SIDs: Source and Sink Independent Drivers

TEP:	114
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	Gilman Tolle, Philip Levis, and David Gay

Abstract

This memo documents a set of hardware- and sensor-independent interfaces for data sources and sinks in TinyOS 2.x.

1. Introduction

Sensing is an integral part of any sensor network application. The diversity of sensors can lead to a wide variety of different software interfaces to these sensors. However, the burden of connecting a general sensor data management application to every one of these different interfaces suggests that sensors also provide a simple, general-purpose interface for data acquisition. Therefore, TinyOS 2.0 has telescoping sensor abstractions, providing both sensor-independent and sensor-specific interfaces. This memo documents a set of hardware- and sensor-independent interfaces for data sources and sinks in TinyOS 2.x.

2. Sensors in TinyOS 1.x

Early TinyOS sensors were generally analog. To sample one of these sensors, an application makes an analog-to-digital conversion using the MCU ADC. Because all early sensors required ADC conversions, the ADC interface has become the de-facto 1.x sensor interface. However, the ADC interface was originally designed for inexpensive, interrupt-driven sampling. All of its commands and events are async and sensor values are always 16 bits, although only some subset of the bits may be significant (e.g., a 12-bit value).

Because sensing is an integral part of high-level application logic, having asynchronous events means that high-level components must work with atomic sections, even if the sampling rate is very low (e.g., every five minutes) and so could be easily placed in a task. Race conditions are problematic and possible in any real time multitasking design. Race conditions are a failure in design, and especially difficult to detect at low sampling rates.

Additionally, not all sensors require ADC conversions from the MCU. Many sensors today are digital. To sample these sensors, the MCU sends a sample command and receives the corresponding data over a bus (e.g., SPI, I2C). The latency involved, combined with possible Resource arbitration[1], means that these bus operations are often synchronous code. In the command direction, this can force a task allocation to convert async to sync; in the event direction, the application has to deal with async code even though the event is, in practice, in a task.

Finally, the simplicity of the ADC interface has led many sensor modules to introduce several new interfaces for calibration and control, such as `Mic` and `MagSetting`. Because ADCs generally do not have error conditions, the ADC interface has no way to signal that a sample failed. This turns out to be important for sensors where the sampling request is split-phase, such as sensors over a bus. In these cases, it is possible that the driver accepts the request to sample, but once acquiring the bus discovers something is wrong with the sensor. This property has led bus-based sensors to also have a separate `ADCError` interface; this interface breaks the basic TinyOS pattern of a tight coupling between split-phase commands and their completion events, as the command is in ADC but the completion event is in ADCError.

All of these complications provide the context of the challenge to write high-level code that is sensor independent. Sensors, when possible, should follow an approach similar to the HAA[_haa], where they have sensor- or sensor-class-specific interfaces for high performance or special case use, but also simple and common interfaces for basic and portable use. Providing a telescoping sensor abstraction allows both classes of use.

3. Sensors in TinyOS 2.x

TinyOS 2.x contains several nesC interfaces that can be used to provide sensor-independent interfaces which cover a range of common use cases. This document describes these interfaces, and explains how to use these interfaces to write a Source- or Sink-Independent Driver (SID). A SID is source/sink independent because its interfaces do not themselves contain information on the sort of sensor or device they sit on top of. A SID SHOULD provide one or more of the interfaces described in this section.

This table summarizes the SID interfaces:

Name	Phase	Data type	Section
Read	Split	Scalar	3.1
Get	Single	Scalar	3.2
Notify	Trigger	Scalar	3.3
ReadStream	Split	Stream	3.4

3.1 Split-Phase Small Scalar I/O

The first set of interfaces can be used for low-rate scalar I/O:

```
interface Read<val_t> {
  command error_t read();
  event void readDone( error_t result, val_t val );
}
```

If the `result` parameter of the `Read.readDone` and `ReadWrite.readDone` events is not SUCCESS, then the memory of the `val` parameter MUST be filled with zeroes.

If the call to `read` has returned SUCCESS, but the `readDone` event has not yet been signaled, then a subsequent call to `read` MUST return EBUSY or FAIL. This simple locking technique, as opposed to a more complex system in which multiple read/readDone pairs may be outstanding, is intended to reduce the complexity of SID client code.

Examples of sensors that would be suited to this class of interface include many basic sensors, such as photo, temp, voltage, and ADC readings.

3.2 Single-Phase Scalar I/O

Some devices may have their state cached or readily available. In these cases, the device can provide a single-phase instead of split-phase operation. Examples include a node's MAC address (which the radio stack caches in memory), profiling information (e.g., packets received), or a GPIO pin. These devices MAY use the Get interface:

```
interface Get<val_t> {
  command val_t get();
}
```

3.3 Notification-Based Scalar I/O

Some sensor devices represent triggers, rather than request-driven data acquisition. Examples of such sensors include switches, passive-IR (PIR) motion sensors, tone detectors, and smoke detectors. This class of event-driven sensors can be presented with the Notify interface:

```
interface Notify<val_t> {
  command error_t enable();
  command error_t disable();
  event void notify( val_t val );
}
```

The Notify interface is intended for relatively low-rate events (e.g., that can easily tolerate task latencies). High-rate events may require more platform- or hardware-specific async interfaces.

The enable() and disable() command enable and disable notification events for the interface instance used by a single particular client. They are distinct from the sensor's power state. For example, if an enabled sensor is powered down, then when powered up it will remain enabled.

If `enable` returns SUCCESS, the interface MUST subsequently signal notifications when appropriate. If `disable` returns SUCCESS, the interface MUST NOT signal any notifications.

The val parameter is used as defined in the Read interface.

3.4 Split-Phase Streaming I/O

Some sensors can provide a continuous stream of readings, and some actuators can accept a continuous stream of new data. Depending on the rate needed and jitter bounds that higher level components can tolerate, it can be useful to be able to read or write readings in blocks instead of singly. For example, a microphone or accelerometer may provide data at a high rate that cannot be processed quickly enough when each new reading must be transferred from asynchronous to synchronous context through the task queue.

The ReadStream interface MAY be provided by a device that can provide a continuous stream of readings:

```
interface ReadStream<val_t> {

  command error_t postBuffer( val_t* buf, uint16_t count );

  command error_t read( uint32_t usPeriod );

  event void bufferDone( error_t result,
                         val_t* buf, uint16_t count );

  event void readDone( error_t result );
}
```

The postBuffer command takes an array parameterized by the sample type, and the number of entries in that buffer. A driver can then enqueue the buffer for filling. The client can call postBuffer() more than once, to "pre-fill" the queue with any number of buffers. The size of the memory region pointed to by the buf parameter MUST be at least as large as the size of a pointer on the node architecture plus the size of the uint16_t count argument. This requirement supports drivers that may store the queue of buffers and count sizes by building a linked list.

After posting at least one buffer, the client can call read() with a specified sample period in terms of microseconds. The driver then begins to fill the buffers in the queue, signaling the bufferDone() event when a buffer has been filled. The client MAY call postBuffer() after read() in order to provide the device with new storage for future reads.

If the device ever takes a sample that it cannot store (e.g., due to buffer underrun), it MUST signal readDone() with an appropriate failure code. If an error occurs during a read, then the device MUST signal readDone() with an appropriate failure code. Before a device signals readDone(), it MUST signal bufferDone() for all outstanding buffers. If a readDone() is pending, calls to postBuffer MUST return FAIL.

In the ReadStream interface, `postBuffer` returns SUCCESS if the buffer was successfully added to the queue, FAIL otherwise. A return value of SUCCESS from `read` indicates reading has begun and the interface will signal `bufferDone` and/or `readDone` in the future. A return value of FAIL means the read did not begin and

the interface MUST NOT signal `readDone` or `bufferDone`. Calls to `read` MAY return EBUSY if the component cannot service the request.

4. Implementation

An implementation of the Read interface can be found in `tos/system/SineSensorC.nc` and `tos/system/ArbitratedReadC.nc`.

An implementation of the Get interface can be found in `tos/platforms/telosb/UserButtonC.nc`.

An implementation of the ReadStream interface can be found in `tos/sensorboards/mts300/MageXStrea`

Implementations of the Notify interface can be found in `tos/platforms/telosb/SwitchToggleC.nc` and `tos/platforms/telosb/UserButtonP.nc`.

5. Summary

According to the design principles described in the HAA[_haa], authors should write device drivers that provide rich, device-specific interfaces that expose the full capabilities of each device. In addition, authors can use the interfaces described in this memo to provide a higher-level device-independent abstractions: SIDs. By providing such an abstraction, driver authors can support developers who only need simple interfaces, and can reduce the effort needed to connect a sensor into a more general system.

6. Author's Address

Gilman Tolle
501 2nd St. Ste 410
Arch Rock Corporation
San Francisco, CA 94107

phone - +1 415 692 0828
email - gtolle@archrock.com

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

phone - +1 650 725 9046
email - pal@cs.stanford.edu

David Gay
2150 Shattuck Ave, Suite 1300

Intel Research
Berkeley, CA 94704

phone - +1 510 495 3055
email - david.e.gay@intel.com

7. Citations

[1] TEP 108: Resource Arbitration.