Boot Sequence

From TinyOS Wiki

Contents

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 Boot;
  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 (http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html) 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 (http://www.tinyos.net/tinyos-2.x/doc/html/tep112.html) 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 (http://www.tinyos.net/tinyos-2.x/doc/html/tep106.html)
- TEP 107: Boot Sequence (http://www.tinyos.net/tinyos-2.x/doc/html/tep107.html)

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* (http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf)

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