TinyOS 2.0 Overview

Author: Philip Levis
Date: Oct 30 2006

Note

This document gives a brief overview of TinyOS 2.0, highlighting how and where it departs from 1.1 and 1.0. Further detail on these changes is detailed in TEP (TinyOS Enhancement Proposal) documents.

1. Introduction

TinyOS 2.0 is a clean slate redesign and re-implementation of TinyOS. Its development was motivated by our belief that many aspects of 1.x strain to meet requirements and uses that were not foreseen when it was designed and implemented. The structure and interfaces 1.x defines have several fundamental limitations. While these limitations can be worked around, this practice has led to tightly coupled components, hard to find interactions, and a very steep learning curve for a newcomer to sensor network programming.

TinyOS 2.0 is not backwards compatible with 1.x: code written for the latter will not compile for the former. However, one important aspect of 2.0's design is to minimize the difficulty of upgrading code. Therefore, while porting a 1.x application to 2.0 will require some work, it should not be very much.

This document provides a high-level overview of 2.0 and describes some of the ways in which it departs from 1.x. It covers the basic TinyOS abstractions, such as hardware abstractions, communication, timers, the scheduler, booting and initialization. Further detail on each of these can be found in TEPs (TinyOS Enhancement Proposals), which document and describe these abstractions.

2. Platforms/Hardware Abstraction

Platforms exist in the `tos/platforms` subdirectory. In TinyOS 2.0, a *platform* is a collection of *chips* and some glue code that connects them together. For example, the mica2 platform is the CC1000 radio chip and the ATmega128 microcontroller, while the micaZ platform is the CC2420 radio and the ATmega128 microcontroller, and the Teloi platforms are the CC2420 radio and the MSP430 microcontroller. Chip code exists in `tos/chips`. A platform directory generally has a `.platform` file, which has options to pass to the nesC compiler. For example, the mica2 .platform file tells ncc to look in `chips/cc1000` and `chips/atm128` directories, as well as to use avr-gcc to compile a mote binary (Teloi platforms tell it to use msp430-gcc).

Hardware abstractions in TinyOS 2.0 generally follow a three-level abstaction heirarchy, called the HAA (Hardware Abstraction Architecture).

At the bottom of the HAA is the HPL (Hardware Presentation Layer). The HPL is a thin software layer on top of the raw hardware, presenting hardare such as IO pins or registers as nesC interfaces. The HPL generally has no state besides the hardware itself (it has no variables). HPL components usually have the prefix `Hpl`, followed by the name of the chip. For example, the HPL components of the CC1000 begin with `HplCC1000`.

The middle of the HAA is the HAL (Hardware Abstraction Layer). The HAL builds on top of the HPL and provides higher-level abstractions that are easier to use than the HPL but still provide the full functionality of the underlying hardware. The HAL components usually have a prefix of the chip name. For example, the HAL components of the CC1000 begin with `CC1000`.

The top of the HAA is the HIL (Hardware Independent Layer). The HIL builds on top of the HAL and provides abstractions that are hardware independent. This generalization means that the HIL usually does not provide all of the functionality that the HAL can. HIL components have no naming prefix, as they represent abstractions that applications can use and safely compile on multiple platforms. For example, the HIL component of the CC1000 on the mica2 is `ActiveMessageC`, representing a full active message communication layer.

The HAA is described in TEP 2: Hardware Abstraction Architecture[TEP2].

Currently (as of the 2.0 release in November 2006), TinyOS 2.0 supports the following platforms:

- eyesIFXv2
- intelmote2
- mica2
- mica2dot
- micaZ
- telosb
- tinynode
- btnode3

The btnode3 platform is not included in the RPM.

3. Scheduler

As with TinyOS 1.x, TinyOS 2.0 scheduler has a non-preemptive FIFO policy. However, tasks in 2.0 operate slightly differently than in 1.x.

In TinyOS 1.x, there is a shared task queue for all tasks, and a component can post a task multiple times. If the task queue is full, the post operation fails. Experience with networking stacks showed this to be problematic, as the task might signal completion of a split-phase operation: if the post fails, the component above might block forever, waiting for the completion event.

In TinyOS 2.x, every task has its own reserved slot in the task queue, and a task can only be posted once. A post fails if and only if the task has already been posted. If a component needs to post a task multiple times, it can set an internal state variable so that when the task executes, it reposts itself.

This slight change in semantics greatly simplifies a lot of component code. Rather than test to see if a task is posted already before posting it, a component can just post the task. Components do not have to try to recover from failed posts and retry. The cost is one byte of state per task. Even in large systems such as TinyDB, this cost is under one hundred bytes (in TinyDB is is approximately 50).

Applications can also replace the scheduler, if they wish. This allows programmers to try new scheduling policies, such as priority- or deadline-based. It is important to maintain non-preemptiveness, however, or the scheduler will break all nesC's static concurrency analysis. Details on the new scheduler and how to extend it can be found in TEP 106: Schedulers and Tasks[TEP106].

4. Booting/Initialization

TinyOS 2.0 has a different boot sequence than 1.x. The 1.x interface `StdControl` has been split into two interfaces: `Init` and `StdControl`. The latter only has two commands: `start` and `stop`. In TinyOS 1.x, wiring components to the boot sequence would cause them to be powered up and started at boot. That is no longer the case: the boot sequence only initializes components. When it has completed initializing the scheduler, hardware, and software, the boot sequence signals the `Boot.booted` event. The top-level application component handles this event and start services accordingly. Details on the new boot sequence can be found in TEP 107: TinyOS 2.x Boot Sequence[TEP107].

5. Virtualization

TinyOS 2.0 is written with nesC 1.2, which introduces the concept of a 'generic' or instantiable component. Generic modules allow TinyOS to have reusable data structures, such as bit vectors and queues, which simplify development. More importantly, generic configurations allow services to encapsulate complex wiring relationships for clients that need them.

In practice, this means that many basic TinyOS services are now *virtualized.* Rather than wire to a component with a parameterized interface (e.g., GenericComm or TimerC in 1.x), a program instantiates a service component that provides the needed interface. This service component does all of the wiring underneath (e.g., in the case of timers, to a unique) automatically, reducing wiring mistakes and simplifying use of the abstraction.

6. Timers

TinyOS 2.0 provides a much richer set of timer interfaces than 1.x. Experience has shown that timers are one of the most critical abstractions a mote OS can provide, and so 2.0 expands the fidelity and form that timers take. Depending on the hardware resources of a platform, a component can use 32KHz as well as millisecond granularity timers, and the timer system may provide one or two high-precision timers that fire asynchronously (they have the async keyword). Components can query their timers for how much time remainins before they fire, and can start timers in the future (e.g., 'start firing a timer at 1Hz starting 31ms from now'). TEP 102: Timers[TEP102] defines what HIL components a platform must provide in order to support standard TinyOS timers. Platforms are required to provide millisecond granularity timers, and can provide finer granularity timers (e.g., 32kHz) if needed.

Timers present a good example of virtualization in 2.0. In 1.x, a program instantiates a timer by wiring to TimerC:

```
components App, TimerC;
App.Timer -> TimerC.Timer[unique("Timer")];
```

In 2.0, a program instantiates a timer:

```
components App, new TimerMilliC();
App.Timer -> TimerMilliC;
```

7. Communication

In TinyOS 2.0, the message buffer type is `message_t`, and it is a buffer that is large enough to hold a packet from any of a node's communication interfaces. The structure itself is completely opaque: components cannot reference its fields. Instead, all buffer accesses go through interfaces. For example, to get the destination address of an AM packet named `msg`, a component calls `AMPacket.destination(msg)`.

Send interfaces distinguish the addressing mode of communication abstractions. For example, active message communication has the `AMSend` interface, as sending a packet require an AM destination address. In contrast, broadcasting and collection tree abstractions have the address-free `Send` interface.

Active messages are the network HIL. A platform's `ActiveMessageC` component defines which network interface is the standard communication medium. For example, a mica2 defines the CC1000 active message layer as ActiveMessageC, while the TMote defines the CC2420 active message layer as ActiveMessageC.

There is no longer a TOS_UART_ADDRESS for active message communication. Instead, a component should wire to SerialActiveMessageC, which provides active message communication over the serial port.

Active message communication is virtualized through four generic components, which take the AM type as a parameter: AMSenderC, AMReceiverC, AMSnooperC, and AMSnoopingReceiverC. AMSenderC is virtualized in that the call to send() does not fail if some other component is sending (as it does with GenericComm in 1.x). Instead, it fails only if that particular AMSenderC already has a packet outstanding or if the radio is not in a sending state. Underneath, the active message system queues and sends these outstanding packets. This is different than the QueuedSendC approach of 1.x, in which there is an N-deep queue that is shared among all senders. With N AMSenderC components, there is an N-deep queue where each sender has a single reserved entry. This means that each AMSenderC receives 1/n of the available post-MAC transmission opportunities, where n is the number of AMSenderC components with outstanding packets. In the worst case, n is the number of components; even when every protocol and component that sends packets is trying to send a packet, each one will receive its fair share of transmission opportunities.

Further information on message_t can be found in TEP 111: message_t[TEP111], while further information on AM can be found in TEP 116: Packet Protocols[TEP116].

The current TinyOS release has a low-power stack for the CC1000 radio (mica2 platform) and an experimental low-power stack for the CC2420 radio (micaz, telosb, and intelmote2 platforms).

8. Sensors

In TinyOS 2.0, named sensor components comprise the HIL of a platform's sensors. TEP 114 describes a set of HIL data acquisition interfaces, such as Read, ReadStream, and Get, which sensors provide according to their acquisition capabilities.

If a component needs high-frequency or very accurate sampling, it must use the HAL, which gives it the full power of the underlying platform (highly accurate platform-independent sampling is not really feasible, due to the particulars of individual platforms). `Read` assumes that the request can tolerate some latencies (for example, it might schedule competing requests using a FIFO policy).

Details on the ADC subsystem can be found in TEP 101: Analog-to-Digital Converters[TEP101]; details on the organization of sensor boards can be found in TEP 109: Sensorboards[TEP109], and the details of the HIL sensor interfaces can be found in TEP 114: Source and Sink Independent Drivers[TEP114].

9. Error Codes

The standard TinyOS 1.x return code is `result_t`, whose value is either SUCCESS (a non-zero value) or FAIL (a zero value). While this makes conditionals on calls very easy to write (e.g., `if (call A.b())`), it does not allow the callee to distinguish causes of error to the caller. In TinyOS 2.0, `result_t` is replaced by `error_t`, whose values include SUCCESS, FAIL, EBUSY, and ECANCEL. Interface commands and events define which error codes they may return and why.

From the perspective of porting code, this is the most significant different in 2.0. Calls that were once:

```
if (call X.y()) {
  busy = TRUE;
}
```

now have their meanings reversed. In 1.x, the busy statement will execute if the call succeeds, while in 2.0 it will execute if the call fails. This encourages a more portable, upgradable, and readable approach:

```
if (call X.y() == SUCCESS) {
  busy = TRUE;
}
```

10. Arbitration

While basic abstractions, such as packet communication and timers, can be virtualized, experiences with 1.x showed that some cannot without either adding significant complexity or limiting the system. The most pressing example of this is a shared bus on a microcontroller. Many different systems -- sensors, storage, the radio -- might need to use the bus at the same time, so some way of arbitrating access is needed.

To support these kinds of abstractions, TinyOS 2.0 introduces the Resource interface, which components use to request and acquire shared resources, and arbiters, which provide a policy for arbitrating access between multiple clients. For some abstractions, the arbiter also provides a power management policy, as it can tell when the system is no longer needed and can be safely turned off.

TEP 108: Resource Arbitration[TEP108] describes the Resource interface and how arbiters work.

11. Power Management

Power management in 2.0 is divided into two parts: the power state of the microcontroller and the power state of devices. The former, discussed in TEP 112: Microcontroller Power Management[TEP112], is computed in a chip-specific manner by examining which devices and interrupt souces are active. The latter, discussed in TEP 115: Power Management of Non-Virtualised Devices{TEP115], is handled through resource abiters. Fully virtualized services have their own, individual power management policies.

TinyOS 2.0 provides low-power stacks for the CC1000 (mica2) and CC2420 (micaz, telosb, imote2) radios. Both use a low-power listening apporach, where transmitters send long preambles or repeatedly send packets and receivers wake up periodically to sense the channel to hear if there is a packet being transmitted. The low-power stack CC1000 is standard, while the CC2420 stack is experimental. That is, the default CC1000 stack (chips/cc1000) has low-power-listening, while the default CC2420 stack (chips/cc2420) does not. To use the low-power CC2420 stack, you must include chips/cc2420_lpl in your application Makefile.

12. Network Protocols

TinyOS 2.0 provides simple reference implementations of two of the most basic protocols used in mote networks: dissemination and collection. Dissemination reliably delivers small (fewer than 20 byte) data items to every node in a network, while collection builds a routing tree rooted at a sink node. Together, these two protocols enable a wide range of data collection applications. Collection has advanced significantly since the most recent beta release; experimental tests in multiple network conditions have seen very high (>98%) deliver rates as long as the network is not saturated.

13. Conclusion

TinyOS 2.0 represents the next step of TinyOS development. Building on user experiences over the past few years, it has taken the basic TinyOS architecture and pushed it forward in several directions, hopefully leading to simpler and easier application development. It is still under active development: future prereleases will include non-volatile storage, basic multihop protocols (collection routing, dissemination), and further power management abstractions.

14. Acknowledgments

TinyOS 2.0 is the result of a lot of hard work from a lot of people, including (but not limited to) David Gay, Philip Levis, Cory Sharp, Vlado Handziski, Jan Hauer, Kevin Klues, Joe Polastre, Jonathan Hui, Prabal Dutta, Gilman Tolle, Martin Turon, Phil Buonodonna, Ben Greenstein, David Culler, Kristin Wright, Ion Yannopoulos, Henri Dubois-Ferriere, Jan Beutel, Robert Szewczyk, Rodrigo Fonseca, Kyle Jamieson, Omprakash Gnawali, David Moss, and Kristin Wright.

15. Author's Address

Philip Levis
358 Gates
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

phone - +1 650 725 9046

email - pal@cs.stanford.edu

16. Citations

[TEP1] TEP 1: TEP Structure and Keywords. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep1.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP2] TEP 2: Hardware Abstraction Architecture. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep2.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP3] TEP 3: Coding Standard. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep3.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP101] TEP 101: Analog-to-Digital Converters. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep101.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP102] TEP 102: Timers. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep102.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP106] TEP 106: Schedulers and Tasks. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep106.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP107] TEP 107: Boot Sequence. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep107.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP108] TEP 108: message_t. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep108.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP109] TEP 109: Sensorboards. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep109.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP110] TEP 110: Service Distributions. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep110.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP111] TEP 111: message_t. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep111.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP112] TEP 112: Microcontroller Power Management. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep112.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP113] TEP 113: Serial Communication. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep113.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP114] TEP 114: SIDs: Source and Sink Independent Drivers. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep114.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP115] TEP 115: Power Management of Non-Virtualised Devices. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep115.html?pathrev=tinyos-2_0_devel-BRANCH

[TEP116] TEP 116: Packet Protocols. http://tinyos.cvs.sourceforge.net/*checkout*/tinyos/tinyos-2.x/doc/html/tep116.html?pathrev=tinyos-2_0_devel-BRANCH