleep and
perform an Rx check
   * on the radio.  Setting the sleep interval to 0
will keep the radio
   * always on.
   *
   * This is the equivalent of setting the local duty
cycle rate.
   *
   * @param sleepIntervalMs the length of this node's
Rx check interval, in [ms]
   */
  command void setLocalSleepInterval(uint16_t
sleepIntervalMs);

  /**
   * @return the local node's sleep interval, in [ms]
   */
  command uint16_t getLocalSleepInterval();

  /**
   * Set this node's radio duty cycle rate, in units
of [percentage*100].
   * For example, to get a 0.05% duty cycle,
   *   call LowPowerListening.setDutyCycle(5);
   *
```

```
   * For a 100% duty cycle (always on),
   *   call LowPowerListening.setDutyCycle(10000);
   *
   * This is the equivalent of setting the local sleep
interval explicitly.
   *
   * @param dutyCycle The duty cycle percentage, in
units of [percentage*100]
   */
  command void setLocalDutyCycle(uint16_t dutyCycle);

  /**
   * @return this node's radio duty cycle rate, in
units of [percentage*100]
   */
  command uint16_t getLocalDutyCycle();

  /**
   * Configure this outgoing message so it can be
transmitted to a neighbor mote
   * with the specified Rx sleep interval.
   * @param msg Pointer to the message that will be
sent
   * @param sleepInterval The receiving node's sleep
interval, in [ms]
   */
  command void setRxSleepInterval(message_t *msg,
uint16_t sleepIntervalMs);

  /**
   * @return the destination node's sleep interval
configured in this message
   */
  command uint16_t getRxSleepInterval(message_t *msg);

  /**
   * Configure this outgoing message so it can be
transmitted to a neighbor mote
   * with the specified Rx duty cycle rate.
   * Duty cycle is in units of [percentage*100], i.e.
0.25% duty cycle = 25.
   *
   * @param msg Pointer to the message that will be
sent
```

```
   * @param dutyCycle The duty cycle of the receiving
mote, in units of
   *     [percentage*100]
   */
  command void setRxDutyCycle(message_t *msg, uint16_t
dutyCycle);

  /**
   * @return the destination node's duty cycle
configured in this message
   *     in units of [percentage*100]
   */
  command uint16_t getRxDutyCycle(message_t *msg);

  /**
   * Convert a duty cycle, in units of
[percentage*100], to
   * the sleep interval of the mote in milliseconds
   * @param dutyCycle The duty cycle in units of
[percentage*100]
   * @return The equivalent sleep interval, in units
of [ms]
   */
  command uint16_t dutyCycleToSleepInterval(uint16_t
dutyCycle);

  /**
   * Convert a sleep interval, in units of [ms], to a
duty cycle
   * in units of [percentage*100]
   * @param sleepInterval The sleep interval in units
of [ms]
   * @return The duty cycle in units of
[percentage*100]
   */
  command uint16_t sleepIntervalToDutyCycle(uint16_t
sleepInterval);

}
```

This interface is located in `tos/interfaces` in the standard TinyOS tree. Take a look at the comments for each command to familiarize yourself with how this interface can be used.

Using this interface typically involves first setting a nodes local duty cycle within the `Boot.booted()` event of the top level application. For each packet the application wishes to send, the duty cycle of its destination is then specified as metadata so that the correct number of preambles can be prepended to it. The code snippet found below demonstrates this usage pattern:

```
event void Boot.booted() {
  call LPL.setLocalSleepInterval(LPL_INTERVAL);
  call AMControl.start();
}

event void AMControl.startDone(error_t e) {
  if(e != SUCCESS)
    call AMControl.start();
}

...

void sendMsg() {
  call LPL.setRxSleepInterval(&msg, LPL_INTERVAL);
  if(call Send.send(dest_addr, &msg, sizeof(my_msg_t))
!= SUCCESS)
    post retrySendTask();
}
```

The `AMControl` interface is provided by `ActiveMessageC`, and is used, among other things, to enable the operation of low-power listening for the radio. Once `AMControl.start()` has completed successfully, the radio begins to duty cycle itself as specified by the parameter to the `setLocalSleepInterval()` command. Calling `setRxSleepInterval()` with a specific sleep interval then allows the correct number of preambles to be sent for the message specified in its parameter list.

Microcontroller Power Management

Microcontrollers often have several power states, with varying power draws, wakeup latencies, and peripheral support. The microcontroller should always be in the lowest possible power state that can satisfy application requirements. Determining this state accurately requires knowing a great deal about the power state of many subsystems and their peripherals. Additionally, state transitions are common. Every time a microcontroller handles an interrupt, it moves from a low power state to an active state, and whenever the TinyOS scheduler finds the task queue empty it returns the microcontroller to a low power state. TinyOS uses three mechanisms to decide what low power state it puts a microcontroller into: status and control registers, a dirty bit, and a power state override. Please refer to **TEP 112** for more information.

As a developer, you will not have to worry about MCU power managment at all in most situations. TinyOS handles everything for you automatically. At times, however, you may want to use the provided power state override functionality. Take a look at `tos/chips/atm128/timer/HplAtm128Timer0AsyncP.nc` if you are interested in seeing an example of where this override functionality is used.

Low Power Sensing Application

A fully functional low-power sensing application has been created that combines each of the techniques found in this tutorial. At present, this application is not included in the official TinyOS distribution (<= 2.0.2). If you are using TinyOS from a cvs checkout, you will find it located under `apps/tutorials/LowPowerSensing`. Otherwise, you can obtain it from cvs by running the following set of commands from a terminal window:

```
cd $TOSROOT/apps/tutorials
cvs
-d:pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsro
ot/tinyos login
cvs -z3 -
d:pserver:anonymous@tinyos.cvs.sourceforge.net:/cvsroo
t/tinyos co -P -d LowPowerSensing
tinyos-2.x/apps/tutorials/LowPowerSensing
```

Just hit enter when prompted for a CVS password. You do not need to enter one.

This application has been tested on telosb and mica2 platforms, but should be usable on others without modification. Take a look at the README file found in the top level directory for more information.

Related Documentation

- **TEP 103: Permanent Data Storage (Flash)**
- **TEP 105: Low Power Listening**
- **TEP 108: Resource Arbitration**
- **TEP 109: Sensors and Sensor Boards**
- **TEP 112: Microcontroller Power Management**
- **TEP 114: SIDs: Source and Sink Independent Drivers**
- **TEP 115: Power Management of Non-Virtualised Devices**
- **Integrating Concurrency Control and Energy Management in Device Drivers**