Standard TinyOS Sensorboard Interface V1.1

David Gay, Wei Hong, Phil Levis, and Joe Polastre
dgay@intel-research.net, pal@cs.berkeley.edu, jpolastre@cs.berkeley.edu

February 18, 2004

1 Introduction

This document defines the "standard" interface a sensor board is expected to offer in TinyOS. As this is still relatively early days, there will undoubtably be sensor boards which cannot conform to this specification, but such boards should attempt to follow its spirit as closely as possible.

This standard assumes that sensors return uninterpreted 16-bit values, and, optionally uninterpreted, arbitrary-size calibration data. Conversion of sensor values to something with actual physical meaning is beyond the (current) scope of this document.

This standard departs from current conventions. Sensor board components and applications will need to be converted.

2 Directory Organisation

1. A sensor board should have a unique name, composed of letters, numbers and underscores. Case is significant, but two sensor boards must differ in more than case.[1] We will use S to denote the sensor board name in the rest of this document.
2. Each sensor board should have its own directory named S; standard TinyOS sensor boards will be placed in `tinyos-1.x/tos/sensorboards`, but sensor board directories can be placed anywhere as long as the nesC compiler receives a `-I` directive pointing to the sensor board's directory. [2].
3. Each sensor board directory must contain a `.sensor` file. This file is a perl script which contains any additional compiler settings needed for this sensor board (this file will be empty in many cases).
4. If the sensor board wishes to define any C types or constants, it should place these in a file named S`.h` in the sensor board's directory.
5. The sensor board directory should contain *sensor board components* for accessing each sensor on the sensor board. The conventions for these components are detailed in Section 3.
6. A sensor board may include additional components providing alternative or higher-level interfaces to the sensors (e.g., for TinyDB). These components are beyond the scope of this document. Future versions of this standard are likely to discuss TinyDB attributes.
7. Finally, the sensor board can contain any number of components, interfaces, C files, etc for internal use. To avoid name collisions, all externally visible names (interface types, components, C constants and types) used for internal purposes should be prefixed with S.

A simple example: the basic sensor board is named `basicsb`, it's directory is

`tinyos − 1.x/tos/sensorboards/basicsb`

[1]Necessary to support platforms where filename case differences are not significant.
[2]This is supported in v1.1.1 and later of the nesC compiler

It has no basicsb.h file and its .sensor file is empty. It has two components, Photo and Temp representing its two sensors.

3 Sensor Board Components

We have not yet selected any naming conventions for sensor board components. Please select reasonable names. . .

A sensor board component must provide a StdControl or SplitControl interface for initialisation and power management, some set of Sensor or QSensor interfaces for sampling, and, optionally, some set of CalibrationData interfaces for obtaining calibration data. These interfaces are shown in Figure 1. A component can provide additional interfaces for other purposes; these are beyond the scope of this document. A sensor board component should be as lightweight as possible - it should just provide basic access to the physical sensors and not attempt to do calibration, signal processing, etc. If such functionality is desired, it should be provided in separate components.

If a Sensor or QSensor interface named X has a corresponding calibration interface, that interface should be called XCalibration.

The commands and events in the QSensor interface are marked async, i.e., may execute as part of an interrupt handler. This is to support applications that require, and sensors that provide, low-jitter sampling. Figure 2 shows a component that converts a QSensor interface into a regular Sensor interface.

The Sensor and QSensor interfaces return uninterpreted 16-bit data. This might represent an A/D conversion result, a counter, etc. The optional calibration interface returns uninterpreted, arbitrary-size data.

Some common setups for sensor board components are:

- A single Sensor (or QSensor) interface. This is probably the most common case, where a single component corresponds to a single physical sensor, e.g., for light, temperature, pressure and there is no expectation of high sample rates.
- Multiple Sensor (or QSensor) interfaces. Some sensors might be strongly related, e.g., the axes of an accelerometer. A single component could then provide a sensor interface for each axis. For instance, a 2-axis accelerometer which can be sampled at high speed, and which has some calibration data might be declared with:

```
configuration Accelerometer2D {
  provides {
    interface StdControl
    interface QSensor as AccelX;
    interface CalibrationData as AccelXCalibration;
    interface QSensor as AccelY;
    interface CalibrationData as AccelYCalibration;
  }
}
```

- A parameterised Sensor (or QSensor) interface. If a sensor board has multiple similar sensors, it may make sense to provide a single component to access all of these, using a parameterised Sensor interface. For instance, a general purpose sensor board with multiple A/D channels might provide an Sensor interface parameterised by the A/D channel id.

Sensor board components are expected to respect the following conventions on the use of the StdControl, SplitControl and Sensor interfaces. These are given assuming StdControl is used, but the behaviour with SplitControl is identical except that init, start and stop are not considered complete until the initDone, startDone and stopDone events are signaled. The conventions are:

1. StdControl.init: must be called at mote boot time.

```
interface Sensor {
  /* Sensor is for sensors which will not be sampled at high rates */

  /** Request sensor sample
   *  @return SUCCESS if request accepted, FAIL if it is refused
   *    dataReady or error will be signaled if SUCCESS is returned
   */
  command result_t getData();

  /** Return sensor value
   * @param data Sensor value
   * @return Ignored
   */
  event result_t dataReady(uint16_t data);

  /** Signal that the sensor failed to get data
   * @param info error information, sensor board specific
   * @return Ignored
   */
  event result_t error(uint16_t info);
}

interface QSensor {
  /* QSensor is for sensors which need to be sampled at high rates or
     at precise times.
     The only difference with Sensor is that the getData/dataReady
     functions can be called from interrupt handlers. */
  async command result_t getData();
  async event result_t dataReady(uint16_t data);
  event result_t error(uint16_t errorInformation);
}

interface CalibrationData {
  /* Collect uninterpreted calibration data from a sensor */

  /** Request calibration data
   *  @return SUCCESS if request accepted, FAIL if it is refused
   *    data error will be signaled if SUCCESS is returned
   */
  command result_t get();

  /** Returns calibration data
   * @param x Pointer to (uinterpreted) calibration data. This data
   *   must not be modified.
   * @param len Length of calibration data
   * @return Ignored.
   */
  event result_t data(const void *x, uint8_t len);
}
```

Figure 1: Sensorboard interfaces

```
module QSensorAsSensor {
  provides interface Sensor;
  uses interface QSensor;
} implementation {
  uint16_t temp;

  command result_t Sensor.getData() {
    return QSensor.getData();
  }

  task void ready() {
    signal Sensor.dataReady(temp);
  }

  async command result_t QSensor.dataReady(uint16_t data) {
    temp = data;
    post ready();
    return SUCCESS;
  }

  command result_t QSensor.error(uint16_t info) {
    return Sensor.error(info);
  }
}
```

Figure 2: Convert a `QSensor` to a `Sensor` interface

2. `StdControl.start`: ensure the sensor corresponding to this component is ready for use. For instance, this should power-up the sensor if necessary. The application can call `getData` once `StdControl.start` completes.

   If a sensor takes a while to power-up, the sensor board implementer can either use a `SplitControl` interface and signal `startDone` when the sensor is ready for use, or delay `dataReady` events until the sensor is ready. The former choice is preferable if the sensor is going to be used for high-frequency sampling.

3. `StdControl.stop`: put the sensor in a low-power mode. `StdControl.start` must be called before any further readings are taken. The behaviour of calls to `StdControl.stop` during sampling (i.e., when an `dataReady` event is going to be signaled) is undefined.

4. `Sensor/QSensor.getData`: get a sample from a sensor.

5. `Sensor/QSensor.dataReady(uint16_t data)`: signals the sample value to the application.

6. `Sensor/QSensor.error(uint16_t info)`: reports a sensing problem to the application (not all sensors will report errors). The values for `info` are sensor-board specific.

4 `.sensor` File

This file is a perl script which gets executed as part of the `ncc` nesC compiler frontend. It can add or modify any compile-time options necessary for a particular sensor board. It can modify the following perl variables:

- `@new_args`: This is the array of arguments which will be passed to `nescc`. For instance, you might add an include directive to `@new_args` with

$$\texttt{push @new\_args,'-Isomedir'}$$

- `@commonboards`: This can be set to a list of sensor board names which should be added to the include path list. These sensor boards must be in `tinyos-1.x/tos/sensorboards`.

5 Example: micasb

The mica sensor board (micasb) has five sensors (and one actuator, the sounder):

Name	Component	Sensor Interfaces	Other Interfaces
Accelerometer	Accel	AccelX AccelY	
Magnetometer	Mag	MagX MagY	MagSetting
Microphone	Mic	MicADC	Mic MicInterrupt
Light	Photo	PhotoADC	
Temperature	Temp	TempADC	

Each physical sensor is represented by a separate component. Specific sensors that have more than one axis of measurement (e.g., Accel and Mag) provide more than one Sensor interface on a single component. Some sensors, such as the magnetometer and microphone, have additional functionality provided through sensor-specific interfaces.

Although light and temperature are represented by separate components, in reality they share a single ADC pin. The two components Photo and Temp sit on top of the PhotoTemp component, which controls access to the shared pin, and orchestrates which sensor is currently connected to it. From a programmer's perspective, they appear as individual sensors, even though their underlying implementation is a bit more complex.

The board's `micasb.h` file contains private configuration data (pin usage, ADC ports, etc).

The mica sensor board has an empty `.sensor` file. For a more interesting example, refer to the micawbdot sensor board.