TinyOS Toolchain

From TinyOS Wiki

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.

Contents

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 telosbAdditionally, 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 RadioCountMs

RadioCountMsg.class: RadioCountMsg.java
	javac RadioCountMsg.java

RadioCountMsg.java: RadioCountToLeds.h
	mig java -target=$(PLATFORM) $(CFLAGS) -java-classname=RadioCountMsg RadioCountToLeds.h RadioCountMsg -c

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 (http://oucsace.cs.ohiou.edu/~bhumphre/makefile.html)).

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 RadioCountToLe
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 (http://www.xbow.com)
- telos mote Getting Started Guide for Moteiv (http://www.moteiv.com)
- 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.

< **Previous Lesson** | **Top** | **Next Lesson** >
