PWM Motor Control Based on Sensed Temperature

Clayton Crawford, Jeff Terrell, Michael Bass, Taahir Ahmed, Desmond Uzor,
Narayanan Rengaswamy, Dipanjan Saha, Ashton Jackson

ABSTRACT

The intent of this project is to control the voltage supplied to a motor-driven fan based on the temperature sensed by a thermistor. The fan starts rotating whenever the measured temperature exceeds a threshold, and the motor voltage vis-à-vis the fan speed is higher for a higher value of temperature above the threshold. There are two thermistors, T_1 and T_2, and two motors, M_1 and M_2, such that M_1 responds only to T_1 and M_2 only to T_2. To accomplish the task in real time, the team utilizes JSON, a SQL database, and QT visualization software to read, store, and graphically display data.

1. INTRODUCTION

This one-and-a-half week exercise is intended to provide the team with a hands-on experience on system-level integration, i.e. how to conceive of and build a working system as a disciplined assembly of mutually communicating functional units. Each unit consists of a number of hardware and/or software components. The components utilized include:

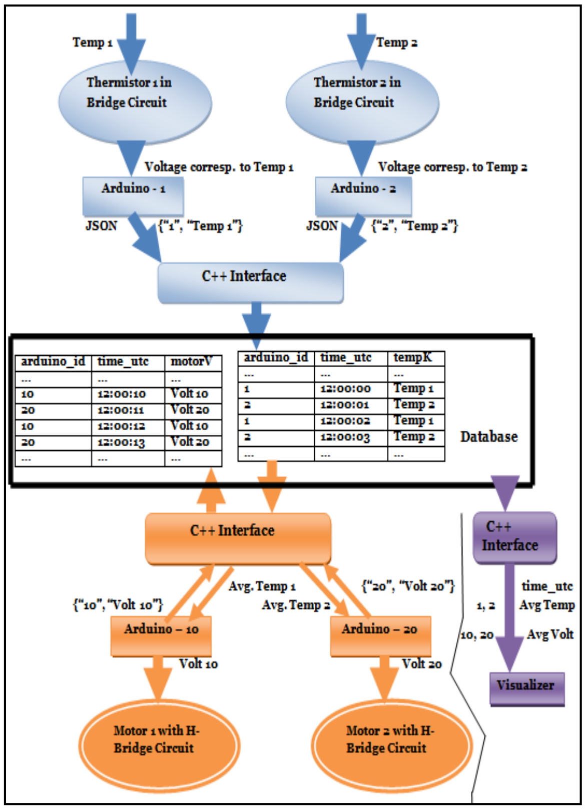
(i) <u>Hardware</u> – 4 Arduino UNOs, 2 thermistors, 2 motors, 2 L298N H-bridge ICs.

(ii) <u>Software Platforms</u> – Arduino, Eclipse, PostgresQL, Qt Visualization

Equipped with the above requirements, let us now proceed to the next section which illustrates the system through a schematic and also dissects the functional units.

2. SYSTEM OVERVIEW

A schematic of the system is shown in Figure 1. It is made up of four functional units – the database, the input, the control and the visualization. In the input side, the temperatures sensed by thermistors are converted to voltages via bridge circuit and read by the input Arduinos which are programmed to give back the measured temperature values. The input Arduinos encode their IDs and the temperature values using JavaScript Object Notation (JSON) and write the same onto the PostgreSQL database through a C++ interface. The output side C++ interface reads those values, computes the average of last ten stored temperatures for each input Arduino and sends the average to the respective control Arduino. At this point, the control Arduinos generate the voltage commands for the motors and, like the input side Arduinos, encode their respective Arduino IDs and voltage readings as JSON objects. The C++ interface reads and decodes the JSON objects and writes the Arduino IDs and voltage readings onto the database. The visualization unit reads and computes the averages of last ten stored temperature and voltage readings separately and plots them on the screen.



arduino_id	time_utc	motorV
...	...	...
10	12:00:10	Volt 10
20	12:00:11	Volt 20
10	12:00:12	Volt 10
20	12:00:13	Volt 20
...	...	...

arduino_id	time_utc	tempK
...	...	...
1	12:00:00	Temp 1
2	12:00:01	Temp 2
1	12:00:02	Temp 1
2	12:00:03	Temp 2
...	...	...

*Figure 1: System schematic showing the **input**, **database**, **control** and **visualization** units*

3. FUNCTIONAL UNITS

3.1. Database – The Common Link

Taahir Ahmed, Desmond Uzor, Ashton Jackson

The SQL database stores readings from the input side JSON and feedback from the control system.

The database is named **motor_team** and is located on **fulla.ece.tamu.edu**, with username **motor_team_user** and password **motor_team_password**. It must be accessed via the campus network or a VPN connection.

The temperatures read from the input side are stored in a table named **temp_readings**, with fields **reading_time_utc** (UTC timestamp), **arduino_id** (integer), and **temp_kelvin** (real scalar value in Kelvin). Similarly, the control side feedback data is stored in a table named **motor_readings** with fields **reading_time_utc** (UTC timestamp) and **motor_voltage** (real scalar value in Volts).

Values are added to the database using postgresql. The function PqconnectdbParams is used to connect to the database. Commands are sent to the database using PqexecParams, including commands for inserting values into the database tables. Pqgetvalue reads values from the database tables while specifying the number of rows using Pqntuples.

3.2. Input Side: JSON

Michael Bass

The input side is comprised of thermistors, Arduinos, the control program, and writing the recorded data to a database.

The thermistor's resistance is dependent on the temperature, and since we are using NTC thermistors, the resistance of the thermistor will decrease as its temperature increases. Each Arduino is equipped with its own resistor and thermistor, allowing each Arduino to be able to collect different data.

The Arduino samples the voltage across a thermistor and then digitizes the value with an analog to digital converter. The voltage is then used to calculate the resistance of the thermistor. Once the resistance of the thermistor is known the temperature is calculated using the Steinhart-Hart equation:

$$\frac{1}{T} = A + B\ln R + C\,(\ln(R))^3$$

The A, B, and C coefficients of the Steinhart-Hart equation were determined using a table of given values, and then solving for the coefficients. This will result in a small amount of inaccuracy because no thermistor will mirror the exact same response to temperature as the values reported in the datasheet. For a more accurate measurement, each individual thermistor can be calibrated separately.

The Arduino is connected to a PC using a serial connection over USB. The Arduino packages the temperature value into a JSON object that has a single field, tempK, containing the scalar temperature value in Kelvin. For example, at a room temperature of 300 K:

$$\{\ \text{" tempK "} : 300\ \}$$

A JSON object is composed of a series of fields and values. Each field in a JSON object is a string surrounded by " ". In the example above, the name of the field is tempK. Each field must have an associated value. The format is " fieldName" followed by a colon(:), and then the value of the field. The value of a field can be a literal constant, such as a string, integer, floating point number, an array, or another object. For our use it sufficed to have only one field with the temperature presented as a floating point number. A JSON object must also be enclosed with { }.

The Arduino follows a hand shaking protocol with the control program. Once the Arduino writes the JSON object to the control program it waits for the control program to write back the character '1' over the serial line. Once the Arduino has received a '1', it will take another temperature reading, place it in a JSON object, and write that object out to the control program. The hand shaking protocol forces the Arduino to not send data faster than the control program can receive and process the data. If the control program were to lag behind the number of readings being recorded, then changes in temperature would go unnoticed until the control program is able to process the data revealing the temperature change.

The JSON object is sent over the serial line to the PC. The PC runs the control program and receives JSON objects containing temperature readings from multiple Arduinos. The control program performs two checks on the JSON object before extracting the temperature value. First, the JSON object is checked to ensure that it contains the tempK field. If the JSON object does not contain the tempK field then the reading is considered a bad reading and will be discarded and not recorded in the database. Second, the JSON object is inspected to ensure that it follows proper JSON format. If the object fails to be a properly formatted JSON object then it risks being a corrupted or incomplete read from the serial input. Objects that are not properly formatted JSON objects are considered to be bad readings and are discarded and not recorded in the database. If both of these checks are passed then the temperature value is extracted and recorded in the database.

Both of these checks are performed using a user defined Reading class. By passing the JSON object, a string, to the constructor of the Reading class, the checks are automatically applied to the JSON object. If either of the checks failed, then the member function GetTemperature function of the Reading class will return negative one. When an error occurs the Reading class provides an error message that can be read to determine the cause of the error. If there were no errors while parsing the JSON object, then GetTemperature will return the value of the temperature recorded by the Arduino.

The control program can host an unlimited number Arduinos being connected to the PC, given that the PC does not run out USB ports or computing resources. The control program uses multithreading to handle the variable number of Arduinos. Each connection is created in its own thread that allows the PC to reduce timing difficulties, and easily coordinate between which packets came from which Arduino. The parameters for how many Arduinos will be connected, and which COM Ports will be utilized must be given when starting the control program.

The control program must also write the value of each reading to a database located on a network. We achieve this by using the libpq library. The first step to writing to a database is to connect to the database. We use the PQconnectdb function to connect to the database and pass to it a string containing the network address of the database. Using the libpq library, SQL queries can be created out of strings, and then passed to the PQexec function. The PQexec function takes as parameters the database connection, and the query to be executed. The query we create is a simple SQL insert command:

```
"INSERT INTO temp_readings(arduino_id, reading_time_utc, temp_kelvin) VALUES (" + valueOfArduinoId + ", 'NOW', " + valueOfTempReading + ")"
```

This insert statement above adds a new record to the database in the temp_readings table. The arduino_id is determined when the thread is launched, and is retained for every insert. The reading_time_utc is determined by the database. Using the 'NOW' command in the SQL query tells the database to timestamp the field with the time when the record is created. The last value is the temperature value transmitted by the Arduino. Every time an Arduino sends a temperature reading, if it is valid it will be recorded in the database.

3.3. Output Side: Control

Dipanjan Saha, Clayton Crawford, Jeff Terrell

Stages

The control side is comprised of a few stages. The first stage involves a PC receiving an average temperature reading for the latest ten temperature readings of a particular input Arduino by fetching the temperature readings from the database, and the second involves using the average temperature reading to control the speed of a motor.

Once data from the input-side is stored in the SQL database, it is ready to be read, processed, and used for controlling the motors. The motors will provide feedback in the form of a JSON object containing the unique Arduino ID and the motor's approximated voltage level. This information is then written back to the SQL database to be interpreted and visualized.

In this stage, the data is read from the database and then filtered by calculating a moving average. This acts as a buffer between rapid temperature fluctuations and the response of the motor. The data is then packaged into a JSON object and sent over a serial port to be read by an output-stage Arduino. The Arduino parses the JSON objects and retrieves meaningful data.

This data is then used to set a PWM value to send to an L298N dual H-Bridge motor controller IC.

The output side schematic is shown in Figure 2.

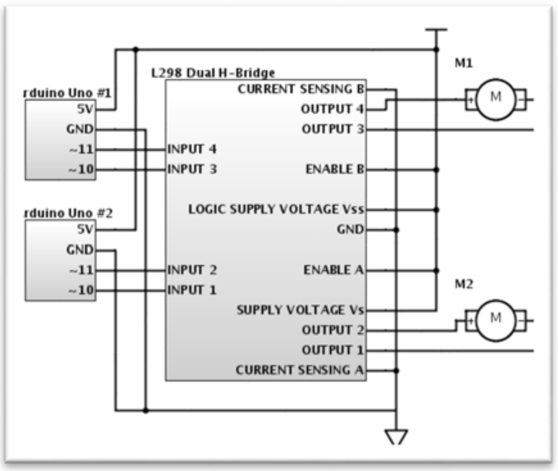


Figure 2: Control system schematic

The motor voltages that are sent back to the database are not measured values. Instead, a given motor speed is assigned a voltage based on the PWM duty cycle. This data is transmitted over the serial line and placed in the **motor_voltage** table, along with the Arduino ID. The timestamp is created at the time of insertion into the table. This is done by populating a row and excluding the timestamp field, filling the timestamp record with the default data type.

Difficulties

Originally, the control side C++ program was being developed under a Linux operating system. However, various difficulties and lack of time required Michael to step in and reconfigure the program to be used in a Windows environment.

3.4. Visualization

Narayanan Rengaswamy, Taahir Ahmed

In order to build this project, we need to install Qt, PostgreSQL and the QPSQL plugin for Qt on Windows. The detailed directions for installation are provided in Appendix A. Now, we need to make use of the graphical capabilities of the QCustomPlot library. A specific example of QCustomPlot describes how to develop a real-time graph. This was used to initially understand the functioning and was later modified to suit the need of this project. The basic idea is to first setup the graph configurations such as number of functions to be plotted, axes labels etc. Then, a timer is set to call a particular slot on its timeout() signal. This allows for the speed of the plotting to be adjusted. The slot that is called contains the code to interface with the database, execute queries, fetch results in local variables and finally plot them on their corresponding graphs in the GUI.

This project requires the team to plot the graphs for Temperature vs. Time for 4 thermistors and Motor Voltage vs. Time for 4 motors that are driven by the thermistors. Thus, four graphs were created in the project for temperatures and voltages, resulting in 8 graphs. The GUI, when run, asks for the mapping of the Arduino IDs connected to the thermistors and the motors. The graph only has 4 colors even though there are 8 graphs being plotted. One of the same-colored graphs corresponds to the temperature and the other to the voltage, for each of the 4 pairs. This enables easier interpretation of the control system when temperatures vary. Also, to be able to plot temperatures and voltages on the same graph, the vertical axes range has to remain constant. So, the temperatures were converted to “degrees Celsius” so that they typically occupy a range of 0-40. The motor voltages are in the range 0-5V, so they were multiplied by 8 to fit the range of the temperatures. Thus, the visualization was made far better than plotting temperatures and voltages on two separate GUI windows. A snapshot of this is provided in Figure 3, seen in Appendix A.

4. INTEGRATION & RESULTS

There are two main communication protocols in use between components of the system. On the serial links between the Arduinos and their controlling computers, JSON-serialized information is used. For the arduinos performing the temperature sensing, the JSON objects are of the following form:

{ "tempK" : <floating-point temperature in Kelvin> }

For the arduinos controlling the motors, there is communication in both directions. The JSON objects sent from the controlling computer to the arduino, the objects have the form:

```
{
    "arduinoID" : <integer unique ID>,
    "averageTemp" : <floating-point average
temperature in Kelvin>
}
```

From the arduino to the controlling computer:

```
{
    "arduinoID" : <integer unique ID>,
    "motorVoltage" : <floating-point average
temperature in Kelvin>
}
```

The other main communication protocol is the PostgreSQL SQL variant used by libpq for communicating with the database. This communication is structured around the tables defined earlier.

Once integration was completed, the system worked well and was reasonably responsive. Adjusting the temperature at each temperature sensor evoked a response from its associated motor, and the visualization system was able to siphon data points from the database and perform visualization.

5. RESOURCES

PostgreSQL Documentation:
http://www.postgresql.org/docs/9.3/static/libpq.html

L298N datasheet:
https://www.sparkfun.com/datasheets/Robotics/L298_H_Bridge.pdf

6. APPENDIX A – VISUALIZATION TUTORIAL

6.1. Introduction to Qt

Qt is a cross-platform project developed for C++ programmers to build effective visualizations for various applications. On Windows, it can run either on the Visual Studio compiler or on the MinGW (Mimimalist GNU for Windows) compiler but the setup file for both of them are different. The setup includes all the necessary binaries and headers to build a Qt project. It also comes with the Qt Creator tool to facilitate the development of a GUI for the programmer's application. The Qt Assistant tool can be used for any reference regarding the complete Qt project. It includes all the classes, functions that are used to develop the project. The setup file for the complete package is available for free from their official website: http://qt-project.org/downloads. But, it is important to download the correct setup file according to the compiler the user wishes to run it with.

Just like any other project, Qt has a whole lot of example projects to make the programmer get accustomed to the development environment. Even though Qt includes all C++ headers, it is generally advisable to use Qt's corresponding custom headers to make much better use of the environment. For example, even though one can include <string.h> to perform string operations in his application, it's more advisable to use the QString data type because the standard "string" data type may not interface with many Qt functions. Similarly, "qDebug()" is used instead of "std::cout". Once the programmer browses through a few projects, it's very easy to get accustomed to this "custom" environment of Qt.

6.2. A Qt Project's General Architecture

Every project consists of a "project" file with a ".pro" extension. This contains the general settings for the project and includes the names of all files that come along with the project categorized as Source files, Header files etc. If one creates a new Qt project without changing any default file names, then the project consists of a "main.cpp" file, a "mainwindow.cpp" file, a "mainwindow.h" header file and a "mainwindow.ui" file.

The "main.cpp" file just instantiates the QWindow class to create a new output window when the project is built and run. Then it transfers the control over to the "mainwindow.cpp" file which is the complete "brain" of the project. Any global declarations can be done in the "main.cpp" file with the extern keyword prefixed so that they can be accessed in every function (internal and external) of the "mainwindow.cpp" file. As every C++ programmer would guess, the "mainwindow.h" file has just has the declarations for every function, signal and slot defined in the corresponding .cpp file. The “signal and slot” mechanism of Qt is explained next.

All GUI development tools will have a mechanism to trigger a code based on an event that happens when the user interacts with the GUI. For Qt, this is the “signal and slot” mechanism. There are “signals” that the programmer define on specific events on different objects included in the GUI. While the events are defined by Qt, the signals to be “emitted” are defined by the user. These signals not only carry information defined in their code but also carry some sort of an encrypted code which will be used to authenticate the signal at its slot. The slot is again any function defined in any class of the project that is assigned to receive this signal using the “connect()” statement defined for every signal. Thus, any “emitted” signal can be defined to reach any “slot” which has the code to be performed corresponding to the event triggered by the user’s interaction. In a nutshell, this is what happens: the user clicks a button or hovers over a field or drags the scroll bar or does any such actions on an object in the GUi; this triggers a signal defined by the programmer for that event; the connect() statement directs the signal to the corresponding “slot”; the slot authenticates the signal with its code (which is not in the programmer’s control); then, it executes the code defined under it to produce a feedback for the user’s interaction.

On executing the “mainwindow.show()” in “main.cpp”, the code “MainWindow::MainWindow(QWidget *)” is first executed. Then, depending upon the flow of the code defined by the programmer, the other functions, signals and slots are executed. The “qDebug() << ” command can be used to find the state of the application anytime. The string to the right of the “<<” is output on the Qt Console (which is integrated with the Qt Creator tool itself).

The “mainwindow.ui” file defines the GUI completely. Qt Creator allows the developer to “drag and drop” GUI components directly and creates the corresponding xml file automatically thereby enabling the programmer to concentrate on improving the “working” of the application rather than spending an equivalent time on creating the GUI directly by tedious coding.

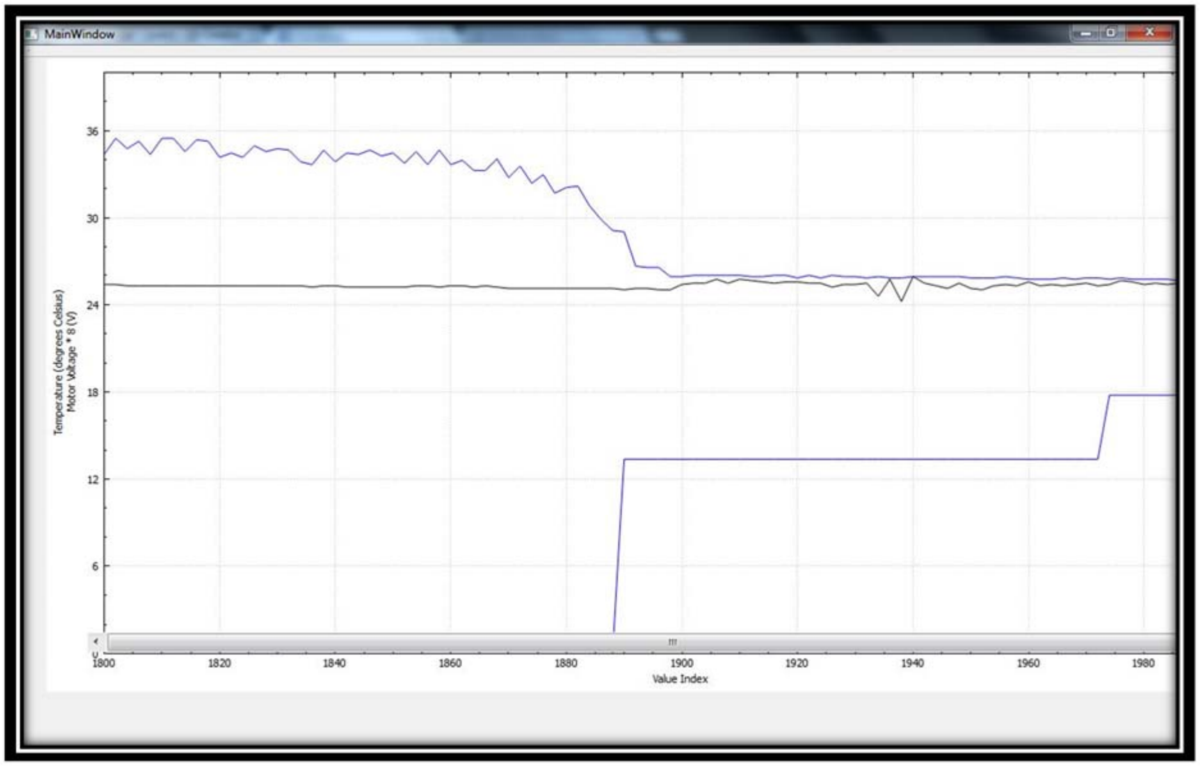
6.3. QCustomPlot Library & QPSQL Driver

The application that was intended for this project was to plot graphs in real-time by accessing a PostgreSQL database. So, it required a well-defined library to do this in a short span of time. The QCustomPlot library has extensive graphing capabilities which can be used to do almost everything that one wishes to see in a graph. It just contains a header named “qcustomplot.h” and a source “qcustomplot.cpp” which needs to be included in the project to be able to use it. Fortunately, it also comes with a lot of examples that demonstrate plotting of different type of graphs and also real-time ones. The complete documentation and downloads can be found at http://www.qcustomplot.com/index.php/introduction.

PostgreSQL can be downloaded and installed for Windows from the following link: http://www.postgresql.org/download/windows/. It has its own GUI tool “pgAdmin” to enable easier interaction directly with the database. The installation directory has to be used next for QPSQL plugin.

The next task is to be able to access the PostgreSQL database for which Qt needs a driver. The QPSQL driver is the one for this purpose and can again be integrated with Qt depending on whether Qt runs on the Visual Studio compiler or the MinGW compiler. Since the Qt project was installed to run with the MinGW compiler on Windows for this project, the steps to be followed to include the driver were followed as described in http://www.qtcentre.org/wiki/index.php?title=Building_the_QPSQL_plugin_on_Windows_using_MinGW. This is a straightforward process except when the versions of Qt mismatch with the one given in the link. In such case, change the “%QTDIR%” given in the webpage to be “C:\QtMinGW\Qt5.1.1\5.1.1\Src\qtbase” if the Qt installation directory is “C:\QtMinGW\Qt5.1.1”. Other than this, the installation of the QPSQL plugin is troublefree.

Figure 3 – The Visualization Screenshot


MainWindow
Temperature (degrees Celsius)
Motor Voltage * 8 (V)
36
30
24
18
12
6
0
1800
1820
1840
1860
1880
1900
1920
1940
1960
1980
Value Index