SIDs: Source and Sink Independent Drivers

TEP:	114
Group:	Core Working Group
Type:	Documentary
Status:	Draft
TinyOS-Version:	2.x
Author:	Gilman Tolle, Philip Levis, and David Gay
Draft-Created:	30-Oct-2005
Draft-Version:	1.7
Draft-Modified:	2007-01-10
Draft-Discuss:	TinyOS Developer List <tinyos-devel at mail.millennium.berkeley.edu>

Note

This memo documents a part of TinyOS for the TinyOS Community, and requests discussion and suggestions for improvements. Distribution of this memo is unlimited. This memo is in full compliance with TEP 1.

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. Having a wide variety of sensor interfaces usually does not impose a large burden on an application developer, as any given application uses a small and static set. However, applications often build on top of more general systems, such as management or database layers, which may need to sample sensors. Since these are general and sensor-independent systems, they require a sensor-independent interface. TinyOS 2.0 therefore has telescoping sensor abstractions, providing both simple and sensor-independent as well as sensor-specific interfaces.

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 can be a part of high-level application logic, the asynchronicity of these events means that high-level components must deal with atomic sections and possible race conditions, even if the sampling rate is very low (e.g., every five minutes) and so could be easily placed in a task.

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 sensors to introduce several new ones 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 can make it difficult to write high-level code that is sensor independent, unless the sensor is a simple ADC reading. 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 has several sensor-independent interfaces, which cover a range of common use cases. These interfaces can be used 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, depending on its expected uses and underlying data model.

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 );
}

interface Write<val_t> {
  command error_t write( val_t val );
  event void writeDone( error_t result );
}
```

A component that provides both read and write functionality might want to use a combined version of the interface, to reduce the amount of wiring required by the client application:

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

  command error_t write( val_t val );
```

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

A component that can support concurrent reads and writes SHOULD provide separate Read/Write interfaces. A component whose internal logic will cause a read to fail while a write is pending or a write to fail while a read is pending MUST provide a ReadWrite interface.

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.read` has returned SUCCESS, but the `Read.readDone` event has not yet been signalled, then a subsequent call to `Read.read` MUST not return SUCCESS. 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 code that is a client of a SID interface.

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 Split-Phase Large Scalar I/O

If the SID's data object is too big to be passed efficienctly on the stack, it can provide read/write interfaces that pass parameters by pointer rather than value:

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

interface WriteRef<val_t> {
  command error_t write( val_t* val );
  event void writeDone( error_t result, val_t* val );
}

interface ReadWriteRef<val_t> {
  command error_t read( val_t* val );
  event void readDone( error_t result, val_t* val );

  command error_t write( val_t* val );
  event void writeDone( error_t result, val_t* val );
}
```

The caller is responsible for allocating storage pointed to by the val pointer which is passed to read() or write(). The SID MUST then take ownership of the storage, store a new value into it or copy a value out of it, and then relinquish it before signaling readDone() or writeDone(). If read or write() returns SUCCESS then the caller MUST NOT access or modify the storage pointed to by the val pointer until it handles the readDone() or writeDone() event.

As is the case with the parameters by value, whether a component provides separate ReadRef and WriteRef or ReadWriteRef affects the concurrency it allows. If a component can allow the two to execute concurrently, then it SHOULD provide separate ReadRef and WriteRef interfaces. If the two cannot occur concurrently, then it MUST provide ReadWriteRef.

If the `result` parameter of the `ReadRef.readDone` and `ReadWriteRef.readDone` events is not SUCCESS, then the memory the `val` parameter points to MUST be filled with zeroes.

Examples of sensors that are suited to this set of interfaces include those that generate multiple simultaneous readings for which passing by value is inefficient, such as a two-axis digital accelerometer.

3.4 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 these single-phase interfaces:

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

interface Set<val_t> {
  command void set( val_t val );
}

interface GetSet<val_t> {
  command val_t get();
  command void set( val_t val );
}
```

If a device's data object is readily available but still too large to be passed on the stack, then the device MAY use these interfaces:

```
interface GetRef<val_t> {
  command error_t get( val_t* val );
}

interface SetRef<val_t> {
  command error_t set( val_t* val );
}

interface GetSetRef<val_t> {
  command error_t get( val_t* val );
  command error_t set( val_t* val );
}
```

3.5 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 MUST remain enabled.

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

3.7 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 ReadStreaming 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, signalling 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., no buffers are available), it MUST signal readDone(). 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.

The following interface can be used for bulk writes:

```
interface WriteStream<val_t> {

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

  command error_t write( uint32_t period );

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

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

postBuffer() and bufferDone() are matched in the same way described for the ReadStream interface, as are write() and writeDone().

4. 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.

5. Author's Address

Gilman Tolle
2168 Shattuck Ave.
Arched Rock Corporation
Berkeley, CA 94704

phone - +1 510 981 8714
email - gtolle@archedrock.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

6. Citations

[1] TEP 108: Resource Arbitration.