Project 4B
Beaglebone Sensors

INTRODUCTION:

One of the things that makes small, low-power-consumption systems interesting is ability they give us to create software that interacts with people in the physical world. The input side of those interactions is via a variety of local sensors (e.g. for sound, light, temperature, position, motion, pressure). The output side can be sensory (e.g. sound, lights, low power displays) or physical (e.g. mechanical actuators), or (through digital and analog outputs) any other controllable device. Building interactive personal appliances requires us to be able to communicate with the types of sensors, indicators and actuators that can be connected to such systems.

In this project we will create applications that run in an embedded system, read data from external sensors, and log the results.

RELATION TO READING AND LECTURES:

This project is an introductory exploration of technologies and services not covered in the reading and lectures.

PROJECT OBJECTIVES:

- Primary: Demonstrate the ability to design, build and debug interactive applications on an embedded system.
- Primary: Develop the ability to work with embedded system sensors and actuators through standard tool kits.
- Primary: Gain experience with the processing of Digital and Analog inputs on an embedded system.

DELIVERABLES:

A single compressed tarball (`.tar.gz`) containing:

- C source files for an embedded application that builds and runs (with no errors or warnings) on a Beaglebone.
- A `Makefile` to build and test your application. The higher level targets should include:
  - default ... build your program (compiling with the **`-Wall`** and **`-Wextra`** options).
  - **check** ... execute an automated smoke-test of your application to see if it runs and can talk to its sensors/actuators.
  - **clean** ... delete all programs and output generated by the Makefile
  - **dist** ... (runnable on a Linux desktop or server) create the deliverable tarball

  Note that this `Makefile` is intended to be executed on a Beaglebone.
- a `README` file containing:
  - descriptions of each of the included files
  - any other comments on your submission submission that you would like to bring to our attention (e.g. research, limitations, features, testing methodology).

PREPARATION:

- If you have not already done so, look at the documentation for the poll(2) system call or the `O_NONBLOCK` option used with the open(2) system call or fcntl(2) `F_SETFL` operation.
- Read the documentation on the Grove Temperature Sensor, and the algorithm for converting a reading into a temperature.
- Review the documentation for the MRAA library, an open source, platform independent library for embedded system I/O. (The MRAA page does not explicitly list the Beaglebone Green Wireless, the device you will be using, as supported, but it is supported.) Also read the General Purpose and Analog I/O tutorial that discusses the particular classes you will be using. These classes include the functions you will be using to read from the

temperature sensor and button. (Note that to link with this library you will have to add `-lmraa` to your library search list).
- Read the instructions that came with your Grove Starter Kit. They include information on all of your sensors and instructions on where to plug each in to the Base Shield.
- Attach your Grove Base Shield (a plug-on daughter-board with connectors for all of your sensors) to your Beaglebone/Arduino board.
- Attach your Grove Temperature Sensor to the A0 connector on your base shield.
- Attach your Grove Button to the D3 connector on your base shield.
- Power-up your Beaglebone and confirm that you can still log in and trasfer files to it.

PROJECT DESCRIPTION:

Write a program (called `lab4b`) that:

- builds and runs on your Beaglebone.
- uses the AIO functions of the MRAA library to get readings from your temperature sensor.
- samples a temperature sensor at a configurable rate (defaulting to 1/second, and controlled by an optional `--period=#` command line parameter that specifies a sampling interval in seconds).
- converts the sensor value into a temperature. By default, temperatures should be reported in degrees Fahrenheit, but this can be controlled with an optional `--scale=C` (or `--scale=F`) command line parameter.
- creates a report for each sample that includes:
  - time of the sample (e.g. `17:25:58`) in the local timezone
  - a single blank/space
  - a decimal temperature in degrees and tenths (e.g. `98.6`)
  - a newline character (\n)
- writes that report to the stdout (fd 1).
- appends that report to a logfile (which it creates on your Beaglebone) if that logging has been enabled with an optional `--log=`*filename* parameter.
- uses the GPIO functions of the MRAA library to samples the state of the button (from your Grove sensor kit) and when it is pushed ...
  - outputs (and logs) a final sample with the time and the string `SHUTDOWN` (instead of a temperature).
  - exits
- Your program can assume that the sensors are connected as recommended by the Grove documentation:
  - The temperature sensor to Analog input 0.
  - The push-button to Digital input 3.

Note that we want you to use MRAA AIO/GPIO functions to access your sensors, and not the (more powerful and convenient) Grove library functions. The Grove library hides all of the details of embedded I/O, sampling and signal interpretation, but only works for the Grove Sensors. We want you to get experience with direct control of and access to the digital and analog I/O pins.

Many people have observed that the recommended calibration constants appear to be off by ten degrees or more. Every sensor (including the ones that will be used when we grade your submissions) seems to read differently.

The time returned from *localtime(3)* will only be in the correct timezone if you have correctly configured the local timezone on your Beaglebone.

Extend your program to (in parallel with generating reports) accept the following commands from stdin:

- **`SCALE=F`**
  This command should cause all subsequent reports to be generated in degrees Fahrenheit.
- **`SCALE=C`**
  This command should cause all subsequent reports to be generated in degrees Centegrade
- **`PERIOD=`***seconds*
  This command should change the number of seconds between reporting intervals. It is acceptable if this command does not take effect until after the next report.
- **`STOP`**
  This command should cause the program to stop generating reports, but continue processing input commands. If

the program is not generating reports, merely log receipt of the command.

- **START**
  This command should cause the program to, if stopped, resume generating reports. If the program is not stopped, merely log receipt of the command.
- **OFF**
  This command should, like pressing the button, output (and log) a time-stamped SHUTDOWN message and exit.

All received commands will be terminated by a new-line character ('\n'). Note that when your program is tested (by the sanity checker or grading program) stdin will not be a console, but a pipe. A single read may return a partial line or multiple lines. Make sure that your program buffers and lexes its input, and does not assume that one call to *read(2)* will return one command.

If logging is enabled, all received commands should be appended to the log file (exactly as received, with no timestamp) but not displayed to standard output. A sample log is shown below:

```
11:37:41 98.6
11:37:42 98.6
11:37:43 98.6
PERIOD=5
11:37:44 98.6
11:37:49 98.6
11:37:54 98.6
SCALE=C
11:37:59 37.0
11:38:04 37.0
STOP
START
11:38:19 37.0
11:38:24 37.0
OFF
11:38:27 SHUTDOWN
```

If you are typing commands to your program, you may see the echos interspersed with temperature reports. This is unimportant, because they should be correctly distinct in your log, and when we test your submission, the input will becoming from a program rather than a console.

The sanity check script will send a series of commands to your program, and some implementations process all the commands before generating any reports. To make sure this doesn't happen, you should generate your first report before you start processing input commands. Again, this will not be an issue during testing because there will be delays between the commands when we test your program.

You could implement this program with separate threads for sensor polling and command processing, but the computation associated with each operation is so small that you may find it much simpler to use a single thread (and polling or non-blocking I/O) to avoid hanging on standard input for commands that only rarely arrive. A poll system call cannot tell you that the button has been pushed, but GPIO pin read is a non-blocking operation, so you can simply read the button status once per second.

To facilitate development and testing you might find it helpful to write your program to, if compiled with a special (`-DDUMMY`) define, include mock implementations for the `mraa_aio_` and `mraa_gpio_` functionality. Doing so will enable you to do most of your testing on your regular computer. When you are satisfied that it works there, modify your Makefile run the command "`uname -r`", check for the presence of the string "`beaglebone`" in that output, and if not found, build with a rule that passed the `-DDUMMY` flag to *gcc*.

SUBMISSION:

Your tarball should have a name of the form `lab4b-`*studentID*`.tar.gz`. You can sanity check your submission with this test script. It should run on a Beaglebone, but if your program is (as above) runnable on your Linux development system, the sanity check script should also run there. There will be no manual re-grading on this project. Submissions that do not pass the test script are likely to receive very low scores.

Your **README** file (and all source files) must include lines of the form:

NAME: *your name*
EMAIL: *your email*
ID: *your student ID*

And, if slip days are allowed on this project, and you use some:

SLIPDAYS: *your student ID,#days*

GRADING:

Points for this project will be awarded:

value	feature
	Packaging and build (10% total)
3%	un-tars expected contents
3%	clean build of correct program w/default action (no warnings)
2%	Makefile has working `clean`, `check`, `dist` targets
2%	reasonableness of `README` contents
	Sensor Functionality (25% total)
10%	samples and reports temperature
5%	reports Fahrenheit temperature (by default)
5%	reports Centegratde temperature (w/`--scale=` parameter)
5%	samples button and exits
	Control Functionality (40% total)
10%	command and data logging
10%	`START`/`STOP` commands
10%	`PERIOD=#` command
5%	`SCALE=C`/`F` commands
5%	`OFF` command
	Code Review (25%)
10%	use of AIO functions to read temperature
10%	use of GPIO functions to read button
5%	general readability and understandability