TinyOS 2.x Boot Sequence

TEP:	107
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	Philip Levis

Abstract

This memo documents the structure and implementation of the mote boot sequence in TinyOS 2.x.

1. Introduction

TinyOS has a set of calling conventions and semantics in its boot sequence. Earlier versions of TinyOS used an interface named "StdControl" to take care of system initialization and starting required software systems. Experience with several hardware platforms showed StdControl to be insufficient, as it provided only a synchronous interface. Additionally, StdControl bundled the notion of initialization, which happens only at boot, with power management and service control. TinyOS 2.x solves these problems by separating what was once StdControl into three separate interfaces: one for initialization, one for starting and stopping components, and one for notification that the mote has booted. This memo describes the TinyOS boot sequence and reasons for its semantics.

2. TinyOS 1.x Boot Sequence

The TinyOS 1.x boot sequence is uniform across most mote platforms (TOSSIM has a very different boot sequence, as it is a PC program). The module RealMain implements main(), and has the following signature:

```
module RealMain {
  uses {
    command result_t hardwareInit();
    interface StdControl;
    interface Pot;
  }
}
```

The mote main() function uses a mix of nesC and C:

```
int main() __attribute__ ((C, spontaneous)) {
  call hardwareInit();
  call Pot.init(10);
  TOSH_sched_init();

  call StdControl.init();
  call StdControl.start();
  __nesc_enable_interrupt();

  while(1) {
    TOSH_run_task();
  }
}
```

Several problems exist. Some of these calls are artifacts of old platforms: the Pot component refers to the mica variable potentiometer for controlling radio transmission power, and for other platforms is a stub component Some of the calls -- TOSH_sched_init and TOSH_run_task -- are C functions that are implemented in other, automatically included files. Separation from the nesC component model makes changing what lies behind these functions more difficult than normal in TinyOS.

More importantly, the initialization sequence has several limitations. The component HPLInit implements the hardwareInit command (wired by the component Main): hardware initialization may not be part of a pure HPL layer. The scheduler is initialized after hardware, which means that no hardware initialization can post a task if it needs one. The StdControl interface combines component initialization (init()) and activation (start()/stop()); if a component needs to be initialized by RealMain, it must also be started. Separating these two leads to more flexible power management, and distinguishes required low-level components that must always be running (such as a Timer) from high level components that can be power managed (such as an application). Finally, some components that need to often need to be started by main, such as a radio, do not follow a synchronous start/stop model. In this case, some components can't operate properly until the radio starts, but main has no mechanism for waiting for the radio start completion event.

3. TinyOS 2.x Boot Interfaces

The TinyOS 2.x boot sequence uses three interfaces:

- Init, for initializing component/hardware state
- Scheduler, for initializing and running tasks
- Boot, for signalling that the system has successfully booted

The Init interface has a single command, init():

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

Init provides a synchronous interface, enabling initialization ordering. Unlike normal execution, in which operations from a wide range of components need to be interleaved effectively, initialization is a sequential, synchronous operation: no component can be started until initialization is complete. If a particular component's initialization requires waiting for interrupts or other asynchronous events, then it must explicitly wait for them (e.g., with a spin loop), MUST NOT return until complete. Otherwise the system may start before initialization is complete.

The Scheduler interface is for initializing and controlling task execution. It is detailed in TEP 106[1].

The Boot interface has a single event, booted(), which the boot sequence signals when it has completed:

```
interface Boot {
  event void booted();
}
```

4. TinyOS 2.x Boot Sequence

The module RealMainP implements the standard TinyOS 2.x boot sequence. The configuration MainC wires some of RealMainP's interfaces to components that implement standard abstractions and exports the others that are application specific. Code above the Hardware Independent Layer (TEP 2) SHOULD wire to MainC and not RealMainP:

```
module RealMainP {
  provides interface Booted;
  uses {
    interface Scheduler;
    interface Init as PlatformInit;
    interface Init as SoftwareInit;
  }
}
implementation {
  int main() __attribute__ ((C, spontaneous)) {
    atomic {
      platform_bootstrap();
      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() { }
}
```

4.1 Initialization

The first step in the boot sequence is initializing the system:

```
atomic {
  platform_bootstrap();
  call Scheduler.init();
  call PlatformInit.init();
  while (call Scheduler.runNextTask());
  call SoftwareInit.init();
```

```
  while (call Scheduler.runNextTask());
}
```

The first call, platform_bootstrap(), is a minimalist function that places the system into an executable state. This function MUST NOT include operations besides those which are absolutely necessary for further code, such as scheduler initialization, to execute. Examples of platform_bootstrap() operations are configuring the memory system and setting the processor mode. Generally, platform_bootstrap() is an empty function. TinyOS's top-level include file, `tos.h`, includes a default implementation of this function which does nothing. If a platform needs to replace the default, it SHOULD put it in a platform's `platform.h` file as a #define. The implementation of `tos.h` supports this:

```
/* This platform_bootstrap macro exists in accordance with TEP
   107. A platform may override this through a platform.h file. */
#include <platform.h>
#ifndef platform_bootstrap
#define platform_bootstrap() {}
#endif
```

The boot sequence has three separate initializations: Scheduler, PlatformInit, and SoftwareInit. The boot configuration (MainC) wires the first two automatically, to TinySchedulerC (discussed in TEP 106) and to PlatformC:

```
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;
}
```

MainC exports the Boot and SoftwareInit interfaces for applications to wire to. TinySchedulerC is the standard name for the TinyOS scheduler. As the initialization sequence requires being able to run tasks, the boot sequence initializes it first. The second step of initialization is to call PlatformInit.init(), which MainC wires to a component named PlatformC. PlatformInit is for initializations which must follow a very specific order due to hidden dependencies, e.g., as part of making the overall hardware platform operable. One example of this sort of initialization is clock calibration. Because PlatformInit calls the component PlatformC, each platform can specify the required initialization order. As these hidden dependencies must be due to hardware, the sequence is platform-specific. A port of TinyOS to a new plaform MUST include a component PlatformC which provides one and only one instance of the Init interface.

Initializations invoked through PlatformC meet some or all of the following criteria:

1. The initialization requires configuring hardware resources. This implies that the code is platform-specific.
2. The initialization should always be performed.
3. The initialization is a prerequisite for common services in the system.

Three example operations that often belong in PlatformInit are I/O pin configuration, clock calibration, and LED configuration. I/O pin configuration meets the first two criteria. It should always be performed (regardless of what components the OS uses) for low-power reasons: incorrectly configured pins can draw current and prevent the MCU from entering its lowest power sleep state[2]. Clock calibration meets all three criteria. LED configuration is a special case: while it nominally meets all three criteria, the most important one is the third: as LEDs are often needed during SoftwareInit initialization, they must be set up before it is invoked.

Note that not all code which meets some of these criteria is wired through PlatformC. In particular, criterion 1 is typically necessary but not sufficient to require PlatformC. For example, a timer system that configures overflow and capture settings or a UART stack that sets the baud rate and transmission options can often be wired to SoftwareInit. They are encapsulated abstractions which will not be invoked or started until the boot event, and only need to be configured if the system includes their functionality.

Components whose initialization does not directly depend on hardware resources SHOULD wire to MainC.SoftwareInit. If a component requires a specific initialization ordering, then it is responsible for establishing that ordering. Due to the semantics of Init, this is usually quite rare; a component SHOULD NOT introduce initialization dependencies unless they are required.

One common approach is for a configuration to "auto-wire" the initialization routines of its internal components. The configuration does not provide an Init interface. Virtualized services often take this approach, as the service, rather than the clients, is what needs to be initialized. For example, the standard Timer virtualization[3], TimerMilliC, wires to TimerMilliP, which is a very simple configuration that takes the underlying implementation (HilTimerMilliC) and wires it to MainC:

```
configuration TimerMilliP {
  provides interface Timer<TMilli> as Timinitialization in order-
erMilli[uint8_t id];
}
implementation {
  components HilTimerMilliC, MainC;
  MainC.SoftwareInit -> HilTimerMilliC;
  TimerMilli = HilTimerMilliC;
}
```

Rather than require an application to wire HilTimerMilliC to MainC, TimerMilliP does it automatically. When a component instantiates a TimerMilliC, that names TimerMilliP, which will automatically make sure that the timer system is initialized when TinyOS boots.

4.2 Interrupts in Initialization

Interrupts are not enabled until all calls to Init.init have returned. If a component's initialization needs to handle interrupts, it can do one of three things:

1) If a status flag for the interrupt exists, the Init.init() implementations SHOULD use a spin loop to test for when an interrupt has been issued.
2) If no such flag exists, the Init.init() implementation MAY temporarily enable interrupts, if doing so will not cause any other components to handle an interrupt. That is, if a component enables an interrupt, it MUST NOT enable interrupts whose handlers would invoke any other component. Furthermore, when Init.init() exits, the interrupts must be disabled.
3) If no such flag exists and there is no way to isolate which interrupt handlers are called, then the component MUST rely on mechanisms outside the Init sequence, such as SplitControl.

The boot sequence assumes that 1) is by far the dominant case. There are, however, possible situations where a component might need to handle an interrupt because of, e.g., hardware limitations (no pending flag) or to catch a brief edge transition. In these cases, a component can handle an interrupt in the boot sequence only if doing so will not cause any other component to handle an interrupt. As they have all been written assuming that interrupts are not enabled until after Init completes, making one of them handle an interrupt could cause it to fail.

Depending on what capabilities the hardware provides, there are several ways to meet these requirements. The simplest is to push these initialization edge cases out of the main boot sequence, e.g., into SplitControl. A second possibility is to redirect the interrupt table, if the MCU supports doing so. Whichever mechanism is chosen, extreme care needs to be used in order to not disrupt the operation of other components.

Unless part of a hardware abstraction architecture (HAA)[4], the Init.init() command MUST NOT assume that other components have been initialized unless it has initialized them, and MUST NOT call any functional interfaces on any components that might be shared or interact with shared resources. Components MAY call functions on other components that are completely internal to the subsystem. For example, a networking layer can call queue operations to initialize its queue, but a link layer must not send commands over an SPI bus. An HAA component MAY make other calls to initialize hardware state. A component that is not part of an HAA SHOULD NOT call Init.init() on other components unless it needs to enforce a temporal ordering on initialization.

If a component A depends on another component, B, which needs to be initialized, then A SHOULD wire B's Init directly to the boot sequence, unless there is a temporal ordering requirement to the initialization. The purpose of this convention is to simplify component initialization and the initialization sequence.

5. Implementation

The following files in `tinyos-2.x/tos/system` contain the reference implementations of the TinyOS boot sequence:

- `RealMainP.nc` is the module containing the function `main`.
- `MainC.nc` is the configuration that wires RealMainP to PlatformC and TinySchedulerC[1].

6. Author's Address

Philip Levis
467 Soda Hall
UC Berkeley
Berkeley, CA 94720

phone - +1 510 290 5283

email - pal@cs.berkeley.edu

7. Citations

[1] TEP 106: Schedulers and Tasks.

[2] TEP 112: Microcontroller Power Management.

[3] TEP 102: Timers.

[4] TEP 2: Hardware Abstraction Architecture.