Todo list

Project SB3: Data Logger
Package Environment Monitoring
Final Report

Andrew Holt
ah635
Team 6
Emmanuel

06 June 2013

summary: motive, method, key results, conclusions

add summary/abstract

Contents

List of Figures

List of Tables

1 Introduction

The express delivery industry is one of the fastest growing industries in the world today. In 2003, the industry made a direct contribution of US$64 billion to worldwide GDP, and has been growing by around 8% per year since [2]. However, customers have very little information of the environmental and handling conditions their package has been subject to during its transit. When shipping fragile or sensitive products, the conditions that the parcel is subject to may be critical in the integrity of the product. The two major issues here are:

1. Fragile goods which arrive broken: it is desirable to determine whether the package has been mishandled, and if so by whom.

2. Perishable goods: if they have been subjected to conditions outside of sensitive temperature and humidity limits they will have a shorted shelf life. This may not be obvious upon inspection of the goods, so detailed condition tracking is required.

While products have been around for a while to carry out this type of tracking on shipping containers, the same problem applies to smaller and domestic packages too. In the past six months, two large companies have announced products to achieve this, but it is clearly an emerging market and a good product launched now could have a great chance of success. One of these, "SenseAware"[4] by FedEx Corporation, is already released and available however this is only available on a select number of carriers and services and is a business level product, not suited for everyday or relatively low value package monitoring. An alternative is the "DropTag" system recently announced by Cambridge Consultants [1], however this system is not yet at the commercial release stage.

With a problem identified and a solution required, as well as a clear business opportunity due to the interest in development from other companies, product development was begun.

2 Overview

In order to effectively track the condition of a parcel during transit, the final product was clearly required to be cheap, low power and physically small and robust. For this project, a simple prototype was to be developed, which if successful could be miniaturised and fabricated for the final product.

2.1 Summary of Design

The first stage of the design process was to evaluate which quantities should be measured and logged. The key environmental variables would be temperature and humidity. Vibration, orientation and shocks would also be required to understand the handling and transport conditions. GPS tracking would also be useful, but it was decided to focus on the other five initially as GPS could be relatively easily included with a dedicated GPS chip communicating over the I^2C protocol, and GPS reception may be limited at many points during transit.

The product would clearly be required to work without being plugged in to a computer, so a standalone mode would be developed. It would be useful for development, debugging and testing to operate with instant feedback from the computer system, so a linked mode was also developed.

To ensure long term operation, on-board memory would be required, so an EEPROM was used for data storage in the standalone mode. This would also require a battery supply.

Some kind of on-board display would also be useful on the board, for monitoring, debugging and status display, so an LCD screen was used along with five LEDs.

The platform choices were specified in the project requirements, so the project was to be developed for the STM32F100 ARM Cortex microcontroller and a PC running the Windows operating system. The key design areas would therefore be hardware, firmware and software. The budget was generous for the development board so cost was not really an issue, however to achieve a commercial product, the cost and size would need to be reduced. This could be achieved by miniaturising, as surface mount components tend to be cheaper; and use of a cheaper microcontroller (which wouldn't need the development features of the STM).

The project block diagram is shown in figure 1.

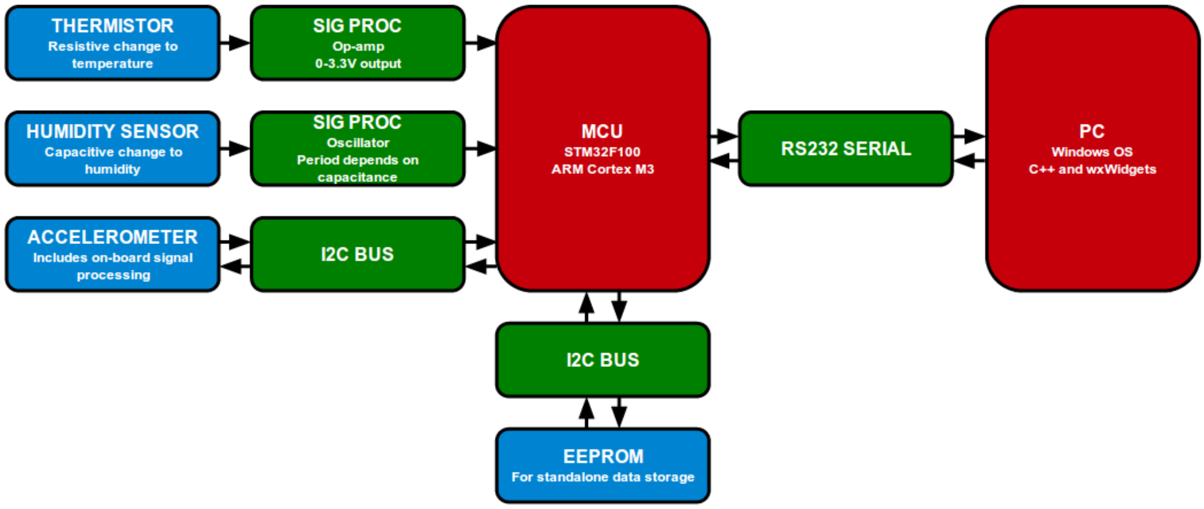

Figure 1: Overall system block diagram

2.2 Project Management

The project was divided between the two team members. It was decided that James would write the firmware for the microcontroller, Andrew would write the software for the computer and the hardware design was to be shared. The general development plan was to start by making a basic but working system and gradually adding features and complexity upon the existing system. This ensured that we would have a fully operational product, if not with every desired feature, by the end of the project, and would also allow the simpler parts to be used in testing more complex parts.

Initially basic hardware circuits were designed and built. The firmware and software were then developed. The hardware circuits were highly useful in testing the firmware, but in writing the firmware, flaws in the circuit designs became apparent, such as the initial plan to use a simple charge-discharge of the humidity sensor. For this reason, the humidity sensor circuit was updated during the project.

3 Design

A brief overview of the design has been given in section 2.1. This section gives technical detail about the parts of the system which I designed. The other parts are detailed in James' report.

add photo of completed board

3.1 Hardware

In order to measure temperature, it was decided to use a thermistor. Thermistors have advantages over other temperature measurement devices such as thermocouples, resonant crystal thermometers and diode junction voltage measurement such as [3]:

1. The dependant quantity is very easy to measure.
2. Large change in measured quantity (thousands of ohms, compared to millivolt changes in thermocouples and diode junctions), meaning low demands are placed on the analogue circuitry.
3. Inexpensive.
4. Good stability.
5. Circuit can easily be designed to give very linear output, making the digital conversion to temperature straightforward.

With these considerations, an ND06P00103K NTC $10\,\mathrm{k\Omega}$ thermistor from AVX was selected, based on price and good electrical characteristics based on the application requirements.

An analogue signal processing circuit was required to process the output from the thermistor to a signal usable by the microcontroller analogue-to-digital converter (ADC). This required a signal in the range of $0 - 3.3\,\mathrm{V}$.

The thermistor was placed in a voltage divider circuit with a series $10\,\mathrm{k\Omega}$ resistor. This was selected to linearize the temperature curve. This combination gives a response linear to within $1\,\%$ from 0 to $40\,°\mathrm{C}$ and $3\,\%$ between -10 and $50\,°\mathrm{C}$. This voltage divider operates from a 3.3V rail, ensuring that the input to the next stage will always lie within its supply rails. At this supply level, the maximum power dissipation through the thermistor was calculated as $3.3\,\mathrm{mW}$ at $50\,°\mathrm{C}$. This is well below the maximum dissipation for the thermistor of $0.71\,\mathrm{mW}$, so will not have a noticeable affect on the temperature measurement.

The output from the voltage divider is input to an op-amp non-inverting amplifier circuit. The voltage divider has relatively high output resistance, so use of a non-inverting op-amp configuration ensures a very large input resistance ($\sim 10^{13}\,\Omega$), avoiding loading on the measurement circuit.

The non-inverting amplifier is slightly non-standard due to a non-zero reference voltage on the inverting terminal. This is selected to give the same output as the thermistor setup at the low end of its response, which was selected as $-20\,°\mathrm{C}$ as this will lie well below the temperature a package would expect to experience in transit and is at the limit of the linear response. This reference voltage was found to be $0.79\,\mathrm{V}$, so is generated using a series combination of $22\,\mathrm{k\Omega}$ and two $10\,\mathrm{k\Omega}$ resistors, giving a very close voltage match with standard resistor values.

The gain was then calculated to produce a $3.3\,\mathrm{V}$ output swing over the input range to be received, and the gain required was found to be 1.47. This was achieved in the non-inverting amplifier using standard resistor values of 56 and $120\,\mathrm{k\Omega}$.

The output swing was chosen to span the full supply level of $0 - 3.3\,\mathrm{V}$ as the op-amp used provides very good rail-to-rail operation, and the extremes of the working temperature range are both unlikely to experienced and reaching the limits of the linear response from the thermistor. For the package tracking application, it is not critical to know how far beyond these limits the temperature has reached, so a simple saturating output is sufficient. This allows the ADC to operate with maximum precision.

The circuit diagram for the temperature monitoring circuit is shown in appendix A, figure 5.

A filter could easily have been implemented in the op-amp to avoid noise, since the signal is of very low frequency. However, the output was found to be very stable without filtering, so filtering was avoided for the sake of simplicity.

3.2 Software

It was suggested to use the National Instruments "LabWindows/CVI" package for developing the PC software and graphical user interface (gui), however it was decided to eschew this in favour of developing in C++ and the wxWidgets cross platform gui toolkit. The reasons for this were:

1. LabWindows is a highly specialist and proprietary development package. While it is simple to use and fast to setup, we found it unlikely that we would use it again in the future.

2. LabWindows is fairly limited in scope of what can be achieved, whereas wxWidgets and C++ allow pretty much anything to be achieved.

3. Our other project was using C++ and wxWidgets, so it was a simpler option to only learn one package, rather than one per project.

4. Past experience of coding in C++ could be utilised, whereas LabWindows relies on C which I have very little experience of.

There are three main components to the software development. The back end functionality required serial port communications and communications protocols with the microcontroller. The front end required a simple user interface to display the logged data and control the microcontroller functions.

The actual data transmission is handled by the serial port library.

The sending of packets to and from the serial port for the microcontroller are handled by the communications level, which provides an interface between the application and the actual data, so the user and user interface level do not depend on the communications system. This is advantageous as it means the top level application is independent of the communication medium, and could be converted to use USB or wifi without modifying the user application. It also simplifies the functions required in the user application as they do not directly handle the raw data, but instead use more user/programmer friendly data structures.

On top of the communications layer is the application and user interface layer. These allow easy and logical viewing and understanding of commands to be sent and data received by the computer.

This hierarchical structure is shown in figure 2. The layers map very roughly onto the Open Systems Interconnect (OSI) reference model. The user interface performs the duties of layer 7 (application). The communications layer is similar to layer 6 (presentation), while the rs232 library spans levels 4 and 5 (transport and session layers, respectively). The lower layers are defined by the RS232 protocol.

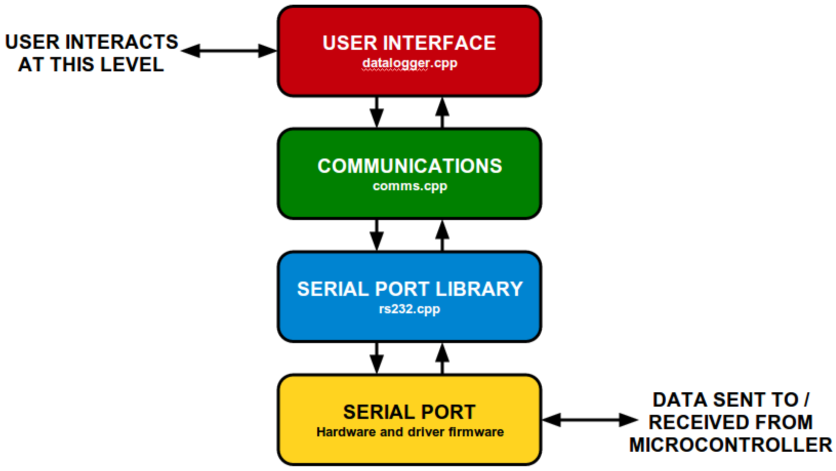

Figure 2: Communications stack showing interaction of each layer.

3.2.1 Serial Port Communications

The files `rs232.cpp` and `rs232.h` contain functions for sending and receiving data from the serial port. These were found online [5], released under the GNU General Public Licence. (See section 4 for more details.)

The important functions provided by this software is are:

`RS_232_OpenComport()` This sets up the connection with the serial port to be used, setting up a two way communications link to the microcontroller.

`RS_232_CloseComport()` Closes the connection to the serial port at the end of the operation.

`RS_232_PollComport()` Reads data that has been sent by the microcontroller to the computer.

`RS_232_SendBuf()` Writes to the serial port to send commands to the microcontroller.

These functions were all provided directly by the software found online, which greatly simplified the serial port reading.

3.2.2 Communications

The functions provided in the RS_232 library are very low level, and perform sending and receiving of individual bytes. For the user interface, it is useful to have a high level data structure, rather than dealing with individual bytes of data. The files `comms.cpp` and `comms.h` provide the means to do this. The functions here call the functions in `rs232.cpp`. It converts the user commands into the required bytes to send to the microcontroller, and organises the returned data into a `vector` data structure.

This level provides functions to be called by the data logger application:

`RS232_Init()` Opens the required comport to establish a link with the microcontroller.

`RS232_Close()` Closes the opened comport.

`send_command()` Sends a command to the microcontroller by writing to the serial port buffer (uses the `SendBuf` function).

`read_eeprom_data()` Reads data stored in the EEPROM into a `vector`.

`get_Readings()` Receives the data sent by the microcontroller in linked mode and stores it in a vector.

3.2.3 User Interface

The top level of the PC system is the user interface, which allows the user to interact with the system in an intuitive and non-technical way. The user interface[1] is shown in figure 3.[2]

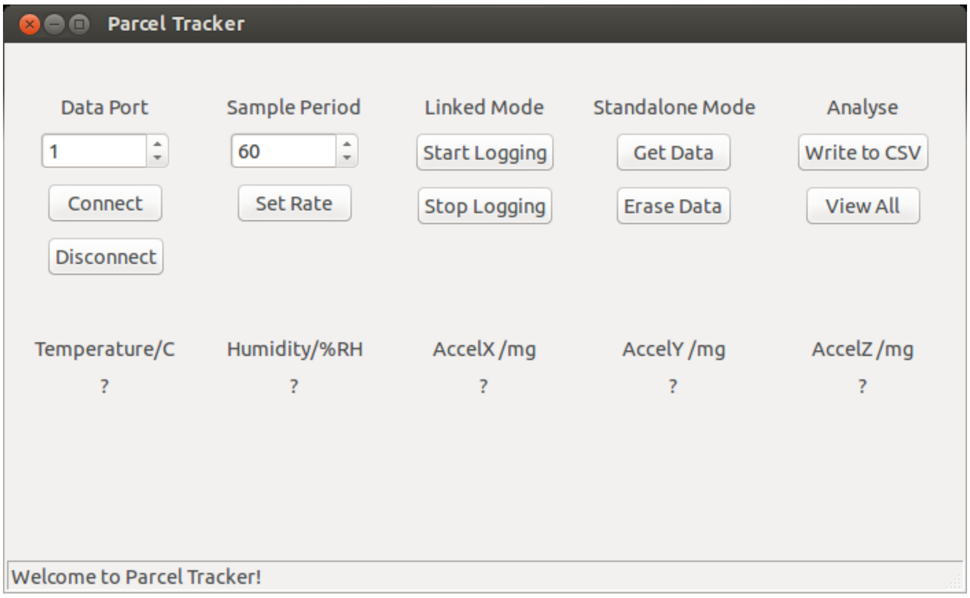

Figure 3: Screenshot of user interface running under Ubuntu Linux.

The user can click buttons and select values from a drop down menu to operate the data logger. The output from the monitoring is displayed directly on the screen during operation in the linked mode. After uploading the data from standalone mode operation, or once a logging session has been completed in linked mode, the data may be exported to a CSV file.

Additionally, the user may set the sample rate of the data logger from the gui. This is stored on board the data logger, so may be used to set the standalone operation rate as well as linked mode.

Development was started on a feature to plot the data that had been logged (accessed with the "View All" button). This utilised a python plotting script, called by the user interface. This system worked on Linux based systems, but was not modified to run under Windows. This would be a useful feature, however it was not implemented to allow development of more critical features. As an interim replacement, the user may open a saved CSV file in their favourite application and plot it (very easy in Microsoft Excel, Matlab and other data analysis packages).

[1]Since this screenshot was taken, additional features have been added to allow setting of a threshold temperature and humidity. If these limits are crossed, a warning is displayed.

[2]It may be noted that James' report shows a user interface which looks quite different, this is due to running it on Windows 8, whereas I used Ubuntu. This shows the cross-platform capabilities of the software.

3.3 Communications

It was required to define a communications protocol. This would ensure both the microcontroller and the PC knew what each byte and bit of a transmitted data packet meant. It was decided to avoid start-bits, stop-bits and check sums in the initial implementation, though room was left for these to be added later if necessary. In general, these weren't required as the PC could operate at sufficiently higher speed that it faced no problems in reading the data.

In linked mode operation, with the microcontroller plugged in to the computer, the microcontroller was to send one packet of data for each sampling period. Each packet was to contain 10 bytes, with the first 2 bytes containing temperature data, in upper and lower bytes, then a byte of humidity data. The remaining 7 bytes were reserved for accelerometer data, as we were initially unsure exactly what information we could send from the accelerometer. These bytes could also be used for check sums to ensure the data was received correctly. Figure 4 shows the layout of a data packet.

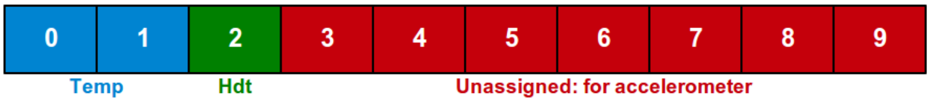

Figure 4: Data packet layout, showing positions of bytes and their contents.

When in standalone mode, the data logger would store all recorded data in the EEPROM. The first 16 bytes of EEPROM data were to be reserved for configuration information: the first two would store the length of the recorded data from the last log, while the third byte would store the current sample rate. The rest of the EEPROM (32,000 bytes) would be divided into 10 byte segments, each to contain one data packet, as specified for the standalone mode.

For sending commands from the PC to the microcontroller, it was decided that all commands sent should be 10 bytes long. Each would contain an ASCII character defining the desired command, as shown in table 1.

Character	Command
L	Begin logging in linked mode.
S	Stop logging (linked mode).
U	Upload data from EEPROM (standalone mode data).
E	Erase EEPROM contents (except 16 bytes of configuration data).
R [num]	Set sample period, in seconds (defined in num).

Table 1: Commands sent from PC to data logger.

The communications protocols defined worked well for the application and did not need changed after definition. The accelerometer data was loaded with the maximum acceleration in each dimension during the sample period.

4 Problems Encountered and Technical Solutions

In this section, some of the problems encountered during the problem will be discussed, along with the solutions we used to overcome these problems. I will focus on the problems in the areas I was focusing on but also give a mention to other problems in James's parts, see James's report for a fuller discussion of these.

4.1 RS232 Library

The decision to use C++ and wxWidgets instead of the LabWindows environment initially cased a problem of being unable to communicate with the RS232 port, since this was a feature built in to the LabWindows software.

Given the short time allowed for the project, it was not a viable option to write an RS232 library from scratch (interesting and challenging though this would doubtless have been!). A free and open-source library was found online[5] to provide this functionality and was found to work well.

Once this was found, it was relatively straightforward to write the communications layer functions. These were initially based on the similar functions in the supplied code, but became more heavily modified as features were added specific to the data logger, such as reading a packet, reading from EEPROM and writing according to the protocol we defined.

4.2 Display of Data

Displaying the data in the gui in a sensible and easily interpretable way turned out to be a significant problem. The original idea had been to create graphs on the gui using OpenGL as a graphics package. However, once the basic user interface had been created and the communication system implemented, there was insufficient time left to do this in time for the demonstration. There was already in place a function for outputting the stored data to a comma separated value (csv) file[3], so it was decided to utilise this in an outside program to do the plotting.

Some Python scripts were written to achieve this, which worked well under Linux. There were issues in running the same scripts on Windows, and since time was short, it was decided to focus efforts in other areas, such as the temperature and humidity boundaries on the gui display. The current recommended way to view the plotted data is to open the csv file in Excel and plot it from there.

4.3 Linked mode streaming

running logging in background while allowing gui functionality.

4.4 Problems with Sensors and Circuits

accelerometer, humidity sensor.

5 Test Procedures

6 Conclusions and Next Steps

A Source Code and Circuit Diagrams

[3] A common and widely used data storage format.

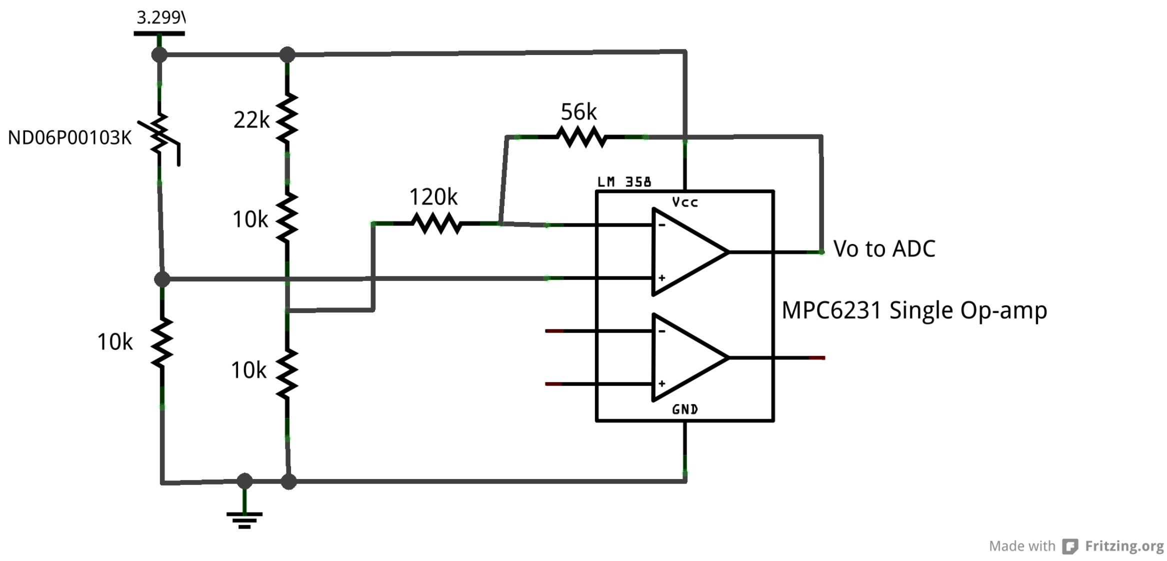

Figure 5: Temperature monitoring circuit.

B Marketing Datasheet

C First Interim Report

References

[1] http://www.cambridgeconsultants.com/news/pr/release/116/en. Feb 2013. Accessed on 3/6/13.

[2] Oxford Economic Forecasting. The impact of the express delivery industry on the global economy. http://eiciindia.org/frontsite/An%20Oxford%20Economic%20Forecasting%20-%20The%20impact%20of%20the%20Express%20Delivery%20Industry%20on%20the%20global%20economy.pdf, 2005. Accessed on 3/6/13.

[3] P. Horowitz and W. Hill. *The Art of Electronics*, chapter 15: measurements and signal processing, pages 992–993. Cambridge University Press, second edition, 1989.

[4] http://news.van.fedex.com/SenseAware. Nov 2012. Accessed on 3/6/13.

[5] Teunis van Beelen. Rs-232 for linux and windows. http://www.teuniz.net/RS-232/.