Assignment #2 - TinyOS Programming
Sensor Networks - v2004 - 336

Bo Gotthardt-Petersen <bogp@diku.dk>,
Morten Bjerre <mortman@diku.dk>,
and Jonas Fonseca <fonseca@diku.dk>

Datalogisk Institut, Københavns Universitet

11th December 2004

Contents

1 Introduction

This report documents the analysis and implementation of a simple TinyOS program capable of sampling light and temperature. A brief analysis explains the design choices made and how the calibration process works.

2 Analysis

2.1 State-graph

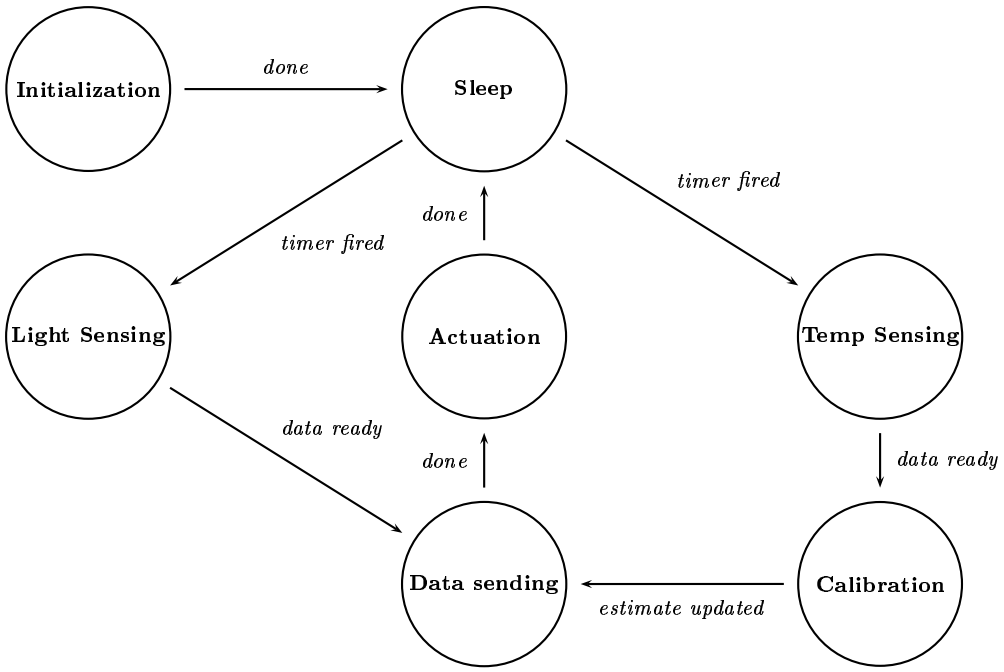

Figure 1: The state graph.

2.2 Calibration

Both sensors output a 16-bit value which needs to be converted before it can be used. The light value needs to be converted into a light is ON/OFF and the temperature value needs to be converted into degrees celsius.

To perform this conversion calibration needs to be performed. In the case of the light sensor calibration would consist of finding a threshold which decides when the light goes from ON to OFF or vice-versa. The best way to find this threshold would be to record the sensor reading in full darkness. The conversion would then consist of comparing the sensor output with the threshold. If the output is less or equal the light is reported as OFF and

if the output is greater then the light is reported as ON. Another possibility would be to assume the sensor is linearly proportional to the actual light intensity and calibrate it like the light sensor. We will use the threshold method.

Calibrating the temperature sensor is more difficult since a range of values is desired. Assuming the output from the sensor is linearly proportional to the actual temperature, calibration would consist of recording two output values for different temperatures. Under the initial assumption those two readings can then be used to convert any output value to the actual temperature. If the ltemperature sensor is not linearly proportional then several readings (5-20) at different temperatures need to be performed and a function needs to be fit to the results. That function could then be used to convert output to degrees Celsius. We assume linear proportionality and use the first method.

2.3 Formatting the output

The assignment requires that 5 light samples and 1 temperature sample are performed and output on the UART every second. Ignoring formatting, the output would then have a bandwidth of 12*16 bits = 192 bps. Adding simple formatting e.g. a whitespace and a letter would bring the bandwidth up to 192 + (6*2*8) = 288 bps. which exceeds the maximum bandwidth. As such careful use of formatting and data encoding is necessary to ensure enough bandwidth for both output and possible debug information.

A simple approach would be to output every sensor reading at once and output a newline every second. This would produce all 6 readings in a row on each line. Example: "123 123 123 125 333 124\n" (192 bps). Distinguishing the different readings and debugging would be difficult though, also there might be problems if multiple readings are output at once.

A better approach would be to output each reading at once and include a letter indicating the type of reading and output a newline every second. Since this would exceed the bandwidth as per the first example, the output should consist of the converted values ie. light as 0 (OFF) or 1 (ON) and temperature in the 0-99 degrees celsius range. Example: "L1 L1 L1 L1 T18 L1\n" (152 bps). This makes it easy to distinguish the readings and debugging is easier, though several outputs at once might still be a problem.

Another approach would be to put all the readings in a buffer. Once the buffer is full (could be every second) the contents are formatted and output. This would solve the problem of trying to output several readings at once.

We assume the assignment requires the readings to be output at once and therefore we use the second approach.

2.4 Timers

The assignment requires samples to be taken every 200ms and every second it therefore seems logical to use two different timers. That would ensure that all the required readings are done and with the correct intervals. The problem is that this might lead to concurrency problems since both timers might wish to access the same resources (the UART e.g.).

A simpler approach would be to use one timer for both combined with one or more counters e.g. a 1ms timer and counters counting to 200 and 1000 respectively, thereby ensuring that the requirements are fulfilled. This would avoid most concurrency problems since only one timer can access resources.

We decided to use the second approach to keep our implementation simple and problem free.

2.5 Avoiding Concurrency Problems

Avoiding concurrency problems: Buffers and atomic operations (ensuring correct number of readings per second)

2.6 Debugging

Debugging the sensorboard is quite difficult since its only basic output are the leds. To ensure easy debugging we used an iterative approach starting with the "TestUart" program and modifying it to our design one small step at a time.

3 Calibration

3.1 Light Sensor

To calibrate the Light Sensor successive readings were taken while the light level was reduced. True darkness turns out to give a sensor reading around 1 as expected, but reaching that level of darness is impractical.

Therefore we have decided to set the darkness threshold at 100, since that is easy to reach under test conditions.

3.2 Temperature Sensor

To calibrate the temperature sensor 6 readings at different temperatures were taken:

Temperature ($°C$)	7	12	17	19	22	25
Sensor reading	126	176	194	197	210	250

A 3-degree polynomial was then fitted to the readings using MatLab. The polynomial was then used to generate a table of values for sensor readings ranging from 25 to 789. Conversion is done by comparing the two nearest entries in the table and interpolating the values.

We would have preferred to use a function rather than a table and linear interpolation, but as far as we could discern there is no logarithmic function in the version of TinyOS we are using. Interpolating the values will give errors since the underlying curve is not linear but we believe the error is small and thus an acceptable compromise with keeping the table small.

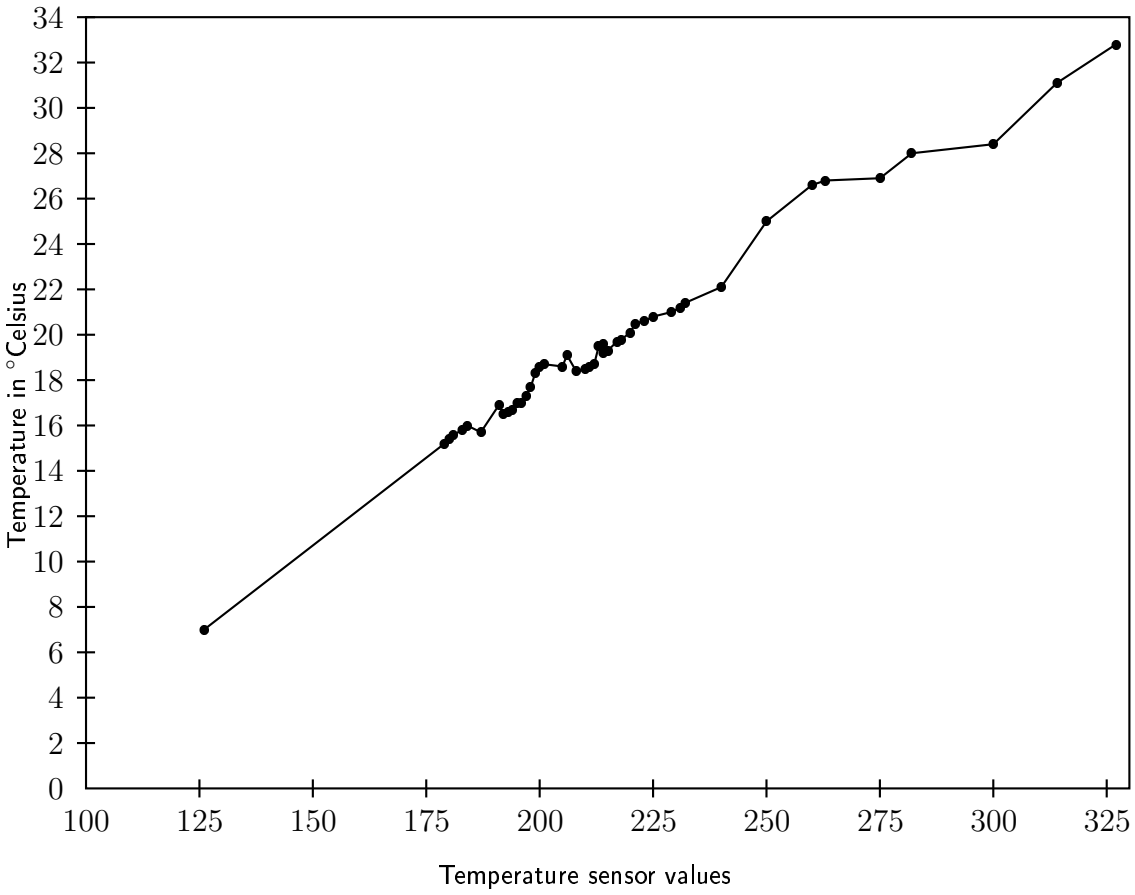

Figure 2: A plot of pairs of sensor values and their corresponding values in °C.

4 Conclusion

We have succesfully completed the assignment. Though our implementation is simple it fulfills all the requirements and avoids the major pitfalls.

We have encountered a few problems mainly the lack of mathematical functions such as logarithms and floating point operations. Our work arounds are not pretty but we believe the errors associated are reasonably small. Calibrating the temperature sensor proved quite difficult since it was hard to create a wide range of different temperature settings (without breaking the sensorboard!). Also it was hard to measure the precise temperature with the poor quality thermometer used.

References

[1] Freescale Semiconductors Danmark A/S.
ZigBee Evaluation Kit (DIG528-2): Main Schematic, Rev. R02.02.

Freescale Semiconductors Danmark A/S, Monday, July 19, 2004.

[2] Freescale Semiconductors, Inc.
MC13192 Evaluation Board, Reference Manual, Rev. 0.0.
Motorola Inc., August 2004.

[3] Freescale Semiconductors, Inc.
MC13192 Product Preview, Rev. 2.4.
Freescale Semiconductors, Inc., July 2004.

[4] Freescale Semiconductors, Inc.
MC13192 Reference Manual, 2.4 GHz Low Power Transceiver for 802.15.4, Rev 1.0.
Motorola Inc., May 2004.

[5] Freescale Semiconductors, Inc.
MC9S08GB/GT Data Sheet, Rev. 2.2.
Freescale Semiconductors, Inc., September 2004.

[6] Maxim Integrated Products.
MAX3316E, MAX3317E, MAX3318E, MAX3319E - ś15kV ESD-Protected, 2.5V, 1ţA, 460kbps, RS-232-Compatible Transceivers.
Maxim Integrated Products,
URL: http://www.maxim-ic.com/quick_view2.cfm/qv_pk/2167, Sun, 21 November 2004.

A Component Graph

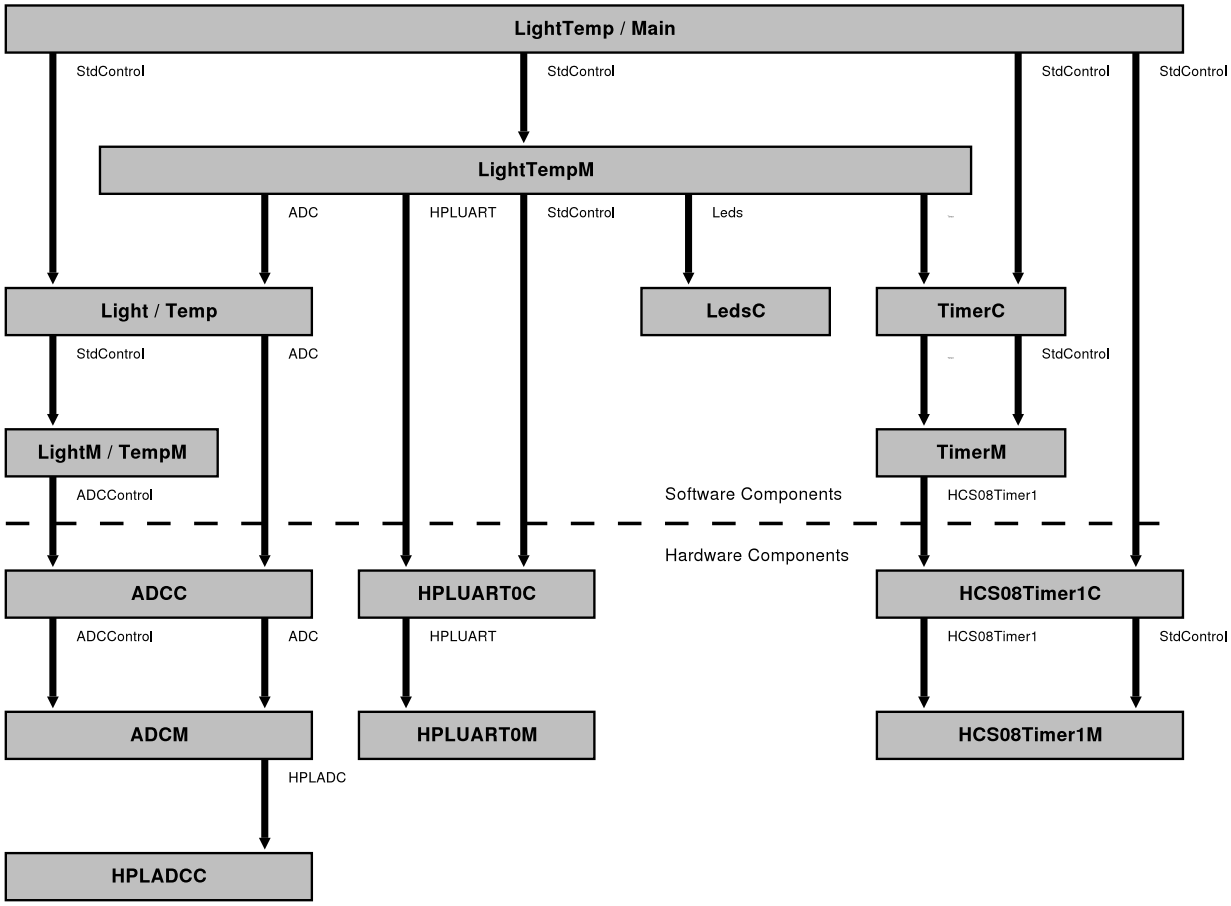

Figure 3: The graph of components and the wired interfaces.

B Source Code

B.1 LightTemp.nc

```
/* $Id: LightTemp.nc,v 1.20 2004/12/11 11:23:47 di040660 Exp $ */

/** Light / temperature sensing and actuating application, that logs to the UART.
 *
 * <p>The application will sample both light and temperature but at different
 * rates. All sensor readings are logged to the UART in the format</p>
 *
 * <pre>L 12     L111     L145     L553     L652     T 22</pre>
 *
 * <p>Where an L-prefix means the following value is from the light sensor
 * and a T</b>-prefix denotes a temperature reading. Note that due to the
 * limited range of number that can be represented with 3 chars values greater
 * than 1000 has a lowercase prefix (e.g. 'l' instead of 'L').</p>
```

```
 *
 * <p>Apart from sensing and logging the application also actuates when
 * the light is off and the temperature is greater than 22 degrees C. The
 * actuation is signaled by turning on LED1 (aka. the ``red'' LED).</p>
 *
 * <p>The LEDS and what they mean:</p>
 * <ul>
 * <li> LED1 (red)    Is on when the light and temperature threshold is reached.
 * <li> LED2 (green)  Is on if some error was detected.
 * <li> LED3 (yellow) Is toggled on completion of each UART output.
 * </ul>
 *
 * <p>Resetting will print a short identification of the authors.</p>
 *
 * @author Bo Gotthardt-Petersen, <bogp@diku.dk>
 * @author Morten Bjerre, <mortman@diku.dk>
 * @author Jonas Fonseca, <fonseca@diku.dk>
 */

configuration LightTemp {
}

implementation {
        /* Include the needed components */
        components Main, LightTempM, Light, Temp, TimerC, LedsC, HPLUART0C;

        /* Wire the interfaces that LightTempM needs onto the components */
        LightTempM.Leds      -> LedsC;
        LightTempM.Uart      -> HPLUART0C;
        LightTempM.Light     -> Light.Light;
        LightTempM.Temp      -> Temp.Temp;
        LightTempM.LightTimer -> TimerC.Timer[unique("Timer")];
        LightTempM.TempTimer  -> TimerC.Timer[unique("Timer")];

        /* Wire up the StdControl interfaces to Main */
        Main.StdControl -> LightTempM.StdControl;
        Main.StdControl -> Light.StdControl;
        Main.StdControl -> Temp.StdControl;
        Main.StdControl -> TimerC.StdControl;
}
```

B.2 LightTempM.nc

```
/* $Id: LightTempM.nc,v 1.44 2004/12/11 11:23:47 di040660 Exp $ */

/** The main module used by the LightTemp application.
 *
 * @author Bo Gotthardt-Petersen, <bogp@diku.dk>
 * @author Morten Bjerre, <mortman@diku.dk>
 * @author Jonas Fonseca, <fonseca@diku.dk>
 */

/* Define to make the sensorboard calibrate temperature readings. */
//#define CONFIG_CALIBRATE_TEMPERATURE

/* The timer intervals used to sample the light and temperature. */
#define TIMER_INTERVAL_LIGHT     200
#define TIMER_INTERVAL_TEMP     1000

/* The buffer slot size allows: <id char> <3 value chars> <separater char> */
/* XXX: The UART can only write 30 or less chars per second so with 6 readings
 * per second keep the slot size <= 5. */
#define BUFFER_SLOT_SIZE        5
#define BUFFER_SLOTS            6
```

```
#define BUFFER_SIZE             (BUFFER_SLOTS * BUFFER_SLOT_SIZE)

/* The thresholds controlling when the ``red'' LED (LED1) is on. */
#define REDLED_THRESHOLD_LIGHT    100     /* Easily triggered darkness */
#define REDLED_THRESHOLD_TEMP      22     /* XXX: This is for calibrated values */

#define checkRedLedThresholds(light, temp) \
        ((light) <= REDLED_THRESHOLD_LIGHT && (temp) > REDLED_THRESHOLD_TEMP)

/* The sensor IDs used internally and to prefix the sensor readings. */
#define SENSOR_ID_TEMP          'T'
#define SENSOR_ID_LIGHT         'L'

/* Max 27 chars. There has to be room for two new lines and a NUL char. */
#define APPLICATION_ID          "Group 3: Bo, Morten & Jonas"

module LightTempM {
        provides {
                interface StdControl;
        }

        uses {
                interface Leds;
                interface HPLUART as Uart;
                interface ADC     as Light;
                interface ADC     as Temp;
                interface Timer   as LightTimer;
                interface Timer   as TempTimer;
        }
}

implementation {
        /* ************************************************************************
         * Component variables:
         * ************************************************************************/

        /** The most recently read value from the light sensor.
         * XXX: Requres atomic access. */
        uint16_t lightData;

        /** The most recently read value from the temperature sensor.
         * XXX: Requres atomic access. */
        uint16_t tempData;

        /** The previously value of the temperature sensor reading. Used for
         * checking if the a new calibration is required. */
        uint16_t prevTempData;

        /** The latest calibrated temperature value.
         * By having separate variables for read sensor values and calibrated
         * values it is ensured that access is synchronous and condition
         * checking will always see a consistent value. */
        uint16_t calibratedTempData;

        /** The two UART buffers. */
        uint8_t buffer[2][BUFFER_SIZE] = { "", "\n" APPLICATION_ID "\n" };

        /* XXX: Initialize the buffer state variables so the ID string will be
         * printed when the first sensor reading is recorded. */

        /** The buffer that sensor readings is actively logged to. */
        uint8_t *outputBuffer   = buffer[1];

        /** The buffer that the UART is currently outputting. */
        uint8_t *writeBuffer    = buffer[0];
```

```
        /** The next slot to log sensor readings to. */
        uint8_t nextSlot         = BUFFER_SLOTS;

        /** The state indicating if the UART is still using the write buffer.
         * XXX: Requres atomic access. */
        bool writeBufferIsInUse = FALSE;


        /* ************************************************************************
         * StdControl:
         * ************************************************************************/

        /** Init.
         *
         * <p>Inits the components that does not support StdControl, and
         * blink a toggle a couple of LEDs.</p>
         */
        command result_t StdControl.init()
        {
                call Leds.init();
                call Uart.init();

                return SUCCESS;
        }

        /** Start
         *
         * <p>Start the timers.</p>
         */
        command result_t StdControl.start()
        {
                return rcombine(
                        call LightTimer.start(TIMER_REPEAT, TIMER_INTERVAL_LIGHT),
                        call TempTimer.start(TIMER_REPEAT, TIMER_INTERVAL_TEMP)
                );
        }

        /** Stop - never called.
         *
         * <p>Stop the timers.</p>
         */
        command result_t StdControl.stop()
        {
                return rcombine(
                        call LightTimer.stop(),
                        call TempTimer.stop()
                );
        }


        /* ************************************************************************
         * UART buffering:
         * ************************************************************************/

        /** Flush the output buffer to the UART
         *
         * <p>Swaps the output buffer with the write buffer and flushes it to
         * the UART if the write buffer is not being used.</p>
         *
         * <p>Toggles the green LED if the UART is busy or the output buffer has
         * to be overwritten. Both are considered an error because sensor
         * readings are discarded.</p>
         *
         * <p>Toggle the yellow LED to signal progress.</p>
```

```
 */
void flushOutputBuffer()
{
        result_t result;
        uint8_t *tmp;
        bool localWriteBufferIsInUse;

        atomic localWriteBufferIsInUse = writeBufferIsInUse;

        /* Signal error if data needs to be overwritten. */
        /* FIXME: Possible improvement includes making it fine-grained
         * so only the oldest buffer slot is actually overwritten. It
         * would require a memmove(). */
        if (localWriteBufferIsInUse == TRUE) {
                call Leds.greenToggle();
                return;
        }

        tmp         = writeBuffer;
        writeBuffer = outputBuffer;

        result = call Uart.put(writeBuffer, &writeBuffer[BUFFER_SIZE]);
        if (result == SUCCESS) {
                atomic writeBufferIsInUse = TRUE;
                call Leds.yellowToggle();
        } else {
                call Leds.greenToggle();
        }
}

/** Allocate a buffer slot
 *
 * <p>Allocates a new buffer slot.</p>
 *
 * @return      A slot in the current output buffer.
 */
uint8_t *getBufferSlot()
{
        if (nextSlot == BUFFER_SLOTS) {
                flushOutputBuffer();

                /* Reset the ouput buffer */
                memset(outputBuffer, ' ', BUFFER_SIZE);
                nextSlot = 0;
        }

        return &outputBuffer[BUFFER_SLOT_SIZE * nextSlot++];
}

/** Record sensor ID and data to the UART output buffer
 *
 * <p>Encodes the data from internal representation to decimal
 * representation. Prefixes the encoded sensor data with the ID of the
 * sensor. If the data can not be represented (i.e. data >= 1000) the ID
 * is lowercased.</p>
 *
 * @param id     The sensor ID that will be prefixed the data value.
 * @param data   The data to encode for the UART.
 */
void putSensorData(uint8_t id, uint16_t data)
{
        uint8_t *string = getBufferSlot();
        uint8_t i = BUFFER_SLOT_SIZE - 1;

        string[i--] = id == SENSOR_ID_LIGHT ? '\t' : '\n';
```

```
                /* Lowercase the ID to increase the span of values */
                if (data >= 1000)
                        id += 32;

                do {
                        uint8_t digit = data % 10;

                        string[i--] = digit + '0';

                        data /= 10;

                } while (data && i >= 1);

                string[0] = id;
        }


        /* ************************************************************************
         * Temperature callibration:
         * ************************************************************************/

#ifdef CONFIG_CALIBRATE_TEMPERATURE
        struct calibration {
                uint16_t actualValue;
                uint16_t sensorValue;
        };

#define foreachCalibration(current, prev, table) \
        for (prev = current = table; \
             current < &table[sizeof(table) / sizeof(struct calibration)]; \
             prev = current++)

        /** The Temperature Calibration Table
         *
         * <p>Based on: xş * 0.0619 - xš * 2.9999 + x * 50.2921 - 100.7167</p>
         */
        const struct calibration tempCalibrationTable[] = {
                {   0,     0 },
                {   3,    25 },
                {   7,   126 },
                {  10,   164 },
                {  13,   182 },
                {  16,   190 },
                {  19,   196 },
                {  20,   200 },
                {  21,   206 },
                {  22,   213 },
                {  23,   222 },
                {  24,   234 },
                {  25,   249 },
                {  28,   314 },
                {  31,   419 },
                {  34,   574 },
                {  37,   789 },
                { 255, 65535 },
        };

        /** Temperature calibration
         *
         * <p>Calibrates an approximation to the actual temperature value using
         * a lookup table of sensor <-> calibrated value pairs.</p>
         *
         * @param data  The value to calibrate.
         * @return      An approximation to an actual temperature value.
```

```
         */
        uint16_t getCalibratedTemperature(uint16_t data)
        {
                const struct calibration *current, *prev;

                foreachCalibration (current, prev, tempCalibrationTable) {
                        uint16_t sensorDelta, actualDelta, dataDelta;

                        if (current->sensorValue == data)
                                return current->actualValue;

                        if (current->sensorValue < data)
                                continue;

                        sensorDelta = current->sensorValue - prev->sensorValue;
                        dataDelta   = data                 - prev->sensorValue;
                        actualDelta = current->actualValue - prev->actualValue;

                        /* Scale down the delta of the actual sensor value. */
                        actualDelta = dataDelta * actualDelta / sensorDelta;

                        return prev->actualValue + actualDelta;
                }

                /* XXX: The data value can never be outside the range of the
                 * calibration table due to 65535u being the upper limit.
                 * So this is mostly to remove the warninog. */
                return prev->actualValue;
        }
#endif /* CONFIG_CALIBRATE_TEMPERATUR* */

        /** The temperature calibration task
         *
         * <p>Reads the saved sample of temperature data and calibrates the data
         * value into degrees celsius.</p>
         *
         * <p>Optionally caches the value in order to reduce the number of
         * useless calibrations when the temperature is stable.</p>
         */
        task void calibrateTemperature()
        {
                uint16_t localTempData;

                atomic localTempData = tempData;

#ifdef CONFIG_CALIBRATE_TEMPERATURE
                if (prevTempData == localTempData)
                        return;

                prevTempData = localTempData;
                calibratedTempData = getCalibratedTemperature(localTempData);
#else
                calibratedTempData = localTempData;
#endif
        }


        /* ********************************************************************
         * Light / Temp sensing and actuaion
         * ********************************************************************/

        /** The light outputter task
         *
         * <p>Reads the saved sample of light data and uses putData() as backend
         * to output the value to the UART.
```

```
 */
task void putLightData()
{
        uint16_t localLightData;

        atomic localLightData = lightData;
        putSensorData(SENSOR_ID_LIGHT, localLightData);
}

/** The temperature outputter task
 *
 * <p>Reads the saved sample of temperature data and uses putData() as
 * backend to output the value to the UART.</p>
 */
task void putTempData()
{
        putSensorData(SENSOR_ID_TEMP, calibratedTempData);
}

/** The condition checking task
 *
 * <p>Reads the saved light and temperature sensor samples and updates
 * the ``red'' LED depending how they relate to the configured
 * thresholds.</p>
 */
task void checkConditions()
{
        uint16_t localLightData;

        atomic {
                localLightData = lightData;
        }

        if (checkRedLedThresholds(localLightData, calibratedTempData)) {
                call Leds.redOn();
        } else {
                call Leds.redOff();
        }
}

/** Receive data from the light sensor
 *
 * <p>Save the data and post a task to output the value to the UART and
 * a task that for threshold conditions.</p>
 *
 * @return      SUCCESS always.
 */
async event result_t Light.dataReady(uint16_t data)
{
        atomic lightData = data;
        post putLightData();
        post checkConditions();

        return SUCCESS;
}

/** Receive data from the temperature sensor
 *
 * <p>Save the data and post a task to output the value to the UART and
 * a task that for threshold conditions.</p>
 *
 * <p>Optionally calibrate the data value into degrees celsius.</p>
 *
 * @return      SUCCESS always.
 */
```

```
async event result_t Temp.dataReady(uint16_t data)
{
        atomic tempData = data;
        post calibrateTemperature();
        post putTempData();
        post checkConditions();

        return SUCCESS;
}


/* ************************************************************************
 * Timer related code
 * ************************************************************************/

/** LightTimer fired.
 *
 * <p>Request a light sensor reading each time the timer fires.</p>
 *
 * @return       The return code of the request.
 */
event result_t LightTimer.fired()
{
        return call Light.getData();
}

/** TempTimer fired.
 *
 * <p>Request a temperature sensor reading each time the timer
 * fires.</p>
 *
 * @return       The return code of the request.
 */
event result_t TempTimer.fired()
{
        return call Temp.getData();
}


/* ************************************************************************
 * Handle stuff from the UART
 * ************************************************************************/

/** Completion of UART outputting.
 *
 * <p>Mark that the UART is ready for more output.</p>
 *
 * @return       SUCCESS always.
 */
async event result_t Uart.putDone()
{
        atomic writeBufferIsInUse = FALSE;
        return SUCCESS;
}

/** Receiving data on the UART.
 *
 * <p>Happily ignore the data.</p>
 *
 * @return       SUCCESS always.
 */
async event result_t Uart.get(uint8_t data)
{
        return SUCCESS;
}
```

}

C Calibration values

126	7.0
179	15.2
180	15.4
181	15.6
183	15.8
184	16.0
187	15.7
191	16.9
192	16.5
193	16.6
194	16.7
195	17.0
196	17.0
197	17.3
198	17.7
199	18.3
200	18.6
201	18.7
205	18.6
206	19.1
208	18.4
210	18.5
211	18.6
212	18.7
213	19.5
214	19.2
214	19.6
215	19.3
217	19.7
218	19.8
220	20.1
221	20.5
223	20.6
225	20.8
229	21.0
231	21.2
232	21.4
240	22.1
250	25.0
260	26.6
263	26.8
275	26.9
282	28.0
300	28.4
314	31.1
327	32.8