Platforms

From TinyOS Wiki

Contents

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 developers, and you can safely skip it if you have no intention of working down to the lowest level or developing 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 developer 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. Developing 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,TaskBasic,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.*

In addition, there must be a file named "platform.h", even if this file is empty.

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
       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
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;
}
```

OK, let's try to make again. If we get something wrong as the following:

```
naoshi@ubuntu:/opt/tinyos-2.x/apps/Null$ make yamp
mkdir -p build/yamp
    compiling NullAppC to a yamp binary
ncc -o build/yamp/main.exe  -Os -fnesc-separator=__ -Wall -Wshadow -Wnesc-all -target=yamp -fnesc-cfile=build/ya
-board= -DDEFINED_TOS_AM_GROUP=0x22 -DIDENT_APPNAME=\"NullAppC\" -DIDENT_USERNAME=\"naoshi\" -DIDENT_HOSTNAME=\"
-DIDENT_USERHASH=0x41acd239L -DIDENT_TIMESTAMP=0x4d7f00a0L -DIDENT_UIDHASH=0x5f8c3a6dL  NullAppC.nc -lm
/opt/tinyos-2.x/tos/system/tos.h:41:22: error: platform.h: No such file or directory
make: *** [exe0] Error 1
```

Here is the solution:

```
$ touch /opt/tinyos-2.x/tos/platforms/yamp/platform.h
```

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: developing the components to drive a radio transceiver or MCU (which are necessary if the platform uses chips that are not currently supported in TinyOS). Developing 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 (https://sourceforge.net/projects/nescc)
- nesC reference manual (http://nescc.sourceforge.net/papers/nesc-ref.pdf)
- TinyOS Programming Guide (http://csl.stanford.edu/~pal/pubs/tinyos-programming-1-0.pdf)
- TEP 2: Hardware Abstraction Architecture (http://www.tinyos.net/tinyos-2.x/doc/html/tep102.html)
- TEP 106: Schedulers and Tasks (http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html)
- TEP 107: TinyOS 2.x Boot Sequence (http://www.tinyos.net/tinyos-2.x/doc/html/tep107.html)
- TEP 117: Low-level I/O (http://www.tinyos.net/tinyos-2.x/doc/html/tep117.html)
- TEP 131: Creating a New Platform for TinyOS 2.x (http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep131.html)

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