Sensing

From TinyOS Wiki

This lesson introduces sensor data acquisition in TinyOS. It demonstrates two sensor applications: a simple application called Sense that periodically takes sensor readings and displays the values on the LEDs. And a more sophisticated application called Oscilloscope where nodes periodically broadcast their sensor readings to a basestation node. Using the Mote-PC serial communication described in the previous lesson the basestation forwards the sensor readings to the PC, where they are visualized with a dedicated graphical user interface.

Contents

Introduction

Sensing is an integral part of sensor network applications. In TinyOS 1.x sensing was syntactically connected with analog-to-digital converters (ADCs): TinyOS 1.x applications such as `Oscilloscope` or `Sense` used the `ADC` and `ADCControl` interfaces to collect sensor data. When new platforms appeared with sensors that were read out via the serial interface, not only did additional interfaces like `ADCError` have to be introduced, but it became clear that equating a sensor with an ADC was not always the appropriate thing to do.

Usually sensing involves two tasks: configuring a sensor (and/or the hardware module it is attached to, for example the ADC or SPI bus) and reading the sensor data. The first task is tricky, because a sensing application like, for example, `Sense` is meant to run on any TinyOS platform. How can `Sense` know the configuration details (like ADC input channel, the required reference voltage, etc.) of an attached sensor? It can't, because the configuration details of sensors will be different from platform to platform. Unless `Sense` knows about all sensors on all platforms it will be unable to perform the configuration task. However, the second task - reading the sensor data - can be solved so that the `Sense` application can collect sensor data even though it is agnostic to the platform it is running on.

In TinyOS 2.0 *platform independent* sensing applications such as `Oscilloscope`, `Sense` or `RadioSenseToLeds` do not use configuration interfaces like `ADCControl` anymore; instead they use the standard data acquisition interfaces `Read`, `ReadStream` or `ReadNow` for collecting sensor data. All configuration details are hidden from the application and this is why you can compile `Sense` and display sensor data on the *telosb* or the *micaz* platform, even though the actual sensors and their connection to the rest of the system may be completely different.

This raises questions like the following:

- Since the `Sense` application component only uses standard data acquisition interfaces who is in charge of defining which sensor it samples?

- If the Sense application component is not configuring the sensor, then who is responsible for that?
- How can an applications like Sense display sensor data when they do not know the details about sensor configuration? This includes questions like "what is the value range of the sensor data" or "is a temperature reading to be interpreted in degrees Celsius or Fahrenheit"?
- Let's assume there are several sensors on a platform: what steps have to be taken to let the Sense or Oscilloscope application display data from a different sensor?

After reading this tutorial you should be able to answer these questions. Using the Sense and Oscilloscope application as an example, the following sections explain how the data acquisition interfaces are used, how the configuration procedure works and, as an example, how Sense can be hooked up to sensors other than the default one on the *telosb* platform.

The Sense application

Sense is a simple sensing demo application. It periodically samples the default sensor and displays the bottom bits of the readings on the LEDs. The Sense application can be found in tinyos-2.x/apps/Sense. Let's first look at the SenseAppC.nc (http://www.tinyos.net/tinyos-2.x/apps/Sense/SenseAppC.nc) configuration:

```
configuration SenseAppC
{
}
implementation {
  components SenseC, MainC, LedsC, new TimerMilliC();
  components new DemoSensorC() as Sensor;

  SenseC.Boot -> MainC;
  SenseC.Leds -> LedsC;
  SenseC.Timer -> TimerMilliC;
  SenseC.Read -> Sensor;
}
```

The SenseAppC configuration looks similar to the BlinkAppC configuration described in lesson 1 (if you have not done so, read the sections on the Blink application in lesson 1). To understand the wiring let's look at the signature of the SenseC.nc (http://www.tinyos.net/tinyos-2.x/apps/Sense/SenseC.nc) module:

```
module SenseC
{
  uses {
    interface Boot;
    interface Leds;
    interface Timer<TMilli>;
    interface Read<uint16_t>;
  }
}
```

Like the BlinkC.nc module the SenseC.nc module uses the interfaces Boot, Leds and Timer<TMilli>. Additionally, it uses the Read<uint16_t> interface. The sequence of actions in the SenseC.nc implementation is as follows: SenseC.nc uses the Boot interface to start a periodic timer after the system has been initialized. Every time the timer expires SenseC.nc is signalled a timer event and reads data via the Read<uint16_t> interface. Reading data is a split-phase operation, it is divided in a command Read.read() and an event Read.readDone(). Thus every time the timer expires SenseC.nc calls Read.read() and waits for the Read.readDone() event. When data is signalled in the Read.readDone() event SenseC.nc displays it on the leds: the least significant bit is displayed on LED 0 (0 = off, 1 = on), the second least significant bit is displayed on LED 1 and so on.

The Read (http://www.tinyos.net/tinyos-2.x/tos/interfaces/Read.nc) interface (in tinyos-2.x/tos/interfaces) can be used to read a single piece of sensor data, let's look at it in detail:

```
interface Read<val_t> {
  /**
   * Initiates a read of the value.
   *
   * @return SUCCESS if a readDone() event will eventually come back.
   */
  command error_t read();

  /**
   * Signals the completion of the read().
   *
   * @param result SUCCESS if the read() was successful
   * @param val the value that has been read
   */
  event void readDone( error_t result, val_t val );
}
```

If you are not familiar with generic interfaces you will wonder what the meaning of `<val_t>` (in the first line) is and why the signature of `SenseC.nc` is using `Read<uint16_t>`. What you see above is a *generic interface definition*, because the `Read` interface takes a type parameter. Generic interfaces are explained in the nesC Language Reference Manual (version 1.2 and above). For now it is enough to know that generic interfaces have at least one type parameter and two components can be wired together only if they provide/use the interface with the same types (note that the `readDone` event passes a parameter of the `<val_t>` parameter, which is a placeholder for the actual data type). This means that since `SenseC.nc` is using the `uint16_t` variant of the `Read` interface, it can only be wired to a component that provides the `Read<uint16_t>` interface and thus `SenseC.nc` expects to read 16 bit unsigned integer sensor data. If you tried to wire `SenseC.nc` to a component that provides, for example, a `Read<uint8_t>` interface you would get an error from the nesC compiler.

Recall that the wiring is defined in the `SenseAppC.nc` configuration. Let's again take a look at which component `SenseC.nc` is wired to using the `Read<uint16_t>` interface in the `SenseAppC` configuration. The interesting lines are

```
  components new DemoSensorC() as Sensor;
```

and

```
  SenseC.Read -> Sensor;
```

This means that the *generic* `DemoSensorC` component provides the `Read<uint16_t>` interface to `SenseC.nc`

It is important to understand that the `SenseC.nc` module has no way of telling which sensor it is connected to; in fact it cannot even tell whether it is getting data from a sensor at all, because it can be wired to any component that provides a `Read<uint16_t>` interface. On a platform without any built-in sensors (like *micaz*) and no attached sensorboard the `DemoSensorC` component could simply return constant values. The last sentence hints that the `DemoSensorC` component is different for every platform: therefore you will not find `DemoSensorC.nc` in the TinyOS libraries. Instead, a different `DemoSensorC.nc` has to be written for every platform, i.e. the `DemoSensorC.nc` implementation for telosb will be different than the `DemoSensorC.nc` implementation for micaz. This is the answer to the first question asked in the introduction section: the *platform dependent* `DemoSensorC` component defines which sensor the `Sense` or `Oscilloscope` application is sampling and every platform that wants to run sensing applications such as `Oscilloscope`, `Sense` or `RadioSenseToLeds` has to provide its own version of `DemoSensorC`. Additionally, sensor boards may come with their own version of `DemoSensorC` (e.g., the `basicsb` sensorboard for the mica-family of motes define `DemoSensorC.nc` to be that board's light sensor).

The DemoSensorC component

Let's take a closer look at the DemoSensorC component. Every DemoSensorC component has the following signature:

```
generic configuration DemoSensorC()
{
  provides interface Read<uint16_t>;
}
```

In its implementation section, however, DemoSensorC may differ from platform to platform. For example, on the *telosb* platform DemoSensorC instantiates a component called VoltageC, which reads data from the MCU-internal voltage sensor. Because the *micaz* doesn't have any built-in sensors its DemoSensorC uses system library component like ConstantSensorC or SineSensorC, which return "fake" sensor data. Thus DemoSensorC is a means of indirecting sensor data acquisition from a platform-specific sensor component (like VoltageC) to platform-independent applications like Sense or Oscilloscope. Usually the configuration of a sensor is done in the component that DemoSensorC instantiates.

How can Sense be changed to sample a sensor other than the platform's default sensor? Usually this requires changing only a single line of code in DemoSensorC; for example, if you wanted to replace the VoltageC component on *telosb* by the constant sensor component ConstantSensorC you could change the following line in DemoSensorC from:

```
components new VoltageC() as DemoSensor;
```

to something like

```
components new ConstantSensorC(uint16_t, 0xbeef) as DemoSensor;
```

What sensors are available depends on the platform. Sensor components are usually located in the respective platform subdirectory (tinyos-2.x/tos/platforms), in the respective sensorboard subdirectory (tinyos-2.x/tos/sensorboards) or, in case of microprocessor-internal sensors, in the respective chips subdirectory (tinyos-2.x/tos/chips). ConstantSensorC and SineSensorC can be found in tinyos-2.x/tos/system.

Running the Sense application

To compile the Sense application, go to the apps/Sense directory and depending on which hardware you have, type something similar to make telosb install. If you get errors such as the following,

```
SenseAppC.nc:50: component DemoSensorC not found
SenseAppC.nc:50: component `DemoSensorC' is not generic
SenseAppC.nc:55: no match
```

your platform has not yet implemented the DemoSensorC component. For a quick solution you can copy DemoSensorC.nc from tinyos-2.x/tos/platforms/micaz to your platform directory (a good starting point on how to create sensor components is probably TEP 101 (http://www.tinyos.net/tinyos-2.x/doc/html/tep101.html) and TEP 114 (http://www.tinyos.net/tinyos-2.x/doc/html/tep114.html)).

If you have a mica-family mote and a "basic" (mda100) sensor board, you can get a more interesting test by compiling with

```
SENSORBOARD=basicsb make platform install
```

to run Sense using the mda100's light sensor.

Once you have installed the application the three least significant bits of the sensor readings are displayed on the node's LEDs (0 = off, 1 = on). It is the least significant bits, because `Sense` cannot know the precision (value range) of the returned sensor readings and, for example, the three most significant bits in a `uint16_t` sensor reading sampled through a 12-bit ADC would be meaningless (unless the value was left-shifted). If your `DemoSensorC` represents a sensor whose readings are fluctuating you may see the LEDs toggle, otherwise `Sense` is not very impressive. Let's take a look at a more interesting application: `Oscilloscope`.

The Oscilloscope application

`Oscilloscope` is an application that let's you visualize sensor readings on the PC. Every node that has `Oscilloscope` installed periodically samples the default sensor via (`DemoSensorC`) and broadcasts a message with 10 accumulated readings over the radio. A node running the `BaseStation` application will forward these messages to the PC using the serial communication. To run `Oscilloscope` you therefore need at least two nodes: one node attached to your PC running the `BaseStation` application (`BaseStation` can be found at `tinyos-2.x/apps/BaseStation` and was introduced in the previous lesson) and one or more nodes running the `Oscilloscope` application.

Let's take a look at the `OscilloscopeAppC.nc (http://www.tinyos.net/tinyos-2.x/apps/Oscilloscope/OscilloscopeAppC.nc)` configuration:

```
configuration OscilloscopeAppC
{
}
implementation
{
  components OscilloscopeC, MainC, ActiveMessageC, LedsC,
    new TimerMilliC(), new DemoSensorC() as Sensor,
    new AMSenderC(AM_OSCILLOSCOPE), new AMReceiverC(AM_OSCILLOSCOPE);

  OscilloscopeC.Boot -> MainC;
  OscilloscopeC.RadioControl -> ActiveMessageC;
  OscilloscopeC.AMSend -> AMSenderC;
  OscilloscopeC.Receive -> AMReceiverC;
  OscilloscopeC.Timer -> TimerMilliC;
  OscilloscopeC.Read -> Sensor;
  OscilloscopeC.Leds -> LedsC;
}
```

The actual implementation of the application is in `OscilloscopeC.nc (http://www.tinyos.net/tinyos-2.x/apps/Oscilloscope/OscilloscopeC.nc)` . This is the signature of `OscilloscopeC.nc`:

```
module OscilloscopeC
{
  uses {
    interface Boot;
    interface SplitControl as RadioControl;
    interface AMSend;
    interface Receive;
    interface Timer;
    interface Read;
    interface Leds;
  }
}
```

`Oscilloscope` is a combination of different building blocks introduced in previous parts of the tutorial. Like `Sense`, `Oscilloscope` uses `DemoSensorC` and a timer to periodically sample the default sensor of a platform. When it has gathered 10 sensor readings `OscilloscopeC` puts them into a message and broadcasts that message via the `AMSend` interface. `OscilloscopeC` uses the `Receive` interface for synchronization purposes (see below) and the `SplitControl` interface, to switch the radio on. If you want to know more about mote-mote radio communication read lesson 3.

Running the Oscilloscope application

To install the `Oscilloscope` application go to `tinyos-2.x/apps/Oscilloscope` and depending on which hardware you have, type something similar to `make telosb install,1`. Note the ",1" after the `install` option, which assigns ID 1 to the node. Assigning IDs to nodes is helpful to differentiate them later on in the GUI, so make sure you assign different IDs to all nodes on which `Oscilloscope` is installed (e.g. install `Oscilloscope` on a second node with `make telosb install,2` and so on). A node running `Oscilloscope` will toggle its second LED for every message it has sent and it will toggle its third LED when it has received an `Oscilloscope` message from another node: incoming messages are used for sequence number synchronization to let nodes catch up when they are switched on later than the others; they are also used for changing the sample rate that defines how often sensor values are read. In case of a problem with the radio connection the first LED will toggle.

Install `BaseStation` on another node and connect it to your PC. As usual, on the `BaseStation` node you should see the second LED toggle for every message bridged from radio to serial.

Running the Java GUI

To visualize the sensor readings on your PC first go to `tinyos-2.x/apps/Oscilloscope/java` and type `make`. This creates/compiles the necessary message classes and the `Oscilloscope` Java GUI. Now start a SerialForwarder and make sure it connects to the node on which you have installed the `BaseStation` application (how this is done is explained in the previous lesson). In case you have problems with the Java compilation or the serial connection work through the previous lesson.

Once you have a SerialForwarder running you can start the GUI by typing `./run` (in `tinyos-2.x/apps/Oscilloscope/java`). You should see a window similar to the one below:

Each node is represented by a line of different color (you can change the color by clicking on it in the mote table). The x-axis is the packet counter number and the y-axis is the sensor reading. To change the sample rate edit the number in the "sample rate" input box. When you press enter, a message containing the new rate is created and broadcast via the `BaseStation` node to all nodes in the network. You can clear all received data on the graphical display by clicking on the "clear data" button.

The `Oscilloscope` (or `Sense`) application displays the raw data as signalled by the `Read.readDone()` event. How the values are to be interpreted is out of scope of the application, but the GUI let's you adapt the visible portion of the y-axis to a plausible range (at the bottom right).

Related Documentation

- nesC reference manual (http://nescc.sourceforge.net/papers/nesc-ref.pdf)
- TEP 101: ADC (http://www.tinyos.net/tinyos-2.x/doc/html/tep101.html)
- TEP 114: SIDs: Source and Sink Independent Drivers (http://www.tinyos.net/tinyos-2.x/doc/html/tep114.html)
- TinyOS Programming Guide (http://csl.stanford.edu/~pal/pubs/tinyos-programming-1-0.pdf)

< **Previous Lesson** | **Top** | **Next Lesson** >

- This page was last modified on 4 April 2011, at 01:07.
- This page has been accessed 100,494 times.