Sensors and Sensor Boards

TEP:	109
Group:	Core Working Group
Type:	Documentary
Status:	Final
TinyOS-Version:	2.x
Author:	David Gay, Philip Levis, Wei Hong, Joe Polastre, and Gilman Tolle

Abstract

This memo documents how sensor drivers are organized in TinyOS and how sets of sensor drivers are combined into sensor boards and sensor platforms, along with general principles followed by the components that provide access to sensors.

1. Principles

This section describes the basic organization principles for sensor drivers in TinyOS.

For background, a sensor can be attached to the microcontroller on a TinyOS platform through a few different types of connections:

- Included within the microcontroller itself
- Connected to general-purpose IO pins for level/edge detection
- Connected to an ADC in the microcontroller for voltage sampling
- Connected to general-purpose IO pins for digital communication
- Connected through a standard digital bus protocol (1-Wire, I2C, SPI)

Physically, these connections can also be decoupled by attaching the sensors to a *sensor board*, which can be removed from the TinyOS platform, and could attach to multiple different TinyOS platforms.

The capabilities of a physical sensor are made available to a TinyOS application through a *sensor driver*.

According to the HAA [TEP2], TinyOS devices SHOULD provide both simple hardware-independent interfaces for common-case use (HIL) and rich hardware-dependent interfaces for special-case use (HAL). Sensor drivers SHOULD follow this spirit as well.

TinyOS 2.x represents each sensor as an individual component. This allows the compilation process to minimize the amount of code included. A sensor board containing multiple sensors SHOULD be represented as a collection of components, one for each sensor, contained within a sensor board directory.

Sensors, being physical devices that can be shared, can benefit from virtualization and arbitration. This document describes a design pattern for sensor virtualization that SHOULD be followed by sensor drivers.

The same physical sensor can be attached to multiple different TinyOS platforms, through platform-dependent interconnections. The common logic of sensor driver SHOULD be factored into chip-dependent, platform-independent components, and those components SHOULD be bound to the hardware resources on a platform by platform-dependent components, and to the hardware resources on a sensor board by sensorboard-dependent components.

A physical sensor has a general class and a specific set of performance characteristics, captured by the make and model of the sensor itself. The naming of the sensor driver components SHOULD reflect the specifc name of the sensor, and MAY provide a component with a generic name for application authors who only care about the general class of the sensor.

This document requires that sensor components specify the range (in bits) of values returned by sensor drivers, but takes no position on the meaning of these values. They MAY be raw uninterpreted values or they MAY have some physical meaning. If a driver returns uninterpreted values, the driver MAY provide additional interfaces that would allow higher-level clients to obtain information (e.g. calibration coefficients) needed to properly interpret the value.

2. Sensor HIL Components

A sensor HIL component MUST provide:

- One or more SID interfaces [TEP114], for reading data.

A sensor HIL component MAY provide:

- One or more SID interfaces [TEP114], for reading or writing calibration coefficients or control registers.

A sensor device driver SHOULD be a generic component that virtualizes access to the sensor. A sensor device driver can provide such virtualization for itself by defining a nesC generic client component. When a client component is being used, a call to a top-level SID interface SHOULD be delayed when the device is busy, rather than failing. Using one of the system arbiters can make the implementation of this requirement easier to accomplish.

For example:

```
generic configuration SensirionSht11C() {
  provides interface Read<uint16_t> as Temperature;
  provides interface ReadStream<uint16_t> as TemperatureStream;
  provides interface DeviceMetadata as TemperatureDeviceMetadata;
```

```
    provides interface Read<uint16_t> as Humidity;
    provides interface ReadStream<uint16_t> as HumidityStream;
    provides interface DeviceMetadata as HumidityDeviceMetadata;
  }
  implementation {
    // connect to the ADC HIL, GPIO HAL, or sensor's HAL
  }
```

When a HIL component is being used, the sensor MUST initialize itself, either by including the *MainC* component and wiring to the *SoftwareInit* interface, or by allowing a lower-level component (like an ADC) to initialize itself.

In addition, the HIL sensor driver MUST start the physical sensor automatically. For sensors without a constant power draw, the sensor MAY be started once at boot time by wiring to the *MainC.Boot* interface. Sensors that draw appreciable power MUST be started in response to a call to one of the top-level SID interfaces, and stopped some time after that call completes. Using one of the power-management components described in [TEP115] can make this implementation easier.

Generally, simple types are made up of octets. However, sensor values often have levels of precision besides a multiple of 8. To account for such cases, each device MUST specify the precision of each one of its interfaces by providing the DeviceMetadata interface:

```
  interface DeviceMetadata {
    command uint8_t getSignificantBits();
  }
```

The name of the instance of DeviceMetadata MUST clearly indicate which interface it corresponds to.

The getSignificantBits() call MUST return the number of significant bits in the reading. For example, a sensor reading taken from a 12-bit ADC would typically return the value 12 (it might return less if, e.g., physical constraints limit the maximum A/D result to 10-bits).

Sensor driver components SHOULD be named according to the make and model of the sensing device being presented. Using specific names gives the developer the option to bind to a particular sensor, which provides compile-time detection of missing sensors. However, wrapper components using "common" names MAY also be provided by the driver author, to support application developers who are only concerned with the particular type of the sensor and not its make, model, or detailed performance characteristics.

A "common" naming layer atop a HIL might look like this:

```
  generic configuration TemperatureC() {
    provides interface Read<uint16_t>;
    provides interface ReadStream<uint16_t>;
    provides interface DeviceMetadata;
  }
  implementation {
    components new SensirionSht11C();
    Read = SensirionSht11C.Temperature;
    ReadStream = SensirionSht11C.TemperatureStream;
    DeviceMetadata = SensirionSht11C.TemperatureDeviceMetadata;
```

```
}

generic configuration HumidityC() {
  provides interface Read<uint16_t>;
  provides interface ReadStream<uint16_t>;
  provides interface DeviceMetadata;
}
implementation {
  components new SensirionSht11C();
  Read = SensirionSht11C.Humidity;
  ReadStream = SensirionSht11C.HumidityStream;
  DeviceMetadata = SensirionSht11C.HumidityDeviceMetadata;
}
```

3. Sensor HAL Components

Sensors with a richer interface than would be supported by the SID interfaces MAY provide a HAL component in addition to a HIL component.

A sensor HAL component MUST provide:

- A SID-based interface or a specific hardware-dependent interface with commands for sampling and controlling the sensor device.

A sensor HAL component MAY need to provide:

- A *StdControl* or *SplitControl* interface for manual power management by the user, following the conventions described in [TEP115].
- A *Resource* interface for requesting access to the device and possibly performing automated power management, following the conventions described in [TEP108] and [TEP115].
- Any other interfaces needed to control the device, e.g., to read or write calibration coefficients.

For example:

```
configuration SensirionSht11DeviceC {
  provides interface Resource[ uint8_t client ];
  provides interface SensirionSht11[ uint8_t client ];
}
implementation {
  // connect to the sensor's platform-dependent HPL here
}
```

4. Sensor Component Organization and Compiler Interaction Guidelines

Sensors are associated either with a particular sensor board or with a particular platform. Both sensors and sensor boards MUST have unique names. Case is significant, but two sensor (or sensor board) names MUST differ in more than case. This is necessary to support platforms where filename case differences are not significant.

Each sensor board MUST have its own directory whose name is the sensor board's unique name (referred to as <sensorboard> in the rest of this section). Default TinyOS 2.x sensor boards are placed in `tos/sensorboards/<sensorboard>`, 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. Each sensor board directory MUST contain a `.sensor` file (described below). If the sensor board wishes to define any C types or constants, it SHOULD place these in a file named `<sensorboard>.h` in the sensor board's directory.

A sensor board MAY contain components that override the default TinyOS *demo sensors*. This allows the sensor board to easily be used with TinyOS sample applications that use the demo sensors. If a sensor board wishes to override the default demo sensor:

- It MUST provide a generic component named `DemoSensorC` with the following signature:

  ```
  provides interface Read<uint16_t>;
  provides interface DeviceMetadata;
  ```

- It MAY provide a generic component named `DemoSensorNowC` with the following signature:

  ```
  provides interface ReadNow<uint16_t>;
  provides interface DeviceMetadata;
  ```

  This component SHOULD sample the same sensor as `DemoSensorC`.

- It MAY provide a generic component named `DemoSensorStreamC` with the following signature:

  ```
  provides interface ReadStream<uint16_t>;
  provides interface DeviceMetadata;
  ```

  This component SHOULD sample the same sensor as `DemoSensorC`.

These components MUST be an alias for one of the sensor board's usual sensors, though they change the precision of the sensor if necessary. For instance, if `DemoSensorC` is an alias for a 20-bit sensor that provides a `Read<uint32_t>` interface, `DemoSensorC` would still provide `Read<uint16_t>` and would include code to reduce the precision of the aliased sensor.

4.1 Compiler Interaction

When the `ncc` nesC compiler frontend is passed a `-board=X` option, it executes the `.sensor` file found in the sensor board directory `X`. This file is a perl script which can add or modify any compile-time options necessary for the sensor board. It MAY modify the following perl variables, and MUST NOT modify any others:

- `@includes`: This array contains the TinyOS search path, i.e., the directories which will be passed to nescc (the TinyOS-agnostic nesC compiler) as `-I` arguments. You MUST add to `@includes` any directories needed to compile this sensor board's components. For instance, if your sensor boards depends on support code found in `tos/chips/sht11`, you would add `"%T/chips/sht11"` to `@includes`.

- `@new_args`: This is the array of arguments which will be passed to nescc. You MUST add any arguments other than `-I` that are necessary to compile your sensor board components to `@new_args`.

If a sensor is associated with a platform *P* rather than a sensor board, then that platform MUST ensure that, when compiling for platform *P*, all directories needed to compile that sensor's component are added to the TinyOS search path (see [TEP131] for information on how to set up a TinyOS platform).

4.2 Sensor Components

A particular sensor is typically supported by many components, including the HIL and HAL components from Sections 2 and 3, A/D conversion components (for analog sensors), digital bus components (e.g., SPI, for digital sensors), system services (timers, resource and power management, ...), glue components (to connect sensors, sensor boards and platforms), etc. These components can be divided into three classes: sensorboard-dependent, platform-dependent and platform-independent. The sensorboard and platform MUST ensure (Section 4.1) that all these components can be found at compile-time.

Because the same physical sensor can be used on many platforms or sensor boards, and attached in many different ways, to maximize code reuse the organization of sensor drivers SHOULD reflect the distinction between sensor and sensor interconnect. The sensor components SHOULD be platform-independent, while the sensor interconnect components are typically sensorboard or platform-dependent. However, some sensors (e.g. analong sensors) will not have a sufficiently large amount of platform-independent logic to justify creating platform-independent components.

The following guidelines specify how to organize sensor and sensor interconnect components within TinyOS's directory hierarchy. These guidelines are only relevant to components that are part of the core source tree. The string `<sensor>` SHOULD reflect the make and model of the sensor device.

- Platform-independent sensor components that exist as part of a larger chip, like a MCU internal voltage sensor, SHOULD be placed in a subdirectory of the chip's directory `tos/<chip>/sensors/<sensor>`.
- Other platform-independent sensor components SHOULD be placed in `tos/chips/<sensor>`.
- Sensorboard-dependent sensor and sensor interconnect components SHOULD be placed either in the `<sensorboard>` directory or in a `<sensorboard>/chips/<sensor>` directory.
- Platform-dependent sensor and sensor interconnect components SHOULD be placed in `tos/<platform>/chips/<sensor>`.

5. Authors' Addresses

David Gay

2150 Shattuck Ave, Suite 1300
Intel Research
Berkeley, CA 94704

phone - +1 510 495 3055

email - david.e.gay@intel.com

Wei Hong
Arch Rock
657 Mission St. Suite 600
San Francisco, CA 94105

email - wei.hong@gmail.com

Philip Levis
358 Gates Hall
Computer Science Department
353 Serra Mall
Stanford, CA 94305

phone - +1 650 725 9046

email - pal@cs.stanford.edu

Joe Polastre
467 Soda Hall
UC Berkeley
Berkeley, CA 94720

email - polastre@cs.berkeley.edu

Gilman Tolle
Arch Rock
657 Mission St. Suite 600
San Francisco, CA 94105

email - gtolle@archrock.com

[TEP2] TEP 2: Hardware Abstraction Architecture

6. Citations

Appendix A: Sensor Driver Examples

1. Analog ADC-Connected Sensor

The Analog sensor requires two components

- a component to present the sensor itself (HamamatsuS1087ParC)
- a component to select the appropriate hardware resources, such as ADC port 4, reference voltage 1.5V, and a slow sample and hold time (HamamatsuS1087ParP).

The AdcReadClientC component and underlying machinery handles all of the arbitration and access to the ADC.

```
tos/platforms/telosa/chips/s1087/HamamatsuS1087ParC.nc

// HIL for the HamamatsuS1087 analog photodiode sensor
generic configuration HamamatsuS1087ParC() {
  provides interface Read<uint16_t>;
  provides interface ReadStream<uint16_t>;
  provides interface DeviceMetadata;
}
implementation {
  // Create a new A/D client and connect it to the Hamamatsu S1087 A/D
  // parameters
  components new AdcReadClientC();
  Read = AdcReadClientC;

  components new AdcReadStreamClientC();
  ReadStream = AdcReadStreamClientC;

  components HamamatsuS1087ParP;
  DeviceMetadata = HamamatsuS1087ParP;
  AdcReadClientC.AdcConfigure -> HamamatsuS1087ParP;
  AdcReadStreamClientC.AdcConfigure -> HamamatsuS1087ParP;
}


tos/platforms/telosa/chips/s1087/HamamatsuS1087ParP.nc

#include "Msp430Adc12.h"

// A/D parameters for the Hamamatsu - see the MSP430 A/D converter manual,
// Hamamatsu specification, Telos hardware schematic and TinyOS MSP430
// A/D converter component specifications for the explanation of these
// parameters
```

```
module HamamatsuS1087ParP {
  provides interface AdcConfigure<const msp430adc12_channel_config_t*>;
  provides interface DeviceMetadata;
}
implementation {
  msp430adc12_channel_config_t config = {
    inch: INPUT_CHANNEL_A4,
    sref: REFERENCE_VREFplus_AVss,
    ref2_5v: REFVOLT_LEVEL_1_5,
    adc12ssel: SHT_SOURCE_ACLK,
    adc12div: SHT_CLOCK_DIV_1,
    sht: SAMPLE_HOLD_4_CYCLES,
    sampcon_ssel: SAMPCON_SOURCE_SMCLK,
    sampcon_id: SAMPCON_CLOCK_DIV_1
  };

  async command const msp430adc12_channel_config_t* AdcConfigure.getConfigu
    return &config;
  }

  command uint8_t DeviceMetadata.getSignificantBits() { return 12; }
}
```

2. Binary Pin-Connected Sensor

The Binary sensor gets a bit more complex, because it has three components:

- one to present the sensor (UserButtonC)
- one to execute the driver logic (UserButtonLogicP)
- one to select the appropriate hardware resources, such as MSP430 Port 27 (HplUserButtonC).

Note that the presentation of this sensor is not arbitrated because none of the operations are split-phase.

```
tos/platforms/telosa/UserButtonC.nc

// HIL for the user button sensor on Telos-family motes
configuration UserButtonC {
  provides interface Get<bool>; // Get button status
  provides interface Notify<bool>; // Get button-press notifications
  provides interface DeviceMetadata;
}
implementation {

  // Simply connect the button logic to the button HPL
  components UserButtonLogicP;
  Get = UserButtonLogicP;
  Notify = UserButtonLogicP;
  DeviceMetadata = UserButtonLogicP;

  components HplUserButtonC;
  UserButtonLogicP.GpioInterrupt -> HplUserButtonC.GpioInterrupt;
  UserButtonLogicP.GeneralIO -> HplUserButtonC.GeneralIO;
}

tos/platforms/telosa/UserButtonLogicP.nc

// Transform the low-level (GeneralIO and GpioInterrupt) interface to the
// button to high-level SID interfaces
module UserButtonLogicP {
  provides interface Get<bool>;
  provides interface Notify<bool>;
  provides interface DeviceMetadata;

  uses interface GeneralIO;
  uses interface GpioInterrupt;
}
implementation {
  norace bool m_pinHigh;

  task void sendEvent();

  command bool Get.get() { return call GeneralIO.get(); }

  command error_t Notify.enable() {
    call GeneralIO.makeInput();

    // If the pin is high, we need to trigger on falling edge interrupt, ar
    // vice-versa
    if ( call GeneralIO.get() ) {
      m_pinHigh = TRUE;
      return call GpioInterrupt.enableFallingEdge();
    } else {
      m_pinHigh = FALSE;
      return call GpioInterrupt.enableRisingEdge();
    }
  }

  command error_t Notify.disable() {
    return call GpioInterrupt.disable();
  }

  // Button changed, signal user (in a task) and update interrupt detectior
  async event void GpioInterrupt.fired() {
    call GpioInterrupt.disable();

    m_pinHigh = !m_pinHigh;

    post sendEvent();
  }

  task void sendEvent() {
    bool pinHigh;
    pinHigh = m_pinHigh;

    signal Notify.notify( pinHigh );

    if ( pinHigh ) {
      call GpioInterrupt.enableFallingEdge();
    } else {
      call GpioInterrupt.enableRisingEdge();
    }
  }

  command uint8_t DeviceMetadata.getSignificantBits() { return 1; }
}
```

```
tos/platforms/telosa/HplUserButtonC.nc

// HPL for the user button sensor on Telos-family motes - just provides
// access to the I/O and interrupt control for the pin to which the
// button is connected
configuration HplUserButtonC {
  provides interface GeneralIO;
  provides interface GpioInterrupt;
}
implementation {

  components HplMsp430GeneralIOC as GeneralIOC;

  components new Msp430GpioC() as UserButtonC;
  UserButtonC -> GeneralIOC.Port27;
  GeneralIO = UserButtonC;

  components HplMsp430InterruptC as InterruptC;

  components new Msp430InterruptC() as InterruptUserButtonC;
  InterruptUserButtonC.HplInterrupt -> InterruptC.Port27;
  GpioInterrupt = InterruptUserButtonC.Interrupt;
}
```

[TEP131] TEP 131: Creating a New Platform for TinyOS 2.x

3. Digital Bus-Connected Sensor

The Digital sensor is the most complex out of the set, and includes six components:

- one to present the sensor (SensirionSht11C)
- one to request arbitrated access and to transform the sensor HAL into the sensor HIL (SensirionSht11P)
- one to present the sensor HAL (HalSensirionSht11C)
- one to perform the driver logic needed to support the HAL, which twiddles pins according to a sensor-specific protocol (SensirionSht11LogicP).
- one to select the appropriate hardware resources, such as the clock, data, and power pins, and to provide an arbiter for the sensor (HplSensirionSht11C).
- one to perform the power control logic needed to support the power manager associated with the arbiter (HplSensirionSht11P).

This bus-connected sensor is overly complex because it does not rely on a shared framework of bus manipulation components. A sensor built on top of the I2C or SPI bus would likely require fewer components.

```
tos/platforms/telosa/chips/sht11/SensirionSht11C.nc

// HIL interface to Sensirion SHT11 temperature and humidity sensor
generic configuration SensirionSht11C() {
  provides interface Read<uint16_t> as Temperature;
  provides interface DeviceMetadata as TemperatureDeviceMetadata;
  provides interface Read<uint16_t> as Humidity;
  provides interface DeviceMetadata as HumidityDeviceMetadata;
}
implementation {
  // Instantiate the module providing the HIL interfaces
  components new SensirionSht11ReaderP();

  Temperature = SensirionSht11ReaderP.Temperature;
  TemperatureDeviceMetadata = SensirionSht11ReaderP.TemperatureDeviceMetada
  Humidity = SensirionSht11ReaderP.Humidity;
  HumidityDeviceMetadata = SensirionSht11ReaderP.HumidityDeviceMetadata;

  // And connect it to the HAL component for the Sensirion SHT11
  components HalSensirionSht11C;

  enum { TEMP_KEY = unique("Sht11.Resource") };
  enum { HUM_KEY = unique("Sht11.Resource") };

  SensirionSht11ReaderP.TempResource -> HalSensirionSht11C.Resource[ TEMP_
  SensirionSht11ReaderP.Sht11Temp -> HalSensirionSht11C.SensirionSht11[ TEM
  SensirionSht11ReaderP.HumResource -> HalSensirionSht11C.Resource[ HUM_KEY
  SensirionSht11ReaderP.Sht11Hum -> HalSensirionSht11C.SensirionSht11[ HUM_
}
```

```
tos/chips/sht11/SensirionSht11ReaderP.nc

// Convert Sensirion SHT11 HAL to HIL interfaces for a single
// client, performing automatic resource arbitration
generic module SensirionSht11ReaderP() {
  provides interface Read<uint16_t> as Temperature;
  provides interface DeviceMetadata as TemperatureDeviceMetadata;
  provides interface Read<uint16_t> as Humidity;
  provides interface DeviceMetadata as HumidityDeviceMetadata;

  // Using separate resource interfaces for temperature and humidity allows
  // temperature and humidity measurements to be requested simultaneously
  // (if a single Resource interface was used, a request for temperature wo
  // prevent any humidity requests until the temperature measurement was co
  uses interface Resource as TempResource;
  uses interface Resource as HumResource;
  uses interface SensirionSht11 as Sht11Temp;
  uses interface SensirionSht11 as Sht11Hum;
}
implementation {
  command error_t Temperature.read() {
    // Start by requesting access to the SHT11
    return call TempResource.request();
  }

  event void TempResource.granted() {
    error_t result;
    // If the HAL measurement fails, release the SHT11 and signal failure
    if ((result = call Sht11Temp.measureTemperature()) != SUCCESS) {
      call TempResource.release();
      signal Temperature.readDone( result, 0 );
    }
  }

  event void Sht11Temp.measureTemperatureDone( error_t result, uint16_t val
    // Release the SHT11 and signal the result
    call TempResource.release();
    signal Temperature.readDone( result, val );
  }

  command uint8_t TemperatureDeviceMetadata.getSignificantBits() { return 1

  command error_t Humidity.read() {
    // Start by requesting access to the SHT11
    return call HumResource.request();
  }

  event void HumResource.granted() {
    error_t result;
    // If the HAL measurement fails, release the SHT11 and signal failure
    if ((result = call Sht11Hum.measureHumidity()) != SUCCESS) {
      call HumResource.release();
      signal Humidity.readDone( result, 0 );
    }
  }

  event void Sht11Hum.measureHumidityDone( error_t result, uint16_t val ) {
    // Release the SHT11 and signal the result
    call HumResource.release();
    signal Humidity.readDone( result, val );
  }

  command uint8_t HumidityDeviceMetadata.getSignificantBits() { return 12;

  // Dummy handlers for unused portions of the HAL interface
  event void Sht11Temp.resetDone( error_t result ) { }
  event void Sht11Temp.measureHumidityDone( error_t result, uint16_t val )
  event void Sht11Temp.readStatusRegDone( error_t result, uint8_t val ) { }
  event void Sht11Temp.writeStatusRegDone( error_t result ) { }

  event void Sht11Hum.resetDone( error_t result ) { }
  event void Sht11Hum.measureTemperatureDone( error_t result, uint16_t val
  event void Sht11Hum.readStatusRegDone( error_t result, uint8_t val ) { }
  event void Sht11Hum.writeStatusRegDone( error_t result ) { }

  // We need default handlers as a client may wire to only the Temperature
  // sensor or only the Humidity sensor
  default event void Temperature.readDone( error_t result, uint16_t val ) {
  default event void Humidity.readDone( error_t result, uint16_t val ) { }
}


tos/platforms/telosa/chips/sht11/HalSensirionSht11C.nc

// HAL interface to Sensirion SHT11 temperature and humidity sensor
configuration HalSensirionSht11C {
  // The SHT11 HAL uses resource arbitration to allow the sensor to shared
  // between multiple clients and for automatic power management (the SHT11
  // is switched off when no clients are waiting to use it)
  provides interface Resource[ uint8_t client ];
  provides interface SensirionSht11[ uint8_t client ];
}
implementation {
  // The HAL implementation logic
  components new SensirionSht11LogicP();
  SensirionSht11 = SensirionSht11LogicP;

  // And it's wiring to the SHT11 HPL - the actual resource management is
  // provided at the HPL layer
  components HplSensirionSht11C;
  Resource = HplSensirionSht11C.Resource;
  SensirionSht11LogicP.DATA -> HplSensirionSht11C.DATA;
  SensirionSht11LogicP.CLOCK -> HplSensirionSht11C.SCK;
  SensirionSht11LogicP.InterruptDATA -> HplSensirionSht11C.InterruptDATA;

  components new TimerMilliC();
  SensirionSht11LogicP.Timer -> TimerMilliC;

  components LedsC;
  SensirionSht11LogicP.Leds -> LedsC;
}
```

```
tos/chips/sht11/SensirionSht11LogicP.nc

generic module SensirionSht11LogicP() {
  provides interface SensirionSht11[ uint8_t client ];

  uses interface GeneralIO as DATA;
  uses interface GeneralIO as CLOCK;
  uses interface GpioInterrupt as InterruptDATA;

  uses interface Timer<TMilli>;

  uses interface Leds;
}
implementation {

  ... bus protocol details omitted for brevity ...

}
```

```
tos/platforms/telosa/chips/sht11/HplSensirionSht11C.nc

// Low-level, platform-specific glue-code to access the SHT11 sensor found
// on telos-family motes - here  the HPL just provides resource management
// and access to the SHT11 data, clock and interrupt pins
configuration HplSensirionSht11C {
  provides interface Resource[ uint8_t id ];
  provides interface GeneralIO as DATA;
  provides interface GeneralIO as SCK;
  provides interface GpioInterrupt as InterruptDATA;
}
implementation {
  // Pins used to access the SHT11
  components HplMsp430GeneralIOC;

  components new Msp430GpioC() as DATAM;
  DATAM -> HplMsp430GeneralIOC.Port15;
  DATA = DATAM;
  components new Msp430GpioC() as SCKM;
  SCKM -> HplMsp430GeneralIOC.Port16;
  SCK = SCKM;

  components new Msp430GpioC() as PWRM;
  PWRM -> HplMsp430GeneralIOC.Port17;

  // HPL logic for switching the SHT11 on and off
  components HplSensirionSht11P;
  HplSensirionSht11P.PWR -> PWRM;
  HplSensirionSht11P.DATA -> DATAM;
  HplSensirionSht11P.SCK -> SCKM;

  components new TimerMilliC();
  HplSensirionSht11P.Timer -> TimerMilliC;

  components HplMsp430InterruptC;
  components new Msp430InterruptC() as InterruptDATAC;
  InterruptDATAC.HplInterrupt -> HplMsp430InterruptC.Port15;
  InterruptDATA = InterruptDATAC.Interrupt;

  // The arbiter and power manager for the SHT11
  components new FcfsArbiterC( "Sht11.Resource" ) as Arbiter;
  Resource = Arbiter;

  components new SplitControlPowerManagerC();
  SplitControlPowerManagerC.SplitControl -> HplSensirionSht11P;
  SplitControlPowerManagerC.ArbiterInit -> Arbiter.Init;
  SplitControlPowerManagerC.ArbiterInfo -> Arbiter.ArbiterInfo;
  SplitControlPowerManagerC.ResourceDefaultOwner -> Arbiter.ResourceDefault
}


tos/platforms/telosa/chips/sht11/HplSensirionSht11P.nc

// Switch the SHT11 on and off, and handle the 11ms warmup delay
module HplSensirionSht11P {
  // The SplitControl interface powers the SHT11 on or off (it's automatica
  // called by the SHT11 power manager, see HplSensirionSht11C)
  // We use a SplitControl interface as we need to wait 11ms for the sensor
  // warm up
  provides interface SplitControl;
  uses interface Timer<TMilli>;
  uses interface GeneralIO as PWR;
  uses interface GeneralIO as DATA;
  uses interface GeneralIO as SCK;
}
implementation {
  task void stopTask();
  command error_t SplitControl.start() {
    // Power SHT11 on and wait for 11ms
    call PWR.makeOutput();
    call PWR.set();
    call Timer.startOneShot( 11 );
    return SUCCESS;
  }

  event void Timer.fired() {
    signal SplitControl.startDone( SUCCESS );
  }

  command error_t SplitControl.stop() {
    // Power the SHT11 off
    call SCK.makeInput();
    call SCK.clr();
    call DATA.makeInput();
    call DATA.clr();
    call PWR.clr();
    post stopTask();
    return SUCCESS;
  }

  task void stopTask() {
    signal SplitControl.stopDone( SUCCESS );
  }
}
```

4. MDA100 Sensor Board Directory Organization

Here we show the organization of the sensor board directory for the mica-family Xbow MDA100CA and MDA100CB sensor boards, which have temperature and light sensors. It is found in `tos/sensorboards/mda100`:

```
./tos/sensorboards/mda100:
.sensor                                      # Compiler configuration
ArbitratedPhotoDeviceP.nc                    # Light sensor support compon
ArbitratedTempDeviceP.nc                     # Temperature sensor support
DemoSensorC.nc                               # Override TinyOS's default s
PhotoC.nc                                    # Light sensor HIL
PhotoImplP.nc                                # Light sensor support compon
PhotoTempConfigC.nc                          # Shared support component
PhotoTempConfigP.nc                          # Shared support component
SharedAnalogDeviceC.nc                       # Shared support component
SharedAnalogDeviceP.nc                       # Shared support component
TempC.nc                                     # Temperature Sensor HIL
ca/TempImplP.nc                              # Temperature sensor support
                                             # (MDA100CA board)
cb/TempImplP.nc                              # Temperature sensor support
```

```
                                             # (MDA100CB board)
  mda100.h                                   # Header file for mda100
```

This sensor board provides only a HIL (PhotoC and TempC components), and overrides the TinyOS demo sensor (DemoSensorC). The demo sensor is an alias for PhotoC.

The two forms of the mda100 differ only by the wiring of the temperature sensor. The user has to specify which form of the sensor board is in use by providing a `-I%T/sensorboards/mda100/ca` or `-I%T/sensorboards/mda100/cb` compiler option.

This sensor board relies on a platform-provided `MicaBusC` component that specifies how the mica-family sensor board bus is connected to the microcontroller.