Getting Started With 1-Wire Bus Devices

by

Vishesh Pamadi and Bradford G. Nickerson

TR15-235, August 25, 2015

Faculty of Computer Science
University of New Brunswick
Fredericton, N.B. E3B 5A3
Canada

Phone: (506) 453-4566
Fax: (506) 453-3566
E-mail: fcs@unb.ca
http://www.cs.unb.ca

Contents

List of Figures

Abstract

This document explores the functioning of wireless sensor networks and bus protocols supported by various sensor motes. An investigation into the 1-wire protocol and its interface with supported devices was conducted. The 1-wire devices which run on parasitic mode can be configured by using only a data and a ground line. These devices are known to consume very low power and deliver real-time data efficiently at relatively high speeds. Various 1-wire bus masters were shortlisted and analyzed in order to control the 1-wire devices in a 1-wire network. We also implemented a 1-wire network containing three 1-wire temperature sensors on the WEL (Web Energy Logger). The sensor data and its fluctuations over time were recorded and analyzed on a web page. The 1-wire temperature sensor (DS18B20) was explored and the resulting temperature was obtained and displayed on a serial terminal interface using the Arduino Duemilanove. The Duemilanove runs on an Atmega328 processor belonging to the Atmel Family, and is a high performance 8-bit microcontroller.

1 Introduction to Wireless Sensor Networks

A wireless sensor network (WSN) contains spatially distributed autonomous sensors to monitor physical or environmental conditions, such as temperature, sound, pressure and humidity. The monitored data is cooperatively passed through the network to a gateway or sink node. The gateway is also referred to as the BaseStation where the data is collected, sorted on a logical basis and analyzed. Modern wireless sensor networks (WSNs) are bi-directional, also enabling control of sensor activity.

A WSN is built of "nodes" numbering from a few to several hundreds or even thousands, where each node is connected to one (or sometimes several) sensors. Each such sensor network node typically has several parts: a radio transceiver with an internal antenna or connection to an external antenna, a microcontroller, an electronic circuit for interfacing with the sensors and an energy source, usually a battery or an embedded form of energy harvesting. A sensor node might vary in size from that of a shoebox down to the size of a grain of dust, although functioning motes of genuine microscopic dimensions have yet to be created. The cost of sensor nodes is similarly variable, ranging from tens to hundreds of dollars, depending on the complexity of the individual sensor nodes. Size and cost constraints on sensor nodes result in corresponding constraints on resources such as energy, memory, computational speed and communication bandwidth. The topology of WSNs can vary from a simple star network to an advanced multi-hop wireless mesh network. The propagation technique between the hops of the network can be routing or flooding.

The main characteristics of Wireless Sensor Networks are :

- Power consumption constraints for nodes using batteries or energy harvesting
- Scalability to large scale deployment

- Heterogeneity of nodes

Major applications of Wireless Sensor Networks include monitoring in the fields of process management, earth sensing and research. Some of its applications include environmental monitoring (e.g. air quality, water, waste, radiation, animal nesting), infrastructure monitoring (e.g. roads, buildings, parking), and real-time operations (e.g. industrial control, animal tracking, robot navigation).

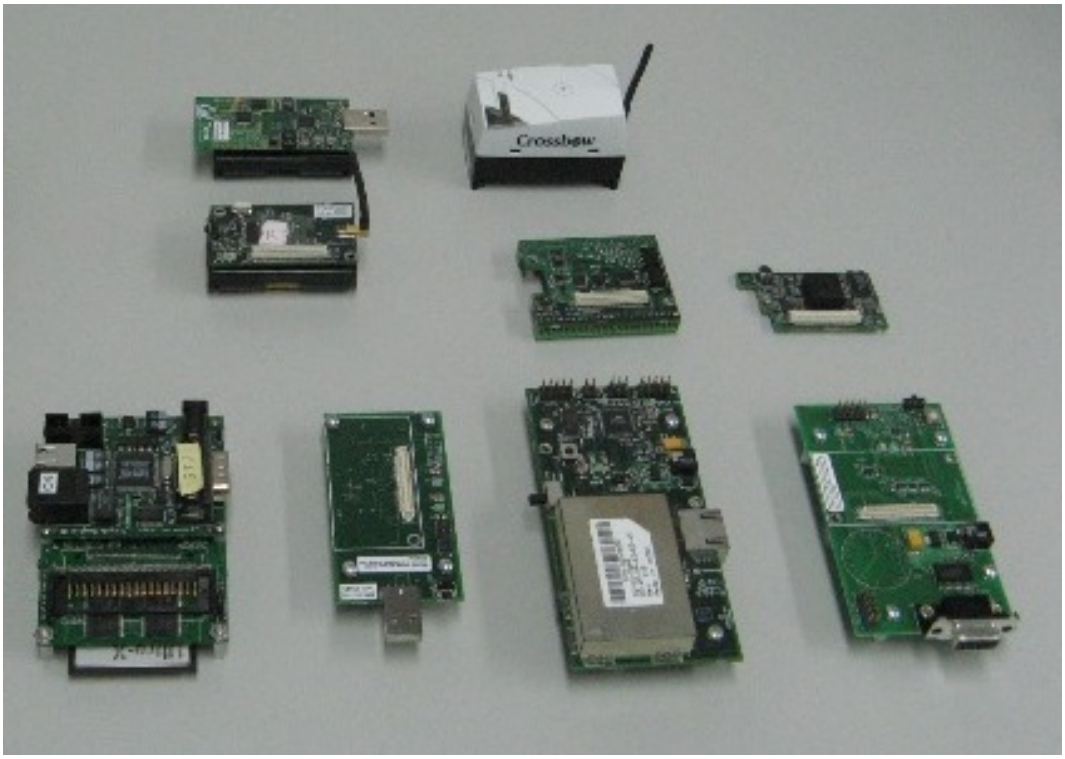

Figure 1: Group of MICA motes belonging to the Crossbow family (from [**?**]).

1.1 List of Sensor Nodes

As mentioned earlier, a WSN consists of various motes configured in a particular network. These motes come in various types and are capable of performing some processing, gathering sensory information and communicating with other connected nodes in the network. Based on the application and its requirements, the sensor nodes with supported protocols are chosen accordingly.

Table 1: A comparison of several sensor nodes.

Name	**MCU**	**Interfaces**	**Expansion Connector**
TinyNode 584	TI MSP430	SPI, UART, I2C (meant for internal sensors), ADC	20 GPIO Pins
Telos A/B	TI MSP430	SPI (used for internal sensors), I2C, ADC, UART, USB debugging	6 GPIO Pins
TI CC2650 SensorTag	ARM Cortex-M3	I2C, ADC, UART, SPI	20 pin SAMTEC
Zolertia Z1	MSP430	SPI, I2C, ADC, UART	52 GPIO pins
UC Mote Mini	MSP430	SPI, I2C	N/A
UC Mote Proton Base	Atmega 128	USB interface only (Acts as gateway node only)	N/A
Shimmer 3	MSP430	I2C, UART, USB debugging, ADC, SPI	8 GPIO pins
OpenMote-CC2538	TI CC2538	I2C, UART, USB, ADC, Ethernet	20 pin XBEE

1.2 TinyOS

TinyOS is a free and open source software component-based operating system and platform targeting wireless sensor networks (WSNs). TinyOS is an embedded operating system written in the nesC programming language as a set of cooperating tasks and processes. Other WSN operating systems include Contiki, LiteOS and OpenWSN. TinyOS applications are written in nesC, a dialect of the C language optimized for the memory limits of sensor networks. Its supplementary tools are mainly in the form of Java and shell script front-ends. Associated libraries and tools, such as the nesC compiler and Atmel AVR binutils toolchains, are mostly written in C.

TinyOS programs are built out of software components, some of which present hardware abstractions. Components are connected to each other using interfaces. TinyOS provides interfaces and components for common abstractions such as packet communication, routing, sensing, actuation and storage.TinyOS is completely non-blocking; it has one stack. Therefore, all I/O operations that last longer than a few hundred microseconds are asynchronous and have a callback. To enable the native compiler to better optimize across call boundaries, TinyOS uses nesC's features to link these callbacks, called events, statically. While being non-blocking enables TinyOS to maintain high concurrency with one stack, it forces programmers to write complex logic by stitching together many small event handlers. To support larger computations, TinyOS provides tasks, which are similar to a deferred procedure call and interrupt handler bottom halves. A TinyOS component can post a task, which the OS will schedule to run later. Tasks are non-preemptive and run in FIFO order.

This simple concurrency model is typically sufficient for I/O centric applications, but has difficulty with CPU-heavy applications. This led to the development of a thread library for TinyOS, named TOSThreads.

TinyOS code is statically linked with program code and is compiled into a small binary, using a custom GNU toolchain. Associated utilities are provided to complete a development platform for working with TinyOS.

nesC is a component-based, event-driven programming language used to build applications for the TinyOS platform. We ran code written in nesC using Eclipse Software with a Yeti 2 plugin used for depicting syntax errors.

2 Getting started with the Tinynode

Tinynodes with standard extension boards provide a very powerful tool for sensor networks. We plan to introduce the 1-wire network interface to the Tinynode that can provide real time data from various sensors.

To get things started, first connect the serial port from the extension board to an RS232 to USB adapter, and connect the adapter to a Linux computer running the Ubuntu Operating System; we used version 14.04. Place two AA batteries into a AA battery pack, and connect to the battery connector of the extension board. Figure **??** shows a Tinynode with standard extension board hooked to a battery pack and to a computer. You could also use a 5 Volt DC adapter to power up the board. For voltage and other information see [**?**] and [**?**].

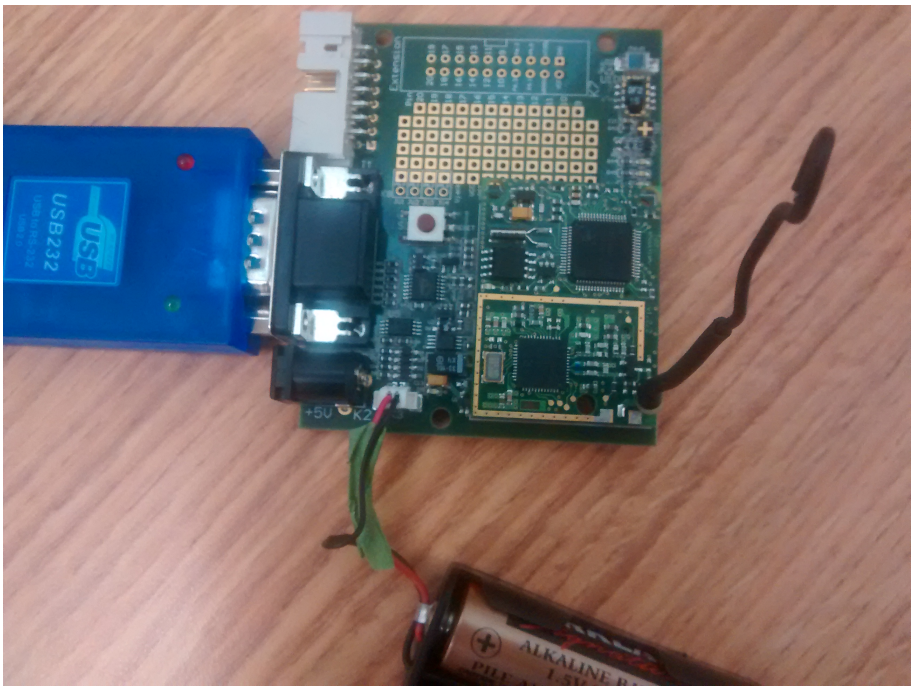

Figure 2: Tinynode 584 with standard extension board with a battery pack and serial connection to the computer.

To obtain the current USB port assigned to the mote, run the command

```
> motelist
```

Output should look like the following:

```
Reference  Device           Description
---------- ---------------- ----------------------------------------------
A900geE8   /dev/ttyUSB0     FTDI FT232R USB UART
```

2.1 Testing the TinyNode using the Blink Application

We programmed the TinyNode motes with Eclipse using the TinyOS environment on a Ubuntu 14.04 Operating System. For more information on installing Ubuntu and TinyOS on your system, please refer to [**?**].

The Blink Application is a basic application and is generally used for testing your board. To run the Blink application, go to `tinyos-2.x\apps\Blink` and run the command:

```
> make tinynode
```

This command compiles the application for the TinyNode platform. Then, assuming the device address obtained from the command `motelist` is `/dev/ttyUSB0`, run the command:

```
> tos-bsl --invert-reset -c /dev/ttyUSB0 -r -e -I -p build/tinynode/main.ihex
```

This command invokes the BootStrapLoader software for the microcontroller (MSP430F1xx) of the TinyNode. This flashes the program on your Tinynode. You should see two LED's blinking once the program is flashed.

2.2 TestSerial Application

The TestSerial application sends and receives packets through the serial port. This application comes with the TinyOS distribution and is used to test the communication between a computer and a mote.

To run the test serial application, go to `tinyos-2.x\apps\tests\TestSerial` and run the command:

```
> make tinynode
```

This command compiles the application for the TinyNode platform. Then, assuming the device address obtained from the command `motelist` is `/dev/ttyUSB0`, run the command:

```
> tos-bsl --invert-reset -c /dev/ttyUSB0 -r -e -I -p build/tinynode/main.ihex
```

This command invokes the BootStrapLoader software for the microcontroller (MSP430F1xx) of the TinyNode. For more detailed information run the command:

```
> man tos-bsl
```

After running tos-bsl, the TestSerial application is installed in the Tinynode. Under the same directory, run the Java TestSerial application with the following command:

```
> java TestSerial
```

One of the LEDs will blink, and the output looks like the following:

```
Sending packet 0
Received packet sequence number 28
Sending packet 1
Received packet sequence number 29
Sending packet 2
Received packet sequence number 30
Sending packet 3
```

These messages show that the communication between the development computer and the Tinynode is happening correctly.

2.3 Oscilloscope Application

The Oscilloscope application is used to extract sensor data from the motes. Two Tinynodes are required. Tinynode 1 is the gateway node that receives radio packets from Tinynode 2, and sends the packets to the computer through the serial connection available on the mote. Tinynode 2 generates data using the built-in sensors, and transmits these packets to the gateway (Tinynode 1) through the radio.

To run the Oscilloscope program, connect Tinynode 2 to the computer, and under the `tinyos-2.x/apps/Oscilloscope` directory, run the following commands:

```
> make tinynode
> tos-bsl --invert-reset -c /dev/ttyUSB0 -r -e -I -p build/tinynode/main.ihex
```

This will compile and install the Oscilloscope application on Tinynode 2. Remove Tinynode 2 from the serial connection. The green LED blinks each time Tinynode 2 sends a packet through the radio.

Now you will need the BaseStation application for Tinynode 1. The BaseStation application acts as a bridge between the serial and radio channels. Connect Tinynode 1 to the computer and under the `tinyos-2.x/apps/BaseStation` directory run:

```
> make tinynode
> tos-bsl --invert-reset -c /dev/ttyUSB0 -r -e -I -p build/tinynode/main.ihex
```

This will compile and install the BaseStation application on Tinynode 1. Leave the mote connected to the computer. The green LED will blink each time the Tinynode 1 receives a packet on the radio from Tinynode 2 and transmits the packet through the serial connection to the computer.

To read messages on the computer, you can run the SerialForwarder application and the Oscilloscope.java program. You can also read the messages with a terminal emulator like Minicom. If Minicom is well configured, once you start Minicom you will see incoming characters that represent the data that is being received by Tinynode 1 from Tinynode 2. The readings shown correspond to the default sensor which is normally the VoltageC component. An example screen shot of the running oscilloscope application is shown in Figure **??**.

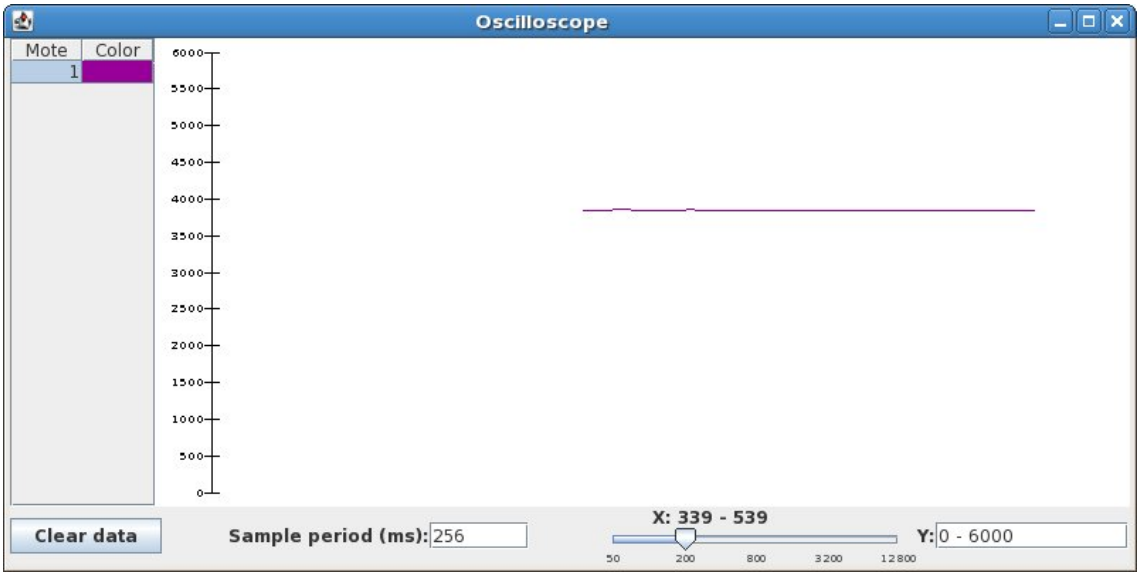

Figure 3: The ADC voltage values plotted using the Oscilloscope.java program.

3 The 1-wire Protocol

The 1-wire protocol is a device communication bus system designed by Dallas Semiconductor Corp. that provides low-speed data, signaling, and power over a single wired channel. One distinctive feature of the bus is the possibility of using only two wires: data and ground. To accomplish this, 1-wire devices include an 800 pF capacitor to store charge, and to power some devices during periods when the data line is active.

1-wire devices may be one of many components on a circuit board within a product, a single component within a device such as a temperature probe, or may be attached to a device being monitored. Some laboratory systems and other data acquisition and control systems connect to 1-wire devices using cables with modular connectors or with CAT-5 cable, with the devices themselves mounted in a socket, incorporated in a small PCB, or attached to the object being monitored. In our experiments, we use a 3-wire twisted, single strand cable as the 1-wire bus as seen in Figure **??**.

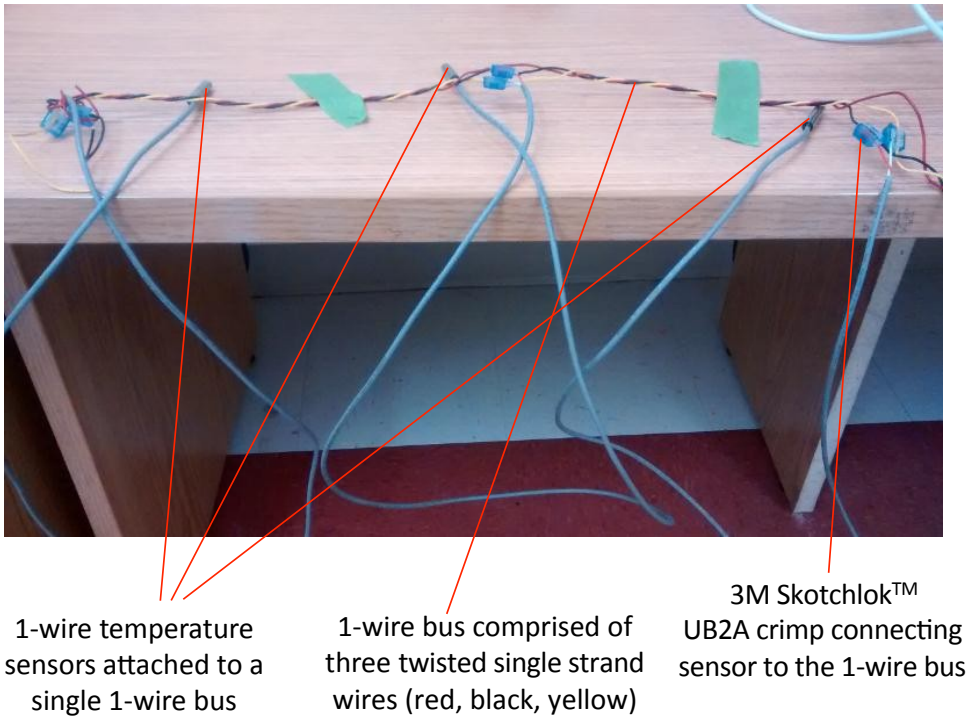

Figure 4: Three 1-wire temperature sensors connected to a 3-wire twisted cable (adapted from [**?**]).

Only the data and ground wires are required for communication with low power devices (e.g. temperature sensor DS18B20) that can run only on parasitic power. The 1-wire network was initially implemented using the WEL (Web Energy Logger) which is described in the next section. 1-wire products provide a combination of memory, mixed signal, and secure authentication functions via a single contact serial interface. With both power and communication delivered over the serial 1-wire protocol, 1-wire devices have the ability to provide key functions in systems where connections must be minimized. A list of approximately 25 different 1-wire devices is given in [**?**].

Devices that have no separate pin for power supply and extract energy from the 1-wire signal bus are called parasitic devices. These devices require only data and ground lines for their interface. All 1-wire devices include a unique 64-bit ID (factory-programmed) which serves as the device address on the 1-wire bus. Figure **??** depicts the architecture of the 1-wire bus protocol.

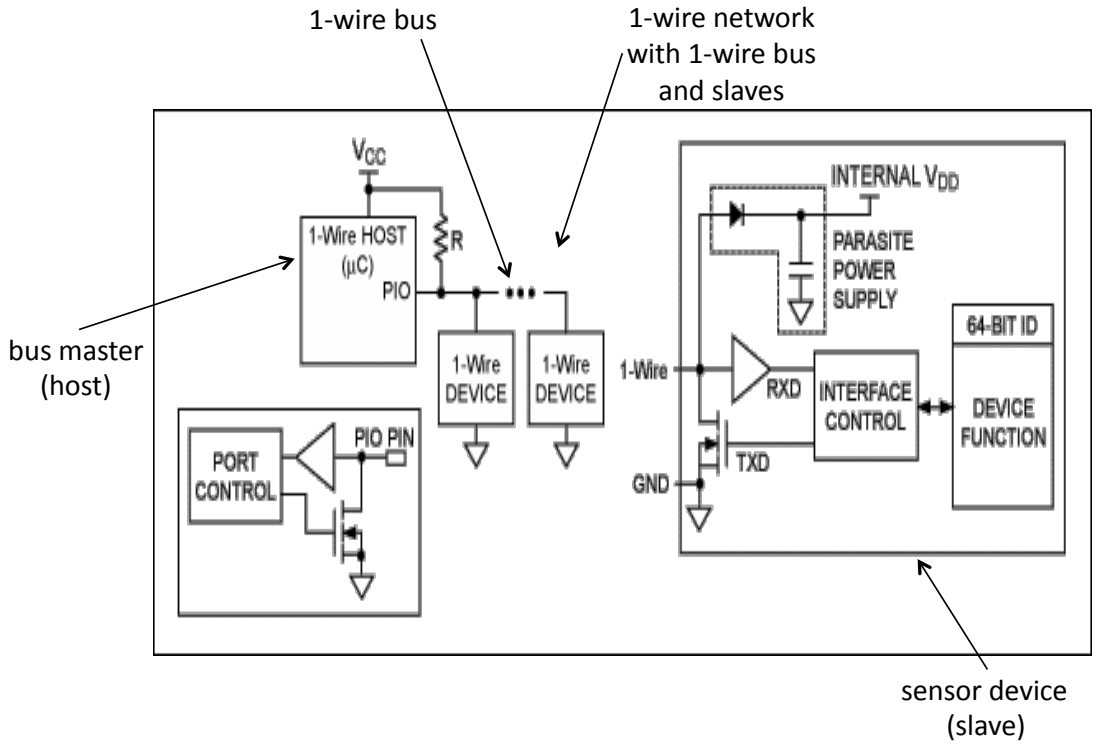

Figure 5: Working of the 1-wire protocol (from [**?**]).

3.1 What Is Special About 1-wire?

The 1-wire bus is a voltage-based digital system that can work with two contacts, data and ground, for half-duplex bidirectional communication. In contrast to other serial communication systems such as I2C or SPI, 1-wire devices are designed for use in a local area network environment with low data rates (e.g. 16.3 Kbps or 160 Kbps in overdrive mode). Either disconnecting from the 1-wire bus or a loss of contact puts the 1-wire slaves into a defined reset state. When the voltage returns, the slaves wake up and signal their presence. With only one contact to protect, the built-in Electrostatic Discharge (ESD) protection of 1-wire devices is extremely high. With two contacts, 1-wire devices are an economical way to add electronic functionality to non-electronic objects for identification, authentication, and delivery of calibration data or manufacturing information.

3.2 Using the Tinynode with a 1-wire network

As seen in Figure **??**, a microcontroller (e.g. TinyNode [**?**] with TI MSP430) can act as the host in a 1-wire network. The 1-wire data bus is connected using a pull up resistor with Vcc to any of the GPIO pins on the host.

A 1-wire master initiates and controls the communication with one or more 1-wire slave devices on the 1-wire bus. In our experiments, we planned to use bus masters DS2480B+ (serial to 1-wire line driver) and DS2482-100+ (I2C to 1-wire line driver) to control 1-wire slaves on the 1-wire network. The TinyNode will connect to the bus master which is in turn connected to a number of 1-wire slaves.

4 Working with the Web Energy Logger (WEL)

The Web Energy Logger (WEL) is designed to monitor and log the energy characteristics of a building. The basic WEL unit can read a large number of networked sensors (e.g. temperature and contact closures), 6 pulse-output devices (watt-meter or flow meter), 8 local contact closures and two 0-10V analog inputs. Filtered data is presented on a series of web pages as well as posted to the WELServer.com website via a standard 10-baseT Ethernet connection wired from the WEL server to the internet via a local port or router connection.

WELServer.com combines the live data with graphic images to generate "system snapshots" that can be displayed on any user's website. Live data is also stored in monthly log files and used to generate trend graphs. Logs can be downloaded by users and imported into data processing packages like Excel.

The WEL server supports the 1-wire network, and offers instant web display of 1-wire capable sensor devices. It can communicate very accurately and access real time data at relatively high speeds. Using the 1-wire bus, we connected three 1-wire temperature sensors (DS18B20) to the WEL server. The WEL uses the Ethernet connection to log the data onto welserver.com which can be accessed by a user using the allotted username and password.

4.1 Hardware Configuration

Figure **??** shows a picture of a WEL device in operation.

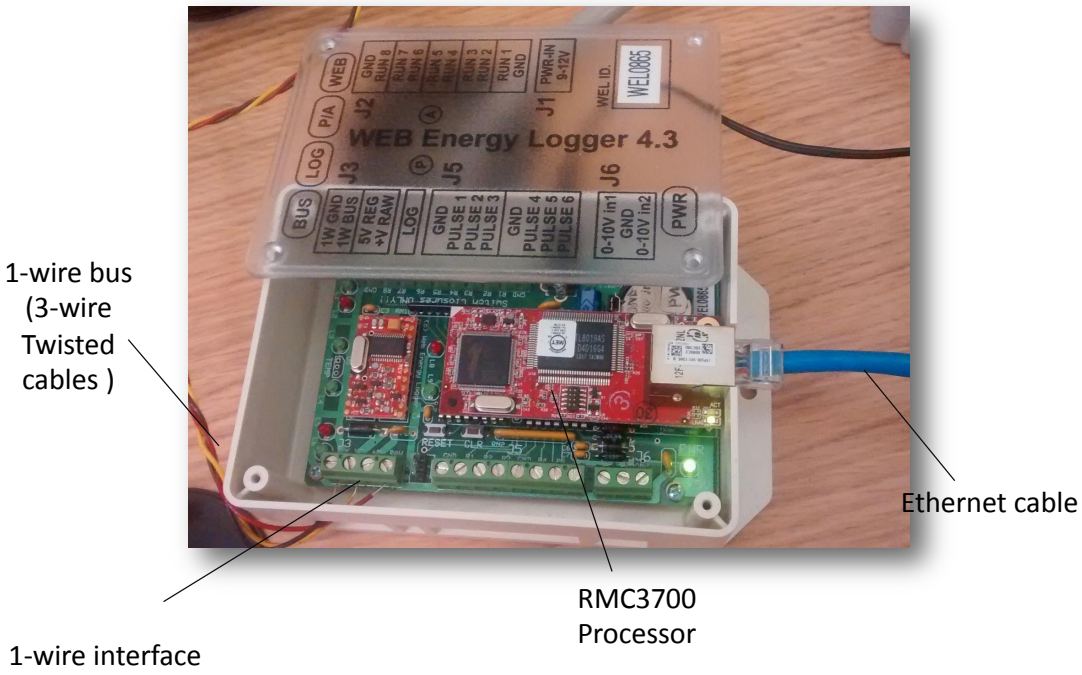

Figure 6: A WEL(Web Energy Logger) version 4.3 connected to the internet and to a 1-wire bus).

The WEL comprises several hardware elements. The WEL uses a simple analog voltage-regulator to generate the required +5V. Unregulated raw DC voltage is applied to the board though a 2.5mm barrel plug (center positive). The raw input voltage must be rectified DC, and should be in the 9V to 12V range.

The WEL provides two different voltages to the 1W bus to power custom circuits. These voltages come from two points on the WEL's own power supply. The raw DC supply comes from the input side of the 5V regulator, and the +5V is from the output side. The +5V voltage can be sent along with the 1-wire bus at the 1-wire interface. Each supply line has an inline resettable fuse to limit excessive current that might prevent the WEL from running.

The WEL runs on the RMC3700 Rabbit Core processor running at a 55 MHz maximum clock speed, and contains FLASH, an Ethernet interface and 1 MB RAM. A robust 1-wire interface called the iButton is used to run the 1-wire network. This interface communicates with the 1-wire network in parasitic mode, writes and reads directly to the device and

enables data logging. Its functionality includes expandable memory and a real-time clock. The device can be used to connect to external devices using analog and serial inputs. Our experiments were limited to the 1-wire interface.

4.2 Setting up the WEL server

Before you can proceed with configuring the hardware, you must have an online account at welserver.com. This account gives you access to all the data obtained from the devices attached to the WEL. One can also see live data, create graphs and log data into files. This account mainly helps you to keep track of WEL devices. A step by step procedure on configuring your WEL on welserver.com is available at [**?**].

The first step is to provide Ethernet connectivity to the WEL. This can be done by connecting the WEL to an ethernet hub or to a port using the same LAN as your computer. Using a web browser connected to port 5150 on the URL of the WEL (see Figure **??**), the user clicks on **Setup overview** option to view his account, then clicks on the option called **Local WEL Admin**. You will be directed to a page that looks like the page in Figure **??**.

Figure 7: Setting up a user account with 1-wire sensors on the WEL via a web browser.

A step by step procedure is provided in the support manual available at [**?**] to configure your device with the WEL and access real time data when the devices are initially connected to the WEL, names are given and calibrations are made. If your device is recognized and configured well, the one wire status would display no errors. After setting up your 1-wire devices, you can set your live data and create graphs to record data.

4.3 Our Experiment with the WEL Server

We connected three one wire temperature sensors(DSB1820) along the 1-wire bus to the WEL. All the hardware components were ordered from `http://welserver.com/store.htm`. A 7 m long, twisted 3-wire, single strand wire was used as the 1-wire bus for this experiment. Three temperature sensors provided in the kit were crimped along the bus using the UB2A connectors and crimping tool. This is to ensure a tight connection between the sensors and bus. The instructions for wiring the sensors to the bus can be found at the page containing all support files [**?**].

The 1-wire bus was plugged into the WEL server. The setup overview for the WEL was configured and the WEL was connected to the internet. An experimental setup was created in the lab, where we placed three temperature sensors running up a wall. One was placed on the ground, another on a ledge near a glass block window about 2.5 m off the floor, and another across the wall. A graph (see Figure **??**) is automatically created containing plotted data sampled, in this case, at 3 minute intervals.

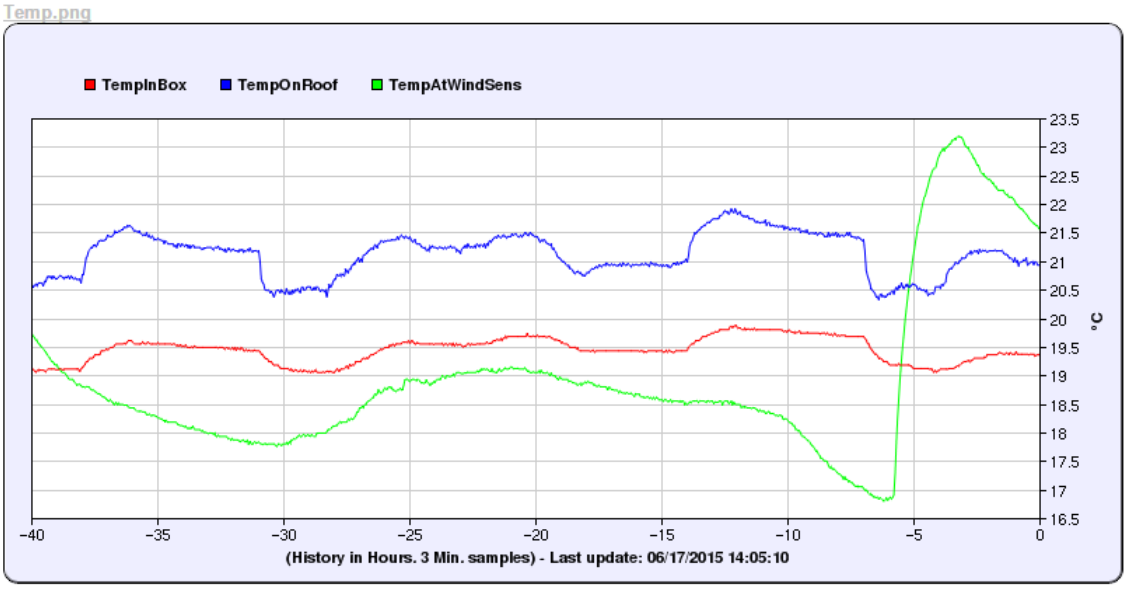

Figure 8: Graphical representation of temperature processed by the WEL, as shown on the web page.

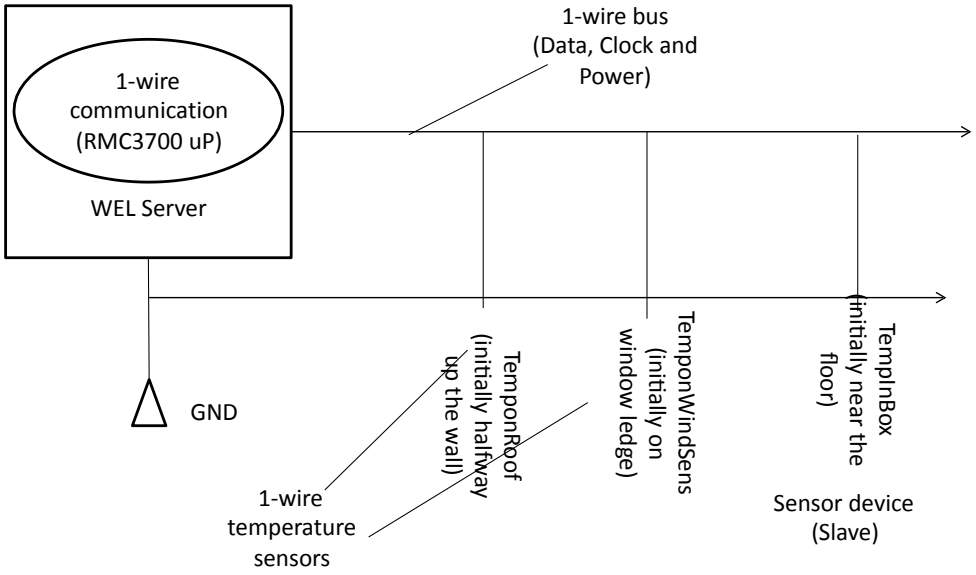

Figure 9: Schematic depiction of WEL operation in the ITB214 lab.

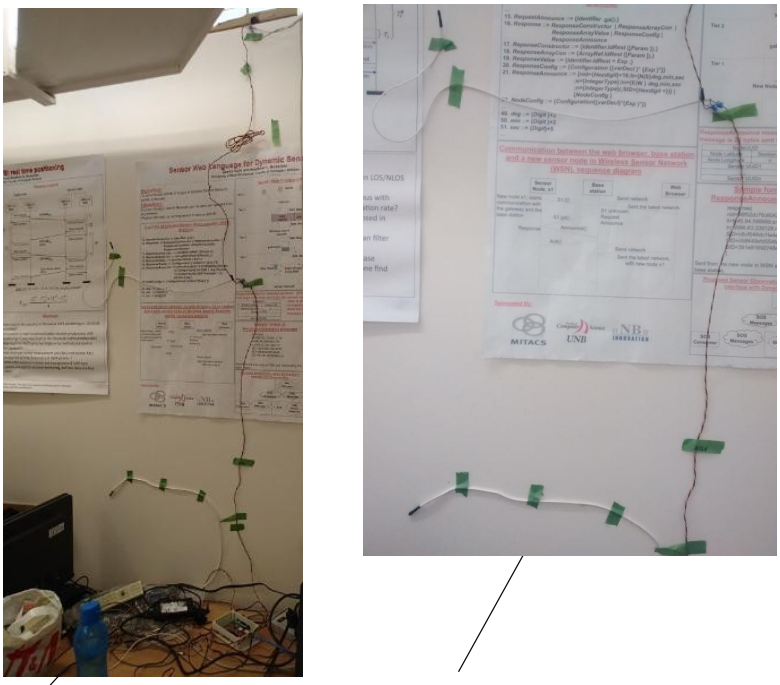

Figure 10: Three temperature sensors connected to a 1-wire bus on the wall in ITB-214.

The graph in Figure **??** depicts :

`TemponRoof represents the temperature sensor placed across the middle of the wall`

`TempAtWindSens represents the sensor placed on the window ledge, beside the glass block`

`TempoinBox represents the sensor placed across the wall, nearest to the floor`

The fluctuations of temperature from the sensor placed near the glass block can be observed changing from 17 to 23 degrees in the span of a few hours. The sensor placed across the wall in the middle mostly fluctuate around room temperature. The sensor placed closest to the floor shows the least temperature. Hence, a distribution of temperature among the sensors can be observed, especially for the sensor on the window ledge which is affected much more by the outdoor air temperature.

5 Interfacing the DS18B20 with the Arduino

We interfaced an Arduino with the DS18B20 using the OneWire and DallasTemperature library. An Arduino Duemilanove, a single DS18B20 temperature sensor, a USB cable for

debugging and the Arduino software was used for the experimental setup. The data line of the temperature sensor is connected to digital pin number 2 on the Arduino board using a pull up resistor of value 4.7 KOhm. Figure **??** depicts the hardware circuit interfacing the DS18B20.

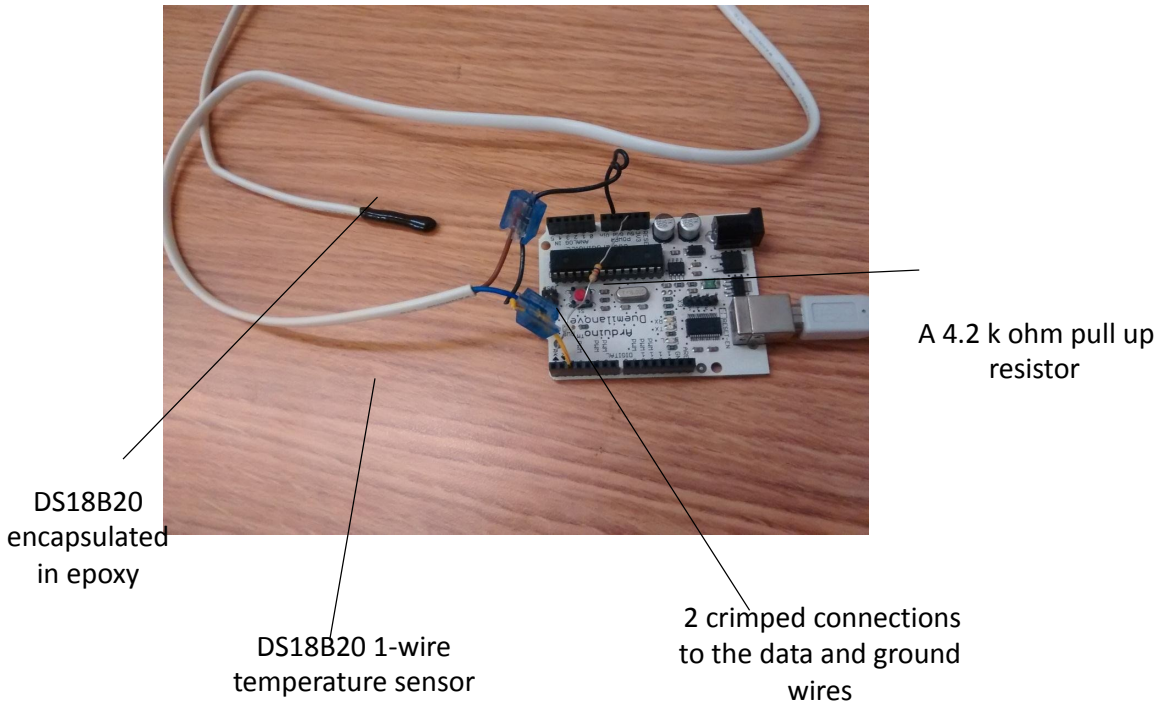

Figure 11: DS18B20 temperature sensor interfaced with an Arduino

On setting the baud rate to 9600 bps, the program is flashed onto the Arduino board. The serial port is opened, and the temperature sampled at every second is updated and seen on the serial port. The code for the above result can be found in Appendix B. Figure **??** displays the serial output terminal of the Arduino.

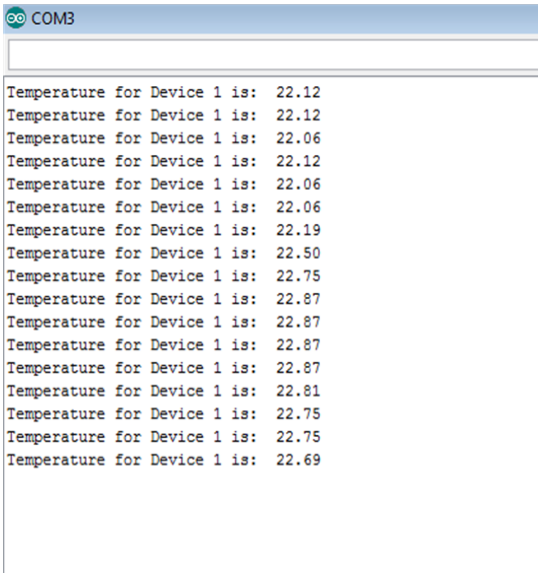

Figure 12: Serial port of the Arduino displaying the temperature received from DS18B20

6 Conclusion

In this report, we have experimented with the 1-wire bus using two platforms, the Web Energy Logger (WEL) and the Arduino connected to a PC. We have built a small (3 temperature sensors) 1-wire network, and interfaced it to the WEL. In turn, we configured the WEL, and connected it to welserver.com for public display of the sensed temperatures. Also, we studied one of the most popular 1-wire device, the DS182B0 (1-wire temperature sensor) and interfaced it with the WEL as well as the Arduino.

References

[1] Our Cool House. Wel wiring tips-support file. http://welserver.com/support_files/WEL_Wiring_Quick_Tips.pdf.

[2] Our Cool House. Web energy logger server users manual. Internet, March 2011. http://welserver.com/support_files/WEL_User_Manual_4.0.3.pdf.

[3] Helmut Kopka and Patrick W. Daly. *Guide to Latex*. Addison Wesley, fourth edition, February 2004.

[4] Maxim/Dallas Semiconductors. List of 1-wire devices and their datasheets. http://www.maximintegrated.com/en/pl_list.cfm/filter/21.

[5] Shockfish. Standard extension board users manual. Internet, January 2011. http://www.tinynode.com/?q=system/files/TNSEB_Users_Manual_v_1_3.pdf.

[6] Shockfish. Tinynode 584 users manual. Internet, January 2011. http://www.tinynode.com/?q=system/files/TN584_Users_Manual_v_1_3.pdf.

[7] Drexel Tutorials. Beginner's guide to crossbow motes. Internet. http://www.pages.drexel.edu/~kws23/tutorials/motes/motes.html.

[8] Maxim Tutorials. Overview of 1-wire technology and its use. Internet. http://www.maximintegrated.com/en/app-notes/index.mvp/id/1796.

[9] TinyOS Wiki. Installing tinyos on ubuntu 12.04 and above. Internet. http://tinyos.stanford.edu/tinyos-wiki/index.php/Installing_TinyOS.

How this Document was Created

LaTeX [**?**], version TeXShop Version 2.47 (2.47) with TeXShop Version 2.47 (2.47) on Mac OS X
Figures using MS PowerPoint version 14.5.8

APPENDIX A

A Oscilloscope Application README File

```
1  README for Oscilloscope
2  Author/Contact: tinyos-help@millennium.berkeley.edu
3
4  Description:
5
6  Oscilloscope is a simple data-collection demo. It periodically samples
7  the default sensor and broadcasts a message over the radio every 10
8  readings. These readings can be received by a BaseStation mote and
9  displayed by the Java "Oscilloscope" application found in the java
10 subdirectory. The sampling rate starts at 4Hz, but can be changed from
11 the Java application.
12
13 You can compile Oscilloscope with a sensor board's default sensor by compiling
14 as follows:
15   SENSORBOARD=<sensorboard name> make <mote>
16
17 You can change the sensor used by editing OscilloscopeAppC.nc.
18
19 Tools:
20
21 To display the readings from Oscilloscope motes, install the BaseStation
22 application on a mote connected to your PC's serial port. Then run the
23 Oscilloscope display application found in the java subdirectory, as
24 follows:
25   cd java
26   make
27   java net.tinyos.sf.SerialForwarder -comm serial@<serial port>:<mote>
28   # e.g., java net.tinyos.sf.SerialForwarder -comm serial@/dev/ttyUSB0:mica2
29   # or java net.tinyos.sf.SerialForwarder -comm serial@COM2:telosb
30   ./run
31
32 The controls at the bottom of the screen allow you to zoom in or out the X
33 axis, change the range of the Y axis, and clear all received data. You can
34 change the color used to display a mote by clicking on its color in the
35 mote table.
36
37 Known bugs/limitations:
38
```

```
39 None.
40
41
42 $Id: README.txt,v 1.6 2008/07/25 03:01:45 regehr Exp $
```

APPENDIX B

B Arduino Duemilanove interface with the DS18B20

B.1 ds18b20.cpp

```
ds18b20

#include <DallasTemperature.h>
#include <OneWire.h>
#define ONE_WIRE_BUS 2

// Setup a oneWire instance to communicate with any OneWire devices
// (not just Maxim/Dallas temperature ICs)
OneWire oneWire(ONE_WIRE_BUS);

// Pass our oneWire reference to Dallas Temperature.
DallasTemperature sensors(&oneWire);

void setup(void)
{
  // start serial port
  Serial.begin(9600);
 // Serial.println("Dallas Temperature IC Control Library Demo");

  // Start up the library
  sensors.begin();
}


void loop(void)
{
  // call sensors.requestTemperatures() to issue a global temperature
  // request to all devices on the bus
  //Serial.print(" Requesting temperatures...");
  sensors.requestTemperatures(); // Send the command to get temperatures
```

```
 // Serial.println("DONE");

  Serial.print("Temperature for Device 1 is:  ");
  Serial.print(sensors.getTempCByIndex(0));
  Serial.println();// Why "byIndex"?
    // You can have more than one IC on the same bus.
    // 0 refers to the first IC on the wire

}
```

B.2 dallastemperature.h

```
#ifndef DallasTemperature_h
#define DallasTemperature_h

#define DALLASTEMPLIBVERSION "3.7.2"

// This library is free software; you can redistribute it and/or
// modify it under the terms of the GNU Lesser General Public
// License as published by the Free Software Foundation; either
// version 2.1 of the License, or (at your option) any later version.

// set to true to include code for new and delete operators
#ifndef REQUIRESNEW
#define REQUIRESNEW false
#endif

// set to true to include code implementing alarm search functions
#ifndef REQUIRESALARMS
#define REQUIRESALARMS true
#endif

#include <inttypes.h>
#include <OneWire.h>

// Model IDs
#define DS18S20MODEL 0x10
#define DS18B20MODEL 0x28
#define DS1822MODEL  0x22

// OneWire commands
#define STARTCONVO      0x44  // Tells device to take a temperature reading and put it on the s
#define COPYSCRATCH     0x48  // Copy EEPROM
```

```
#define READSCRATCH     0xBE  // Read EEPROM
#define WRITESCRATCH    0x4E  // Write to EEPROM
#define RECALLSCRATCH   0xB8  // Reload from last known
#define READPOWERSUPPLY 0xB4  // Determine if device needs parasite power
#define ALARMSEARCH     0xEC  // Query bus for devices with an alarm condition

// Scratchpad locations
#define TEMP_LSB        0
#define TEMP_MSB        1
#define HIGH_ALARM_TEMP 2
#define LOW_ALARM_TEMP  3
#define CONFIGURATION   4
#define INTERNAL_BYTE   5
#define COUNT_REMAIN    6
#define COUNT_PER_C     7
#define SCRATCHPAD_CRC  8

// Device resolution
#define TEMP_9_BIT  0x1F //  9 bit
#define TEMP_10_BIT 0x3F // 10 bit
#define TEMP_11_BIT 0x5F // 11 bit
#define TEMP_12_BIT 0x7F // 12 bit

// Error Codes
#define DEVICE_DISCONNECTED -127

typedef uint8_t DeviceAddress[8];

class DallasTemperature
{
  public:

  DallasTemperature(OneWire*);

  // initalise bus
  void begin(void);

  // returns the number of devices found on the bus
  uint8_t getDeviceCount(void);

  // Is a conversion complete on the wire?
  bool isConversionComplete(void);

  // returns true if address is valid
  bool validAddress(uint8_t*);
```

```
// finds an address at a given index on the bus
bool getAddress(uint8_t*, const uint8_t);

// attempt to determine if the device at the given address is connected to the bus
bool isConnected(uint8_t*);

// attempt to determine if the device at the given address is connected to the bus
// also allows for updating the read scratchpad
bool isConnected(uint8_t*, uint8_t*);

// read device's scratchpad
void readScratchPad(uint8_t*, uint8_t*);

// write device's scratchpad
void writeScratchPad(uint8_t*, const uint8_t*);

// read device's power requirements
bool readPowerSupply(uint8_t*);

// get global resolution
uint8_t getResolution();

// set global resolution to 9, 10, 11, or 12 bits
void setResolution(uint8_t);

// returns the device resolution, 9-12
uint8_t getResolution(uint8_t*);

// set resolution of a device to 9, 10, 11, or 12 bits
bool setResolution(uint8_t*, uint8_t);

// sets/gets the waitForConversion flag
void setWaitForConversion(bool);
bool getWaitForConversion(void);

// sets/gets the checkForConversion flag
void setCheckForConversion(bool);
bool getCheckForConversion(void);

// sends command for all devices on the bus to perform a temperature conversion
void requestTemperatures(void);

// sends command for one device to perform a temperature conversion by address
bool requestTemperaturesByAddress(uint8_t*);
```

```
// sends command for one device to perform a temperature conversion by index
bool requestTemperaturesByIndex(uint8_t);

// returns temperature in degrees C
float getTempC(uint8_t*);

// returns temperature in degrees F
float getTempF(uint8_t*);

// Get temperature for device index (slow)
float getTempCByIndex(uint8_t);

// Get temperature for device index (slow)
float getTempFByIndex(uint8_t);

// returns true if the bus requires parasite power
bool isParasitePowerMode(void);

bool isConversionAvailable(uint8_t*);

#if REQUIRESALARMS

typedef void AlarmHandler(uint8_t*);

// sets the high alarm temperature for a device
// accepts a char.  valid range is -55C - 125C
void setHighAlarmTemp(uint8_t*, const char);

// sets the low alarm temperature for a device
// accepts a char.  valid range is -55C - 125C
void setLowAlarmTemp(uint8_t*, const char);

// returns a signed char with the current high alarm temperature for a device
// in the range -55C - 125C
char getHighAlarmTemp(uint8_t*);

// returns a signed char with the current low alarm temperature for a device
// in the range -55C - 125C
char getLowAlarmTemp(uint8_t*);

// resets internal variables used for the alarm search
void resetAlarmSearch(void);

// search the wire for devices with active alarms
```

```
bool alarmSearch(uint8_t*);

// returns true if ia specific device has an alarm
bool hasAlarm(uint8_t*);

// returns true if any device is reporting an alarm on the bus
bool hasAlarm(void);

// runs the alarm handler for all devices returned by alarmSearch()
void processAlarms(void);

// sets the alarm handler
void setAlarmHandler(AlarmHandler *);

// The default alarm handler
static void defaultAlarmHandler(uint8_t*);

#endif

// convert from celcius to farenheit
static float toFahrenheit(const float);

// convert from farenheit to celsius
static float toCelsius(const float);

#if REQUIRESNEW

// initalize memory area
void* operator new (unsigned int);

// delete memory reference
void operator delete(void*);

#endif

private:
typedef uint8_t ScratchPad[9];

// parasite power on or off
bool parasite;

// used to determine the delay amount needed to allow for the
// temperature conversion to take place
uint8_t bitResolution;
```

```
  // used to requestTemperature with or without delay
  bool waitForConversion;

  // used to requestTemperature to dynamically check if a conversion is complete
  bool checkForConversion;

  // count of devices on the bus
  uint8_t devices;

  // Take a pointer to one wire instance
  OneWire* _wire;

  // reads scratchpad and returns the temperature in degrees C
  float calculateTemperature(uint8_t*, uint8_t*);

  void        blockTillConversionComplete(uint8_t*,uint8_t*);

  #if REQUIRESALARMS

  // required for alarmSearch
  uint8_t alarmSearchAddress[8];
  char alarmSearchJunction;
  uint8_t alarmSearchExhausted;

  // the alarm handler function pointer
  AlarmHandler *_AlarmHandler;

  #endif

};
#endif
```

B.3 Onewire.h

```
#ifndef OneWire_h
#define OneWire_h

#include <inttypes.h>

#if ARDUINO >= 100
#include "Arduino.h"       // for delayMicroseconds, digitalPinToBitMask, etc
#else
```

```
#include "WProgram.h"      // for delayMicroseconds
#include "pins_arduino.h"  // for digitalPinToBitMask, etc
#endif

// You can exclude certain features from OneWire.  In theory, this
// might save some space.  In practice, the compiler automatically
// removes unused code (technically, the linker, using -fdata-sections
// and -ffunction-sections when compiling, and Wl,--gc-sections
// when linking), so most of these will not result in any code size
// reduction.  Well, unless you try to use the missing features
// and redesign your program to not need them!  ONEWIRE_CRC8_TABLE
// is the exception, because it selects a fast but large algorithm
// or a small but slow algorithm.

// you can exclude onewire_search by defining that to 0
#ifndef ONEWIRE_SEARCH
#define ONEWIRE_SEARCH 1
#endif

// You can exclude CRC checks altogether by defining this to 0
#ifndef ONEWIRE_CRC
#define ONEWIRE_CRC 1
#endif

// Select the table-lookup method of computing the 8-bit CRC
// by setting this to 1.  The lookup table enlarges code size by
// about 250 bytes.  It does NOT consume RAM (but did in very
// old versions of OneWire).  If you disable this, a slower
// but very compact algorithm is used.
#ifndef ONEWIRE_CRC8_TABLE
#define ONEWIRE_CRC8_TABLE 1
#endif

// You can allow 16-bit CRC checks by defining this to 1
// (Note that ONEWIRE_CRC must also be 1.)
#ifndef ONEWIRE_CRC16
#define ONEWIRE_CRC16 1
#endif

#define FALSE 0
#define TRUE  1

// Platform specific I/O definitions

#if defined(__AVR__)
```

```
#define PIN_TO_BASEREG(pin)             (portInputRegister(digitalPinToPort(pin)))
#define PIN_TO_BITMASK(pin)             (digitalPinToBitMask(pin))
#define IO_REG_TYPE uint8_t
#define IO_REG_ASM asm("r30")
#define DIRECT_READ(base, mask)         (((*(base)) & (mask)) ? 1 : 0)
#define DIRECT_MODE_INPUT(base, mask)   ((*((base)+1)) &= ~(mask))
#define DIRECT_MODE_OUTPUT(base, mask)  ((*((base)+1)) |= (mask))
#define DIRECT_WRITE_LOW(base, mask)    ((*((base)+2)) &= ~(mask))
#define DIRECT_WRITE_HIGH(base, mask)   ((*((base)+2)) |= (mask))

#elif defined(__MK20DX128__)
#define PIN_TO_BASEREG(pin)             (portOutputRegister(pin))
#define PIN_TO_BITMASK(pin)             (1)
#define IO_REG_TYPE uint8_t
#define IO_REG_ASM
#define DIRECT_READ(base, mask)         (*((base)+512))
#define DIRECT_MODE_INPUT(base, mask)   (*((base)+640) = 0)
#define DIRECT_MODE_OUTPUT(base, mask)  (*((base)+640) = 1)
#define DIRECT_WRITE_LOW(base, mask)    (*((base)+256) = 1)
#define DIRECT_WRITE_HIGH(base, mask)   (*((base)+128) = 1)

#elif defined(__SAM3X8E__)
// Arduino 1.5.1 may have a bug in delayMicroseconds() on Arduino Due.
// http://arduino.cc/forum/index.php/topic,141030.msg1076268.html#msg1076268
// If you have trouble with OneWire on Arduino Due, please check the
// status of delayMicroseconds() before reporting a bug in OneWire!
#define PIN_TO_BASEREG(pin)             (&(digitalPinToPort(pin)->PIO_PER))
#define PIN_TO_BITMASK(pin)             (digitalPinToBitMask(pin))
#define IO_REG_TYPE uint32_t
#define IO_REG_ASM
#define DIRECT_READ(base, mask)         (((*((base)+15)) & (mask)) ? 1 : 0)
#define DIRECT_MODE_INPUT(base, mask)   ((*((base)+5)) = (mask))
#define DIRECT_MODE_OUTPUT(base, mask)  ((*((base)+4)) = (mask))
#define DIRECT_WRITE_LOW(base, mask)    ((*((base)+13)) = (mask))
#define DIRECT_WRITE_HIGH(base, mask)   ((*((base)+12)) = (mask))
#ifndef PROGMEM
#define PROGMEM
#endif
#ifndef pgm_read_byte
#define pgm_read_byte(addr) (*(const uint8_t *)(addr))
#endif

#elif defined(__PIC32MX__)
#define PIN_TO_BASEREG(pin)             (portModeRegister(digitalPinToPort(pin)))
#define PIN_TO_BITMASK(pin)             (digitalPinToBitMask(pin))
```

```
#define IO_REG_TYPE uint32_t
#define IO_REG_ASM
#define DIRECT_READ(base, mask)         (((*(base+4)) & (mask)) ? 1 : 0)  //PORTX + 0x10
#define DIRECT_MODE_INPUT(base, mask)   ((*(base+2)) = (mask))            //TRISXSET + 0x08
#define DIRECT_MODE_OUTPUT(base, mask)  ((*(base+1)) = (mask))            //TRISXCLR + 0x04
#define DIRECT_WRITE_LOW(base, mask)    ((*(base+8+1)) = (mask))          //LATXCLR  + 0x24
#define DIRECT_WRITE_HIGH(base, mask)   ((*(base+8+2)) = (mask))          //LATXSET + 0x28

#else
#error "Please define I/O register types here"
#endif


class OneWire
{
  private:
    IO_REG_TYPE bitmask;
    volatile IO_REG_TYPE *baseReg;

#if ONEWIRE_SEARCH
    // global search state
    unsigned char ROM_NO[8];
    uint8_t LastDiscrepancy;
    uint8_t LastFamilyDiscrepancy;
    uint8_t LastDeviceFlag;
#endif

  public:
    OneWire( uint8_t pin);

    // Perform a 1-Wire reset cycle. Returns 1 if a device responds
    // with a presence pulse.  Returns 0 if there is no device or the
    // bus is shorted or otherwise held low for more than 250uS
    uint8_t reset(void);

    // Issue a 1-Wire rom select command, you do the reset first.
    void select(const uint8_t rom[8]);

    // Issue a 1-Wire rom skip command, to address all on bus.
    void skip(void);

    // Write a byte. If 'power' is one then the wire is held high at
    // the end for parasitically powered devices. You are responsible
    // for eventually depowering it by calling depower() or doing
    // another read or write.
```

```
    void write(uint8_t v, uint8_t power = 0);

    void write_bytes(const uint8_t *buf, uint16_t count, bool power = 0);

    // Read a byte.
    uint8_t read(void);

    void read_bytes(uint8_t *buf, uint16_t count);

    // Write a bit. The bus is always left powered at the end, see
    // note in write() about that.
    void write_bit(uint8_t v);

    // Read a bit.
    uint8_t read_bit(void);

    // Stop forcing power onto the bus. You only need to do this if
    // you used the 'power' flag to write() or used a write_bit() call
    // and aren't about to do another read or write. You would rather
    // not leave this powered if you don't have to, just in case
    // someone shorts your bus.
    void depower(void);

#if ONEWIRE_SEARCH
    // Clear the search state so that if will start from the beginning again.
    void reset_search();

    // Setup the search to find the device type 'family_code' on the next call
    // to search(*newAddr) if it is present.
    void target_search(uint8_t family_code);

    // Look for the next device. Returns 1 if a new address has been
    // returned. A zero might mean that the bus is shorted, there are
    // no devices, or you have already retrieved all of them.  It
    // might be a good idea to check the CRC to make sure you didn't
    // get garbage.  The order is deterministic. You will always get
    // the same devices in the same order.
    uint8_t search(uint8_t *newAddr);
#endif

#if ONEWIRE_CRC
    // Compute a Dallas Semiconductor 8 bit CRC, these are used in the
    // ROM and scratchpad registers.
    static uint8_t crc8(const uint8_t *addr, uint8_t len);
```

```
#if ONEWIRE_CRC16
    // Compute the 1-Wire CRC16 and compare it against the received CRC.
    // Example usage (reading a DS2408):
    //    // Put everything in a buffer so we can compute the CRC easily.
    //    uint8_t buf[13];
    //    buf[0] = 0xF0;    // Read PIO Registers
    //    buf[1] = 0x88;    // LSB address
    //    buf[2] = 0x00;    // MSB address
    //    WriteBytes(net, buf, 3);    // Write 3 cmd bytes
    //    ReadBytes(net, buf+3, 10);  // Read 6 data bytes, 2 0xFF, 2 CRC16
    //    if (!CheckCRC16(buf, 11, &buf[11])) {
    //        // Handle error.
    //    }
    //
    // @param input - Array of bytes to checksum.
    // @param len - How many bytes to use.
    // @param inverted_crc - The two CRC16 bytes in the received data.
    //                       This should just point into the received data,
    //                       *not* at a 16-bit integer.
    // @param crc - The crc starting value (optional)
    // @return True, iff the CRC matches.
    static bool check_crc16(const uint8_t* input, uint16_t len, const uint8_t* inverted_crc, u

    // Compute a Dallas Semiconductor 16 bit CRC.  This is required to check
    // the integrity of data received from many 1-Wire devices.  Note that the
    // CRC computed here is *not* what you'll get from the 1-Wire network,
    // for two reasons:
    //   1) The CRC is transmitted bitwise inverted.
    //   2) Depending on the endian-ness of your processor, the binary
    //      representation of the two-byte return value may have a different
    //      byte order than the two bytes you get from 1-Wire.
    // @param input - Array of bytes to checksum.
    // @param len - How many bytes to use.
    // @param crc - The crc starting value (optional)
    // @return The CRC16, as defined by Dallas Semiconductor.
    static uint16_t crc16(const uint8_t* input, uint16_t len, uint16_t crc = 0);
#endif
#endif
};

#endif
```