Modules and the TinyOS Execution Model

From TinyOS Wiki

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

Contents

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 mote platforms do not have hardware-based memory protection, there is no separation 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 @safe(){
  uses interface Timer<TMilli> as Timer0;
  uses interface Timer<TMilli> as Timer1;
  uses interface Timer<TMilli> as Timer2;
  uses interface Leds;
  uses interface Boot;
}
implementation
{
  event void Boot.booted()
  {
    call Timer0.startPeriodic( 250 );
    call Timer1.startPeriodic( 500 );
    call Timer2.startPeriodic( 1000 );
  }

  event void Timer0.fired()
  {
    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.led1On();
    }
    else {
      call Leds.led1Off();
    }
    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 @safe(){
  uses interface Timer<TMilli> as Timer0;
  uses interface Leds;
  uses 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
-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 -
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
-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 -
    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/lib/timer/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 traditional operating systems are interrupt bottom halves (http://www.tldp.org/LDP/tlk/kernel/kernel.html) 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) <ref name="task">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.</ref>.

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 responsive: 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 interface 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 (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)

<references/>

< Previous Lesson | Top | Next Lesson >