MCTX3420 2013
Exploding Cans Project
Software Team Report

Justin Kruger, 20767264 (Chapter 1)
Sam Moore, 20503628 (Editing/Referencing, Sections 2.1,2.2,2.3 Chapter 3),
Jeremy Tan, 20933708 (Sections 2.4,2.5,2.6)
Callum Schofield, 20947475 (Section 2.7)
James Rosher, 20939143 (Section 2.8)
Rowan Heinrich, 20939081 (Section 2.9)

THE UNIVERSITY OF
WESTERN AUSTRALIA
Achieve International Excellence

November 2013

Contents

1. Introduction and Approach

The following report describes the work of the software team on the MCTX3420 pressurised can project during Semester 2, 2013 at UWA. The report is intended to assist others in comprehending the decisions and processes involved, as well providing a tool for further development of the system. The report serves as a record of the planning, design, coding, testing and integration of the system, with specific reference to the development of the system software. Extensive documentation is also provided via a project wiki[1].

The MCTX3420 project aimed to build an experimental apparatus for measuring the behaviour of a container with pressure — in this case, testing how a drink can deformed as air pressure inside it increased. The desired result was a self-contained, safe, reliable, effective and easy-to-use system which could perform the desired experimental tasks, to be used by both students and wider industry professionals.

Unfortunately, the system is (as of 1st November 2013) still not complete; the hardware components have not been fully tested and integrated with the software, despite extensive work by all students. However, the project is very close to completion. The software can interact in the desired manner with the hardware components, and fulfils the majority of the required functionality. With some further testing and development using the final hardware, the software could easily be implemented — and the report has been written with this in mind, allowing another group in the future to build upon the project software.

The report begins with an overview of the whole system and the design of the software component. Each subsection then focuses on a specific aspect of the software, going into detail about its design, development, functionality, testing, and integration. Following this, there are sections focusing on the administrative aspects of the project, including teamwork, the general development process, and costs. The report concludes with some documentation of the software and recommendations for future development.

1.1 System Overview

To aid understanding of the context of the software project, a brief overview of the system as a whole is presented below. Essentially, the MCTX3420 project apparatus is designed to test the behaviour of a pressure vessel as air pressure inside it is gradually increased. A very basic system diagram showing the main components is shown in Figure 1.1, with control components in red, electronics in green, sensors in purple, pneumatics in blue, and experimental targets in orange.

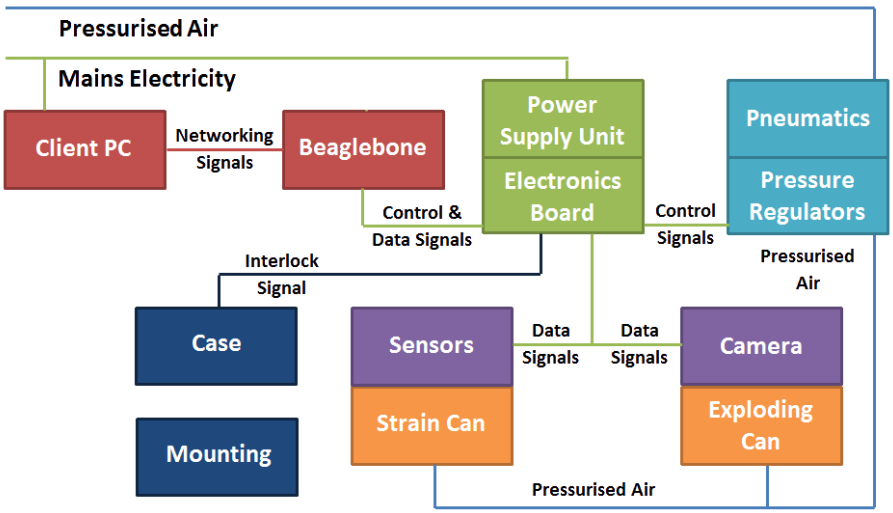

Figure 1.1: Block diagram of the physical system

1.1.1 Experimental Procedure

The general experimental procedure is to increase the pressure inside a pair of pressure vessels (in this case, drink cans), measuring one's deformation behaviour and measuring the other to failure point. The user does this by logging into a web browser interface, starting a new experiment, and increasing system pressure in the desired fashion.

As pressure is increased, the web browser passes this instruction to the system controller, which manipulates the pneumatic pressure regulators to input correct pressure to the measured can. While doing this, the system controller also reads from a collection of sensors and returns this data to the web browser (strain, pressure, dilatometer deformation, visual images). The vessel's deformation with pressure can then be characterised.

This continues until the desired final pressure is reached. Then, pressure in the failure can may be increased further until that can reaches its failure point. The experiment then ends and the system is returned to room pressure. The user can view and download the resulting data.

1.1.2 Components

The main areas of the system are as follows:

- **Control:** The experiment is controlled through web browser interface from a client PC. This web interface connects to a server running on what is effectively a small, integrated PC — the "BeagleBone Black" — and this server directly controls the experiment hardware and reads the sensor data, providing a low-level interface to the system's electronics. The BeagleBone itself is situated inside the experiment case, and the client PC connects to the BeagleBone server through the local network.
- **Electronics:** Because the system features a large array of electronic components, these must be run through a central filtering and amplification system to ensure that correct voltages and currents are provided. There is a circuit board inside the case which performs this task. The board connects the BeagleBone, pneumatics, sensors, and power supply to facilitate system operation. System power is provided by a PSU connected to the mains.

- **Pneumatics:** The system's pneumatics feed the desired air pressure into the two pressure vessels being tested. Air is fed through a series of pipes from the laboratory's pressurised air supply; solenoid valves control on/off air flow through the system, while pressure is controlled via regulators. Exhaust valves are provided for venting the system. Pneumatics are controlled by the BeagleBone, with signals fed through the electronics board.
- **Sensors:** A suite of sensors is used to collect system data and measure the deformation of the pressure vessel. The sensors include strain gauges, pressure sensors, a microphone, a dilatometer/microscope, and a camera — these give a comprehensive set of data to match the can's deformation to the pressure level. Each sensor has a different output and must be conditioned by the central electronics board before its data is recorded by the BeagleBone.
- **Mounting and Case:** The mounting system for the cans uses a screw-in mechanism to achieve an airtight seal. This holds the can in place so that pressure can be fed into it through the base of the mount. The system case holds all of the components in a sealed protective compartment, which ensures that the system will be safe in the event of failure and physically separates the various systems. The case also features an interlock switch that prevents any operation of the system if the lid is not fastened.

The system software essentially is defined by the "control" component: allowing a user to control the experiment hardware. To do this, the software must successfully interface with all of the system areas above so that the desired experiment can be run.

1.2 Development Process

The development process is outlined below. Each part of the software followed the same general process, which is discussed in more detail for each section later in the report.

1.2.1 Planning and Design

First, the actual software task to be completed is identified; this is organised with group input. The software component is then designed according to the requirements. Parameters and features are chosen based on the project guidelines and how the component interacts with other software.

1.2.2 Coding

Each section is then actually written. Most of the initial work is done individually (for consistency) and completed in between meetings. At group meetings the code is presented, and may be edited by other team members to fix issues, increase efficiency, and integrate it with other code sections.

Extremely important to development was the use of the Git system[2, 3] and GitHub website[2]. GitHub is specially designed for software use and is essentially a web-based hosting service for development projects, which uses the Git revision control system. It allows all team members to contribute to the same project by working on their own local "forks", and then "merging" their changes back into the main branch of software[4].

The Git system ensures that work by different team members is tracked[5], that work fits together consistently, and that other work is not accidentally overwritten or changed (important when dealing with large amounts of code). Git also features a notifications and issue tracking system with email alerts whenever a change is made. The basic GitHub process is as follows:

1. Create an individual "fork" of the software, separate from the main branch.
2. Modify this fork on a local machine with proposed changes or additions. This fork is also updated regularly with any changes that were made in the main branch.

3. When work is complete, create a "pull request" to merge local changes back into the main code base.

4. The pull request can be reviewed by other team members; if everything fits, the request is accepted and the local changes become part of the main code.

In this way, GitHub automates the more tedious aspects of code management.

Another important aspect of the coding process is coding style. Throughout the project, all code that was written adhered to the same style to make it consistent and easier to read. One aspect of styling, for example, is use of capitals when defining function names (for example, `Actuator_Init`), variable names (`g_num_actuators`), or definitions of constants (`ACTUATORS_MAX`), to make it immediately clear whether something is a function, variable or constant. Other aspects include use of indentation, the ordering of functions, and frequent use of comments. Essentially, styling is used to ensure the code is consistent, easy to follow, and can therefore be worked on by multiple people.

Coding style is also important when following general code standards. The C language features many standards and style guidelines which were also adhered to, to make the code readable by wider industry professionals. Some examples of this include beginning global variables with `g_` and correct use of brackets as separators[6]. All efforts were made to follow common C and HTML code standards. The use of a common coding style and standards will hopefully make the project software easily expandable by others in the future.

Code was also expected to adhere to safety standards. In the first weeks of the project, a document[7] was created that outlined all aspects of software safety - both for the software design itself, and ensuring that the system was still safe if the software failed. The results of this are explained further later in the report, with one example being the server's "sanity check" functions.

1.2.3 Testing

Once the software section is relatively complete, it can be tested with the larger code base. This was generally done through writing specific test functions. Because the operating system on the BeagleBone (GNU/Linux) is widely available for commercial PCs and laptops, software development and testing could occur without needing to wait for a BeagleBone to become available. Code was also tested on the BeagleBone itself where possible to ensure correct operation. One example is for the sensors software - initially, functions were written that simulated sensors, so it could be tested if data was read correctly. These functions were rewritten for use with actual hardware as the specifics became known later in the project.

1.2.4 Collaboration

After the testing process is satisfied, the final code can be committed to the system. This requires input from the other project teams. If there is any feedback or the requirements change in the future, the code can be edited through the above process.

1.3 Team Collaboration

Collaboration between members of the software group was extremely important throughout the project. Members were often individually responsible for different areas of software — or, alternately, were simultaneously rewriting different sections of the same code — so it was essential to make sure that all parts were compatible, as well as completed on schedule. Communication between the software group and other project groups was similarly vital, to ensure that all work contributed to the project's end goals.

1.3.1 Communication

The primary time for collaboration was during the team's weekly meetings. Meetings occurred at 2pm-4pm on the Monday of every week, and were generally attended by all group members. While most work was expected to be done outside this time, the meetings were valuable for planning and scheduling purposes, for tackling problems and making design decisions as a group. Team members were able to work together in the meetings to complete certain tasks much more effectively. Importantly, at the end of each meeting, a report of the work done during the prior week and a list of tasks to do the following week was produced, giving the project a continuous, clear direction.

GitHub was used as the group's repository for software work. The usefulness of GitHub was explained previously in Section 1.2, but essentially, it is a very effective tool for managing and synchronising a large, multi-person software project. GitHub also features a notifications and issue-tracking system, which was useful for keeping track of tasks and immediately notifying team members of any changes.

Outside of meetings, email was the main form of communication. Email threads exist for all of the project's main areas, discussing any ideas, changes or explanations. Email was also used for announcements and to organise additional meetings. For less formal communication, the software group created their own IRC channel. This was essentially a chat channel that could be used to discuss any aspect of the project and for communication about current work.

1.3.2 Scheduling

At the beginning of the project, an overall software schedule was created, outlining the main tasks to be completed and their target dates. While this was useful for planning purposes and creating an overall impression of the task, it became less relevant as the semester continued. The nature of the software team's work meant that it was often changing from week to week; varying hardware requirements from other teams, unexpected issues and some nebulous project guidelines led to frequent schedule modifications. For instance: use of the BeagleBone turned out to be a significant time-sink, requiring a lot of troubleshooting due to lack of documentation; and a sophisticated login system was not mentioned until late in the project, so resources had to be diverted to implement this. Essentially, while the software group did attempt to keep an overall schedule, this was only useful in planning stages due to the changing priorities of tasks.

Far more useful was the weekly scheduling system. As mentioned in the "Communication" section 1.3.1, a weekly task list was created on each Monday, giving the team a clear direction. This suited the flexibility of the software well; tasks could be shuffled and re-prioritised easily and split between team members. It was still very important to keep the project's overall deadline in mind, and the weekly task lists could be used to do this by looking separately at the main areas of software (such as GUI design, sensors, and so on) and summarising the remaining work appropriately. Brief weekly reports also covered what had been completed so far, providing a further measure of progress.

The group also elected a "meeting convener" to assist with organisation (Samuel Moore). The meeting convener was responsible for organising group meetings week-to-week and coordinating group communication. A single elected convener made this process as efficient as possible.

1.3.3 Group Participation

The nature of software development means that it tends to be very specialised — extensive knowledge of coding is required to be effective, which is difficult to learn in a short time frame. The members of the software team all had varying levels of experience, and therefore could not contribute equally to all areas of the project. Some team members had done very little coding before (outside of introductory units at university) which made it difficult for them to contribute in some areas, while others had the extensive knowledge required.

However, different team members had skills in other areas besides coding, and these skills were allocated to ensure that all members could contribute effectively. For instance, as some people worked on the server code, others worked on the visual GUI design; it made sense for the people who were most efficient with coding to work on those elements while others performed different tasks. Even though the software project was principally coding, there were many supplementary development tasks — writing documentation, hardware testing, et cetera — that were involved. Some areas of the software, such as the BeagleBone interfacing, were new to all team members and were worked on by everyone.

On the whole, group participation was good. Team members regularly attended meetings, did the expected (often more-than-expected) work, and had a good understanding of the project. While all team members contributed significantly, some did stand out — in this case Samuel Moore and Jeremy Tan, who performed a large portion of the vital development work. Without their input and prior experience, the project would not have been completed to such a high standard, and their extensive skills and dedication were vital to its success.

1.3.4 Inter-Team Communication

Communication between the various project teams was also essential: the software had to be able to interact with nearly all aspects of the hardware via the BeagleBone system controller. A weekly Tuesday meeting was therefore set up specifically for inter-team communication, so information could be exchanged between project groups. For the software team most communication was with the electronics, sensors and pneumatics teams, as these three hardware areas are all directly controlled by the software. The fact that the software can interact with these systems should be evidence that communication was relatively effective. Many other meetings also occurred between the software group and others. Extensive time was spent with the electronics team, testing and setting up the BeagleBone with the appropriate inputs and outputs. Other meetings also occurred with the sensors team to select sensors and cameras that were compatible with the software. Practical sessions with the pneumatics, sensors and electronics teams also occurred, in which the software was tested with the hardware to ensure that both systems were operating correctly.

Email was used extensively for other communication. All members of the unit were involved in this, providing input on hardware designs or organising meeting times for testing, and though email was often less effective that face-to-face communication (other teams sometimes did not respond promptly) it was still useful tool. In addition, an MCTX3420 DropBox was set up as a common repository for any project-related files. This was updated often and proved to be a useful reference. The software team chose to keep their work on GitHub rather than DropBox, and the GitHub repository was made publicly accessible so that work could be shared.

1.3.5 Individual Contributions

Software project tasks were divided up between team members, and in this report, each team member has generally been the writer of the sections they actually worked upon. Throughout the project, team members had clear areas of responsibility, and their work can also be followed through the GitHub repository (which allows tracking of individual contributions to the code base). Below is a rough summary of individual areas of interest:

Team Member	**Development**
Samuel Moore	Server coding, BeagleBone interface, GUI implementation, hardware testing
Jeremy Tan	Server coding, BeagleBone interface, GUI implementation, hardware testing
Callum Schofield	Image processing, BeagleBone interface, hardware testing
James Rosher	Overall GUI design, GUI implementation
Justin Kruger	BeagleBone interface, GUI implementation, documentation
Rowan Heinrich	GUI implementation, hardware testing

It should also be noted that team members often helped each other with designing, problem solving and testing, so members did end up contributing in some way to most areas of the software.

Server coding tasks included the threading system, data handling, sensors/actuators control, authentication, server/client communication, HTTP(S) use, FastCGI, AJAX and the server API, which were split mainly by Sam and Jeremy (with significant overlap). BeagleBone interfacing included hardware access, pin control, networking and testing, and involved most members of the team. Sam, Jeremy and Justin focused on pin control, Jeremy and Callum investigated webcam use, and Rowan performed additional testing. GUI design involved the visual design elements, HTML/CSS webpage coding and Javascript functionality. James was primarily in charge of the GUI design, functionality and implementation, with assistance and alternate designs provided by Jeremy. Other team members were responsible for individual GUI sections, including Sam (graphs), Justin (help and data) and Rowan (widgets). Other tasks included image processing with OpenCV (Callum) and project documentation and safety (Justin).

1.3.6 Cost Estimation

The vast majority of the cost of the software team's contribution is in man-hours rather than hardware. The only hardware specifically purchased by software was a BeagleBone Black; all other hardware was part of electronics. Some hardware used for testing was temporarily donated by team members, and has been included here only for completeness.

Item	**Cost**
BeagleBone Black	$45
LinkSys Router (testing)	$50
Logitech Webcam (testing)	$25
Ethernet and other cabling (testing)	$10
Total	$130

In regards to the time spent, it is difficult to get an accurate record. At least three hours per week were spent in weekly meetings, and by consulting the team's technical diaries, it is estimated that team members spent an average of ten hours per week working on the project.

Approximate time per week (individual)	10 hours
Team size	6 people
Approximate time per week (team)	60 hours
Project Duration	13 weeks
Total time spent	780 hours
Hourly rate	$150 / hour
Total cost	$117,000 (+$130 for hardware)

This is a large amount at first glance, though it must be remembered that this was a complex software development project with many interacting parts. There were some inefficiencies which did unfortunately add to cost (such as the BeagleBone's lack of documentation) and these could hopefully avoided in the future. Given the final result, however, the cost appears reasonable.

The GitHub repository was also run through an online cost estimator[8], which resulted in a similar number of $100,000. The estimator takes into account the number of developers, time of development, and amount of code produced.

2. Design and Implementation

Figures 2.1 and 2.2shows the earliest high level design of the software for the system created in the first and last week of the project. In the early stages the options were kept open for specific implementation details. The early design essentially required software to be written for three devices; a client computer (GUI), an experiment server (control over access to the system, interface to the GUI, image processing) and an embedded device (controlling experiment hardware).

As the revised diagram in Figure 2.2 shows, to remove an extra layer of complexity it was decided to use a single device (the BeagleBone Black) to play the role of both the experiment server and the embedded device. From a software perspective, this eliminated the need for an entire layer of communication and synchronization. From a hardware perspective, use of the BeagleBone black instead of a Raspberry Pi removed the need to design or source analogue to digital conversion modules.

Another major design change which occurred quite early in the project is the switch from using multiple processes to running a single multithreaded process on the server. After performing some rudimentary testing (see Section 2.5.1) it became clear that a system of separate programs would be difficult to implement and maintain. Threads are similar to processes but are able to directly share memory, with the result that much less synchronisation is required in order to transfer information.

Note on filenames: In the following, files and directories related to the server are located in the server directory, files related to the (currently used) GUI are in testing/MCTXWeb, and files created for testing purposes are located in testing.

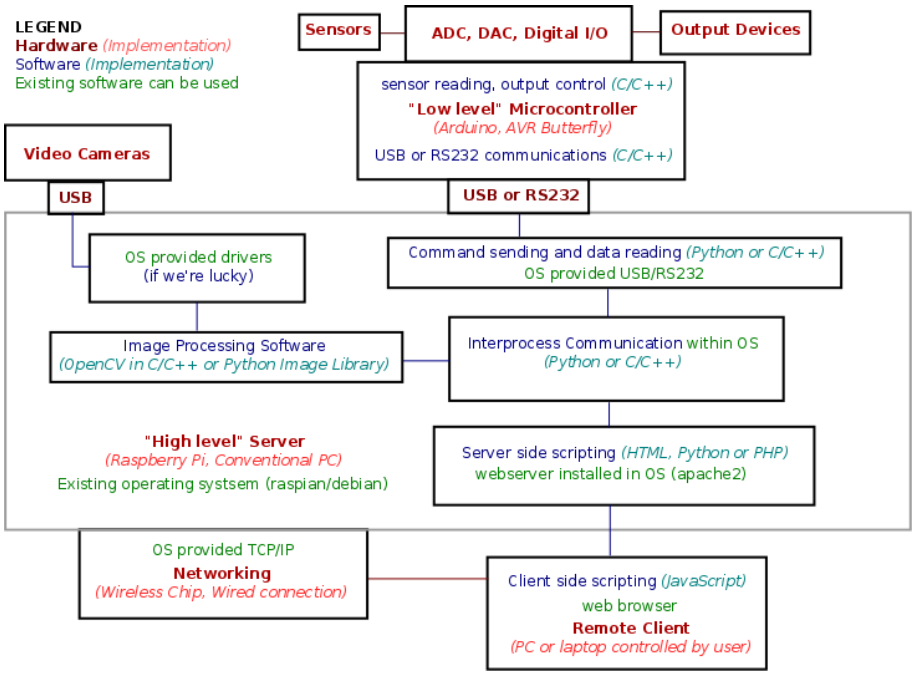

Figure 2.1: Block Diagram from Week 1 of the Project

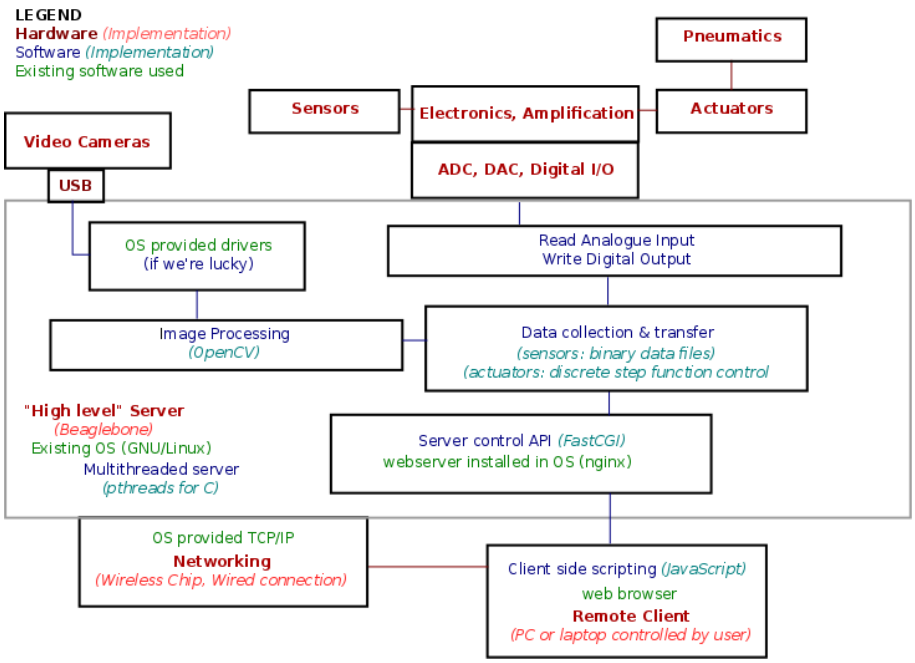

Figure 2.2: Block Diagram from Week 14 of the Project

2.1 Server Program

2.1.1 Threads and Sampling Rates

The Server Program runs as a multithreaded process under a POSIX compliant GNU/Linux operating system[1]. Each thread runs in parallel and is dedicated to a particular task; the three types of threads we have implemented are:

1. Main Thread (Section 2.1.2) - Starts all other threads, accepts and responds to HTTP requests passed to the program by the HTTP server in the `FastCGI_Loop` function (also see Section 2.4)
2. Sensor Thread (Section 2.1.3) - Each sensor in the system is monitored by an individual thread running the `Sensor_Loop` function.
3. Actuator Thread (Section 2.1.4) - Each actuator in the system is controlled by an individual thread running the `Actuator_Loop` function.

In reality, threads do not run simultaneously; the operating system is responsible for sharing execution time between threads in the same way as it shares execution times between processes. Because the Linux kernel is not deterministic, it is not possible to predict when a given thread is actually running. This renders it impossible to maintain a consistent sampling rate, and necessitates the use of time stamps whenever a data point is recorded.

Figure 2.3 shows a distribution of times[2] between samples for a test sensor with the software sampling as fast as possible. Note the logarithmic t axis. Although context switching clearly causes the sample rate to vary (green), the actual process of reading an ADC (red) using `ADC_Read` (*bbb_pin.c*) is by far the greatest source of variation.

[1] Tested on Debian and Ubuntu

[2] The clock speed of the BeagleBone is around 1GHz[9], which is fast enough to neglect the fact that recording the timestamp takes several CPU cycles.

It was not possible to obtain a real time Linux kernel for the BeagleBone. In theory, real time variants of the Linux kernel improve the reliability of sampling rates. However, testing on an amd64 laptop showed very little difference in the sampling time distribution when the real time Linux kernel was used.

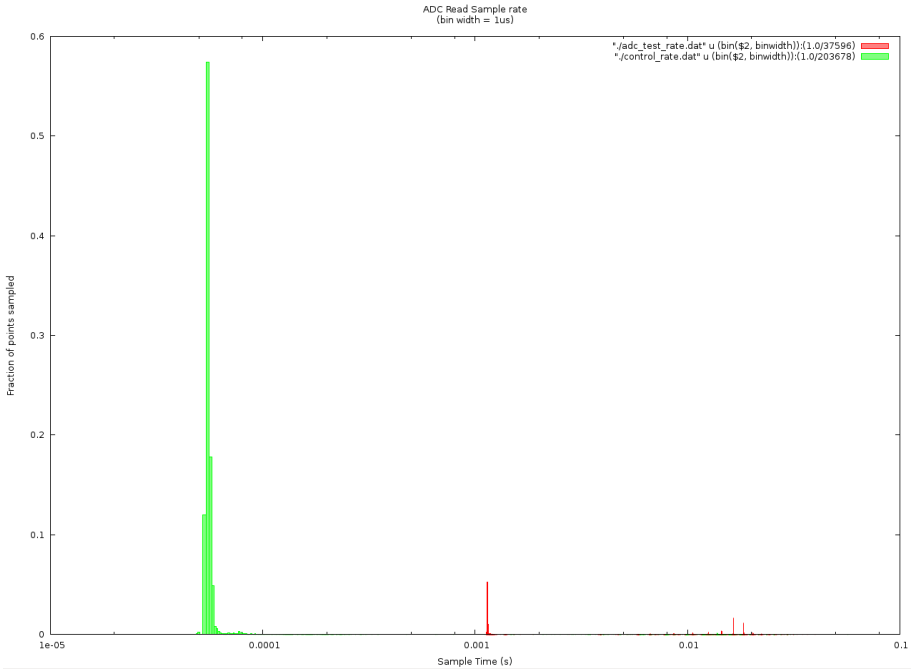

Figure 2.3: Sample Rate Histogram obtained from timestamps with a single test sensor enabled

2.1.2 Main Thread

The main thread of the process is responsible for transferring data between the server and the client through the Hypertext Transmission Protocol (HTTP). A library called FastCGI is used to interface with an existing webserver called nginx[10]. This configuration and the format of data transferred between the GUI and the server is discussed in more detail Section 2.4.

Essentially, the main thread of the process responds to HTTP requests. The GUI is designed to send requests periodically (e.g.: to update a graph) or when a user action is taken (e.g.: changing the pressure setting). When this is received, the main thread parses the request, the requested action is performed, and a response is sent. The GUI is then responsible for updating its appearance or alerting the user based on this response. Figure 2.8 in Section 2.4.3 gives an overview of this process.

2.1.3 Sensor Threads

Figure 2.4 shows a flow chart for the thread controlling an individual sensor. This process is implemented by `Sensor_Loop` and associated helper functions.

All sensors are treated as returning a single floating point number when read. A `DataPoint` consists of a time stamp and the sensor value. `DataPoints` are continuously saved to a binary file as long as the experiment is in process. An appropriate HTTP request (Section 2.4.3) will cause the main thread of the server program to respond with `DataPoints` read back from the file. By using independent threads for reading data and transferring it to the GUI, the system does not rely on maintaining a consistent and synchronised network connection. This means that one the experiment is started with the desired parameters, a user can safely close the GUI or even shutdown their computer without impacting on the operation of the experiment.

As Figure 2.4 indicates, the processes of actually controlling sensor hardware has been abstracted out of the control loop. A `Sensor` structure is defined in `sensor.h` to represent a single sensor. When this structure is initialised, function pointers must be provided; these functions can then be called by `Sensor_Loop` as needed. All functions related to control over specific sensor hardware can be found in the files within the `sensors` sub directory.

Earlier versions of the software instead used a `switch` statement based on the `Sensor`'s id number to determine how to obtain the sensor value. This was found to be difficult to maintain as the number and types of sensors supported by the software were increased.

2.1.4 Actuator Threads

Actuators are controlled by threads in a similar way to sensors. Figure 2.5 shows a flow chart for these threads. This is implemented in `Actuator_Loop`. Control over real hardware is separated from the main logic in the same way as sensors (relevant files are in the `actuators` sub directory). The use of threads to control actuators gives similar advantages in terms of eliminating the need to synchronise the GUI and server software.

The actuator thread has been designed for flexibility in how exactly an actuator is controlled. Rather than specifying a single value, the main thread initialises a structure that determines the behaviour of the actuator over a period of time. The current structure represents a simple set of discrete linear changes in the actuator value. This means that a user does not need to specify every single value for the actuator. The Actuator thread stores a value every time the actuator is changed which can be requested in a similar way to sensor data.

2.1.5 Data Storage and Retrieval

Each sensor or actuator thread stores data points in a separate binary file identified by the name of the device. When the main thread receives an appropriate HTTP request, it will read data back from the binary file. To allow for selection of a range of data points from the file, a binary search has been implemented. Functions related to data storage and retrieval are located in the *data.h* and *data.c* source files.

Several alternate means of data storage were considered for this project. Binary files were chosen because of the significant performance benefit after testing, and the ease with which data can be read from any location in file and converted directly into values. A downside of using binary files is that the server software must always be running in order to convert the data into a human readable format.

2.1.6 Safety Mechanisms

Given the inexperienced nature of the software team, the limited development time, and the unclear specifications, it is not wise to trust safety aspects of the system to software alone. It should also be mentioned that the correct functioning of the system is reliant not only upon the software written during this project, but also the many libraries which are used, and the operating system under which it runs. We found during development that many of the mechanisms for controlling BeagleBone hardware are unreliable and have unresolved issues; see the project wiki pages[1] for more information. We attempted to incorporate safety mechanisms into the software wherever possible.

Sensors and Actuators should define an initialisation and cleanup function. For an actuator (e.g.: the pressure regulator), the cleanup function must set the actuator to a predefined safe value (in the case of pressure, atmospheric pressure) before it can be uninitialised. In the case of a software error or user defined emergency, the `Fatal` function can be called from any point in the software; this will lead to the cleanup functions of devices being called, which will in turn lead to the pressure being set to a safe value.

Sensors and Actuators are designed to include an optional `sanity` function which will check a reading or setting is safe respectively. These checks occur whenever a sensor value is read or an actuator is about to be set. In the case of a sensor reading failing the sanity check, `Fatal` is called immediately and the software shuts down the experiment. In the case of an actuator being set to an unsafe value the software will simply refuse to set the value.

It is recommended that the detection of signals (a mechanism in GNU/Linux by which a program can detect certain types of unexpected crashes) be investigated. This was attempted in early implementations; however difficulties were encountered because any thread can catch the signal and thus will not be able to execute its cleanup function, or in some cases, continue running after the rest of the program has stopped.

An alternative safety mechanism involves modification of the script that starts the server (*run.sh*). This script is already able to detect when the program exits, and it should be possible to further extend this script to react accordingly to different exit codes.

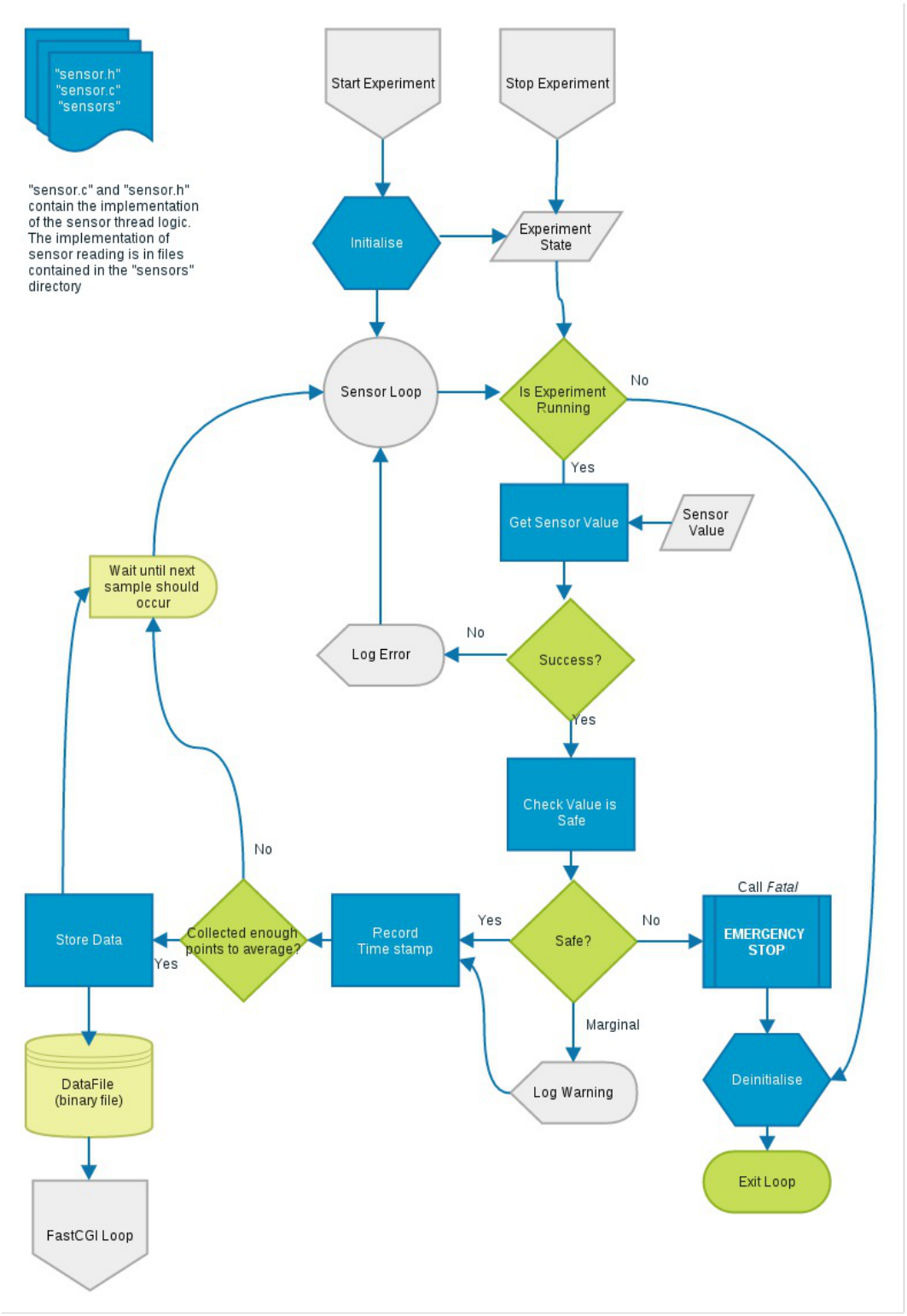

Figure 2.4: Flow chart for a sensor thread

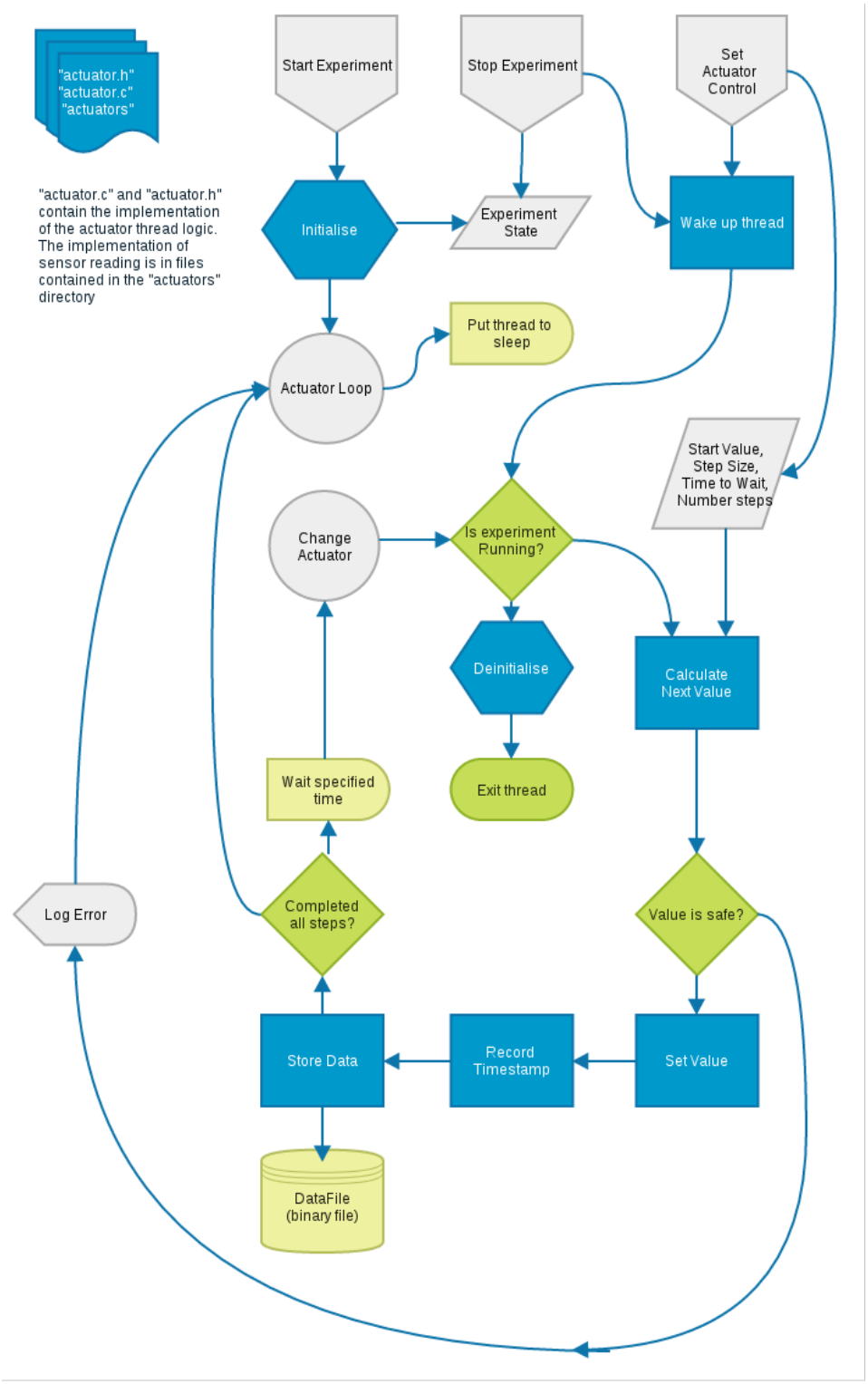

Figure 2.5: Flow chart for an actuator thread

2.2 Hardware Interfacing

Figure 2.6 shows the pin out diagram of the BeagleBone Black. There are many contradictory pin out diagrams available on the internet; this figure was initially created by the software team after trial and error testing with an oscilloscope to determine the correct location of each pin. Port labels correspond with those marked on the BeagleBone PCB. The choice of pin allocations was made by the electrical team after discussion with software when it became apparent that some pins could not be controlled reliably.

2.2.1 Sensors

Code to read sensor values is located in the *sensors* subdirectory. With the exception of the dilatometer (discussed in Section 2.7), all sensors used in this project produce an analogue output. After conditioning and signal processing, this arrives at an analogue input pin on the BeagleBone as a signal in the range $0 \rightarrow 1.8$V. The sensors currently controlled by the software are:

- **Strain Gauges** (x4)

  To simplify the amplifier electronics, a single ADC is used to read all strain gauges. GPIO pins are used to select the appropriate strain gauge output from a multiplexer. A mutex is used to ensure that no two strain gauges can be read simultaneously.

- **Pressure Sensors** (x3)

  There are two high range pressure sensors and a single low range pressure sensor; all three are read independently

- **Microphone** (x1)

  The microphone's purpose is to detect the explosion of a can. This sensor was given a low priority, but has been tested with a regular clicking tone and found to register spikes with the predicted frequency (1.5Hz).

- **Dilatometer** (x2) - See Section 2.7

Additional sensors can be added and enabled through use of the `Sensor_Add` function in `Sensor_Init` in the file *sensors.c*.

The function `Data_Calibrate` located in *data.c* can be used for interpolating calibration. The pressure sensors and microphone have been calibrated in collaboration with the Sensors Team; however only a small number of data points were taken and the calibration was not tested in detail. We would recommend a more detailed calibration of the sensors for future work.

2.2.2 Actuators

Code to set actuator values is located in the *actuators* subdirectory. The following actuators are (as of writing) controlled by the software and have been successfully tested in collaboration with the Electronics and Pneumatics teams. Additional actuators can be added and enabled through use of the `Actuator_Add` function in `Actuator_Init` in the file *actuators.c*.

Relay Controls

The electrical team employed three relays for control over digital devices. The relays are switched using the GPIO outputs of the BeagleBone Black.

- Can select - Chooses which can can be pressurised (0 for strain, 1 for explode)

- Can enable - Allows the can to be pressurised (0 for vent, 1 for enable)
- Main enable - Allows pressure to flow to the system (0 for vent, 1 for enable) and can be used for emergency venting

The use of a "can select" and "can enable" means that it is not a software problem to prevent both cans from simultaneously being pressurised. This both simplifies the software and avoids potential safety issues if the software were to fail.

PWM Outputs

A single PWM output is used to control a pressure regulator. The electrical team constructed an RC filter circuit which effectively averages the PWM signal to produce an almost constant analogue output. The period of the PWM is 2kHz. This actuator has been calibrated, which allows the user to input the pressure value in kPa rather than having to control the PWM duty cycle correctly.

Simplified Beagle Bone Black Pin out Diagram

P9

GND	1	2	GND
DC_3.3V	3	4	DC_3.3V
VDD_5V[1]	5	6	VDD_5V[1]
SYS_5V	7	8	SYS_5V
PWR_BUT	9	10	SYS_RESET
GPIO 30 *Case switch*[6]	11	12	GPIO 60
GPIO 31	13	14	EHRPWM1A (PWM3) [5]
GPIO 48	15	16	EHRPWM1B (PWM4) [5]
GPIO 5	17	18	GPIO 4
I2C	19	20	I2C
EHRPWM0B (PWM1) [5]	21	22	EHRPWM0A (PWM0) [5]
GPIO 49	23	24	GPIO 15
MCASP0	25	26	GPIO 14 *Can select*
GPIO 115 *Can enable*	27	28	ECAP2/MCASP0 (PWM7)[3]
MCASP0	29	30	GPIO 112 *Main enable*
MCASP0	31	32	VADC (1.8V)
AIN4[4] *Preg return*[6]	33	34	AGND
AIN6[4]	35	36	AIN5[4] *Low Pressure*[6]
AIN2[4] *Microphone*	37	38	AIN3[4] *High Pressure 1*
AIN0[4] *Strain Input*	39	40	AIN1[4] *High Pressure 0*
CLKOUT2 (?)	41	42	ECAP0 (PWM2)[3] *Preg set*
GND	43	44	GND
GND	45	46	GND

P8

DGND	1	2	DGND
EMMC2 (SD card)	3	4	EMMC2 (SD card)
EMMC2 (SD card)	5	6	EMMC2 (SD card)
GPIO 66	7	8	GPIO 67
GPIO 69	9	10	GPIO 68
GPIO 45 *Strain Mux Enable*	11	12	GPIO 44 *Strain0 (A) Select*
EHRPWM2B (PWM6) [5]	13	14	GPIO 26 *Strain1 (B) Select*
GPIO 47	15	16	GPIO 46 *Strain2 (C) Select*
GPIO 27	17	18	GPIO 65 *Strain3 (D) Select*
EHRPWM2A (PWM5) [5]	19	20	EMMC2 (SD card)
EMMC2 (SD card)	21	22	EMMC2 (SD card)
EMMC2 (SD card)	23	24	EMMC2 (SD card)
EMMC2 (SD card)	25	26	GPIO 61
GPIO 86[2]	27	28	GPIO 88[2]
GPIO 87[2]	29	30	GPIO 89[2]
GPIO 10[2]	31	32	GPIO 11[2]
GPIO 9[2]	33	34	GPIO 81[2]
GPIO 8[2]	35	36	GPIO 80[2]
GPIO 78[2]	37	38	GPIO 79[2]
GPIO 76[2]	39	40	GPIO 77[2]
GPIO 74[2]	41	42	GPIO 75[2]
GPIO 72[2]	43	44	GPIO 73[2]
GPIO 70[2]	45	46	GPIO 71[2]

[1]: VDD_5V is available only when DC jack is connected
[2]: These GPIO pins are unavailable if HDMI is connected and the HDMI capes are enabled
[3]: It is unknown if these pins are reserved or not (they seem to work)
[4]: **ADC** pins are **1.8V MAX (DO NOT EXCEED)**
[5]: PWM channels xA/xB *must* share the same period. To change the frequency if both are activated, the other has to be unexported.
[6]: These pins were specified but currently haven't been implemented or used in testing

All **GPIO** pins operate at **3.3V levels**. Current source/sinking capacities are limited - **4-6mA out and 8mA in (DO NOT EXCEED)**

Figure 2.6: Pinout Table

2.3 Authentication Mechanisms

The `Login_Handler` function (*login.c*) is called in the main thread when a HTTP request for authentication is received (see Section 1.3.1). This function checks the user's credentials and will give them access to the system if they are valid. Whilst we had originally planned to include only a single username and password, changing client requirements forced us to investigate many alternative authentication methods to cope with multiple users.

Several authentication methods are supported by the server; the method to use can be specified as an argument when the server is started.

1. **Unix style authentication**

   Unix like operating systems store a plain text file (/etc/shadow) of usernames and encrypted passwords[11]. To check a password is valid, it is encrypted and then compared to the stored encrypted password. The actual password is never stored anywhere. The /etc/shadow file must be maintained by shell commands run directly from the BeagleBone. Alternatively a web based system to upload a similar file may be created.

2. **Lightweight Directory Access Protocol (LDAP)**

   LDAP[12, 13] is a widely used data base for storing user information. A central server is required to maintain the LDAP database; programs running on the same network can query the server for authentication purposes.

   The UWA user management system (Pheme) employs an LDAP server for storing user information and passwords. The software has been designed so that it can interface with an LDAP server configured similarly to the server on UWA's network. Unfortunately we were unable to gain permission to query this server. However an alternative server could be setup to provide this authentication mechanism for our system.

3. **MySQL Database**

   MySQL[14] is a popular and free database system that is widely used in web applications. The ability to search for a user in a MySQL database and check their encrypted password was added late in the design as an alternative to LDAP. There are several existing online user management systems which interface with a MySQL database, and so it is feasible to employ one of these to maintain a list of users authorised to access the experiment. UserCake[15] is recommended, as it is both minimalistic and open source, so can be modified to suit future requirements. We have already begun integration of the UserCake system into the project, however a great deal of work is still required.

   MySQL and other databases are vulnerable to many different security issues which we did not have sufficient time to fully explore. Care should be taken to ensure that all these issues are addressed before deploying the system.

2.4 Server/Client Communication

This section describes the methods and processes used to communicate between the server and client. For this system, client-server interaction is achieved completely over the internet, via standard HTTP web requests with TLS encryption. In other words, it has been designed to interact with the client over the internet, **completely through a standard web browser** (Figure 2.7). No extra software should be required from the client. Detailed reasons for this choice are outlined in Section 2.5

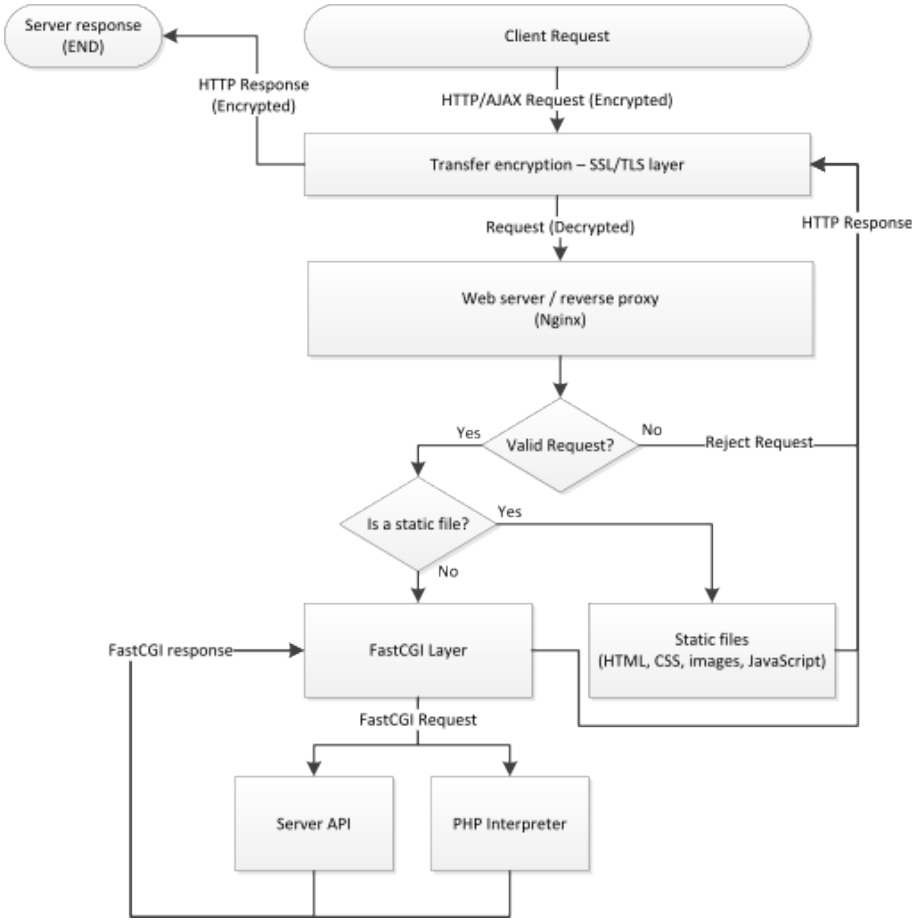

Figure 2.7: High level flow chart of a client request to server response

2.4.1 Web server

Web requests from a user have to be handled by a web server. For this project, the nginx[10] webserver has been used, and acts as the frontend of the remote interface for the system. As shown in Figure 2.7, all requests to the system from a remote client are passed through nginx, which then delegates the request to the required subsystem as necessary.

In particular, nginx has been configured to:

1. Use TLS encryption (HTTPS)
2. Forward all HTTP requests to HTTPS requests (force TLS encryption)
3. Display the full sever program logs if given "`/api/log`" as the address
4. Display the warning and error logs if given "`/api/errorlog`" as the address
5. Forward all other requests that start with "`/api/`" to the server program (FastCGI)
6. Process and display PHP files (via PHP-FPM) for UserCake
7. Try to display all other files like normal (static content; e.g the GUI)

Transport Layer Security (TLS) encryption, better known as SSL or HTTPS encryption has been enabled to ensure secure communications between the client and server. This is primarily important for when user credentials (username / password) are supplied, and prevents what is

called "man-in-the-middle" attacks. In other words, it prevents unauthorised persons from viewing such credentials as they are transmitted from the client to the server.

As also mentioned in Section 2.3 this system also runs a MySQL server for the user management system, UserCake. This kind of server setup is commonly referred to as a LAMP (Linux, Apache, MySQL, PHP) configuration[16], except in this case, nginx has been used in preference to the Apache web server.

Nginx was used as the web server because it is well established, lightweight and performance oriented. It also supports FastCGI by default, which is how nginx interfaces with the server program. Realistically, any well known web server would have sufficed, such as Apache or Lighttpd, given that this is not a large scale service.

2.4.2 FastCGI

Nginx has no issue serving static content — that is, just normal files to the user. Where dynamic content is required, as is the case for this system, another technology has to be used, which in this case is FastCGI.

FastCGI is the technology that interfaces the server program that has been written with the web server (nginx). As illustrated in Figure 2.7, there is a "FastCGI layer", which translates web requests from a user to something which the server program can understand, and vice versa for the response.

2.4.3 Server API - Making Requests

From the client side, the server interface is accessed through an Application Programming Interface (API). The API forms a contract between the client and server; by requesting a URL of a predetermined format, the response will also be of a predetermined format that the client can use.

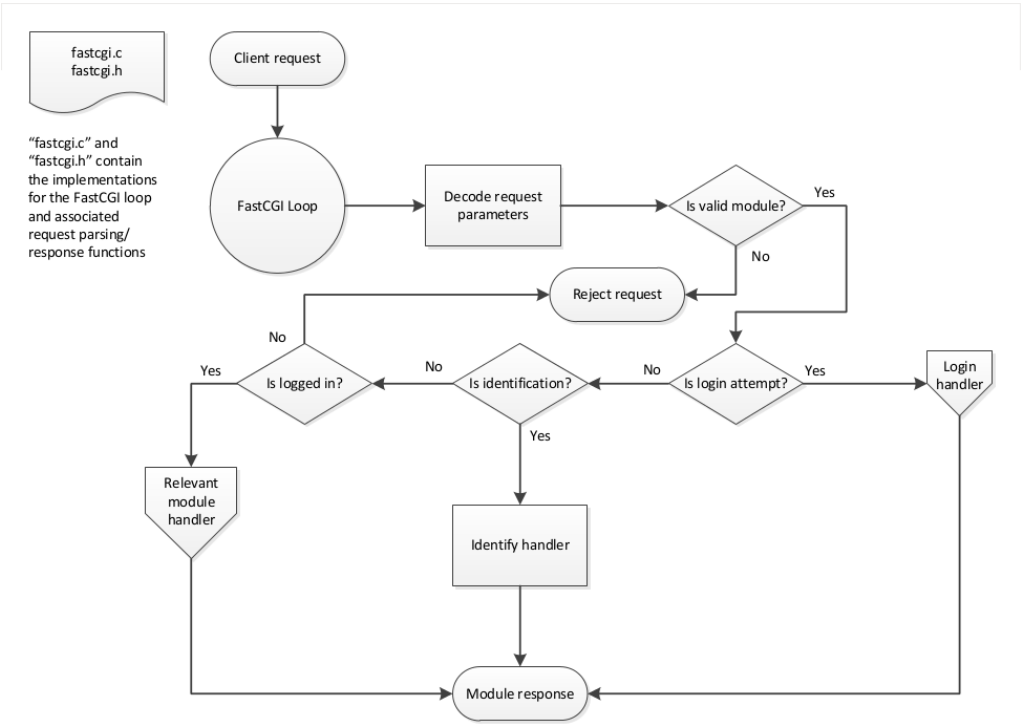

Figure 2.8: Flow chart of a client request being processed (within the server program). Relevant files are *fastcgi.c* and *fastcgi.h*.

In the case of the server API designed, requests are formatted as such:

`https://host/api/module?key1=value1&key2=value2...&keyN=valueN` (where `host` is replaced with the IP address or hostname of the server).

The API consists of modules accepting arguments (specified as key-value pairs), depending on what that module (Figure 2.9) does. For example, to query the API about basic information (running state, whether the user is logged in etc), the following query is used:

`https://host/api/identify`

The server will then respond with this information. In this case, the identify module does not require any arguments. However, it can accept two optional arguments, `sensors` and `actuators`, which makes it give extra information on the available sensors and actuators present. This makes the following queries possible:

- `https://host/api/identify?sensors=1`
- `https://host/api/identify?actuators=1`
- `https://host/api/identify?sensors=1&actuators=1`

These give information on the sensors, actuators, or both, respectively. For other modules some parameters may be required, and are not optional. This form of an API was chosen because it is simple to use, and extremely easy to debug, given that these requests can just be entered into any web browser to see the result. The request remains fairly human readable, which was another benefit when debugging the server code.

Keeping the API format simple also made it easier to write the code that parsed these requests. All API parsing and response logic lies in *fastcgi.c*. The framework in *fastcgi.c* parses a client request and delegates it to the relevant module handler. Once the module handler has sufficiently processed the request, it creates a response, using functions provided by *fastcgi.c* to do so.

This request handling code went through a number of iterations before the final solution was reached. Changes were made primarily as the number of modules grew, and as the code was used more.

One of the greatest changes to request handling was with regards to how parameters were parsed. Given a request of: `http://host/api/actuators?name=pregulator&start_time=0&end_time=2`, The module handler would receive as the parameters `name=pregulator&start_time=0&end_time=2`. This string had to be split into the key/value pairs, so the function `FCGI_KeyPair` being made.

With increased usage, this was found to be insufficient. `FCGI_ParseRequest` was created in response, and internally uses `FCGI_KeyPair`, but abstracts request parsing greatly. In essence, it validates the user input, rejecting anything that doesn't match a specified format. If it passes this test, it automatically populates variables with these values. The `IndentifyHandler` module handler in *fastcgi.c* is a very good example of how this works.

API	File	Function	Purpose
"`/api/identify`"	*fastcgi.c*	`IdentifyHandler`	Provide system information
"`/api/sensors`"	*sensors.c*	`Sensor_Handler`	Query sensor data points or set sampling rate
"`/api/actuators`"	*actuators.c*	`Actuator_Handler`	Set actuator values or query past history
"`/api/image`"	*image.c*	`Image_Handler`	Return image from a camera (See Section 2.7)
"`/api/control`"	*control.c*	`Control_Handler`	Start/Stop/Pause/Resume the Experiment
"`/api/bind`"	*login.c*	`Login_Handler`	Attempt to login to the system (See Section 2.4.5)
"`/api/unbind`"	*login.c*	`Logout_Handler`	If logged in, logout.

Figure 2.9: Brief description of the modules currently implemented by the server.

2.4.4 Server API - Response Format

The server API primarily generates JSON responses to most requests. This was heavily influenced by what the GUI would be programmed in, being JavaScript. This particular format is parsed easily in JavaScript, and is easily parsed in other languages too.

A standard JSON response looks like such:

```
{
    "module" : "identify",
    "status" : 1,
    "start_time" : 614263.377670876,
    "current_time" : 620591.515903585,
    "running_time" : 6328.138232709,
    "control_state" : "Running",
    "description" : "MCTX3420 Server API (2013)",
    "build_date" : "Oct 24 2013 19:41:04",
    "api_version" : 0,
    "logged_in" : true,
    "user_name" : "_anonymous_noauth"
}
```

Figure 2.10: A standard response to querying the "`/api/identify`" module

A JSON response is the direct representation of a JavaScript object, which is what makes this format so useful. For example if the JSON response was parsed and stored in the object `data`, the elements would be accessible in JavaScript through `data.module` or `data.status`.

To generate the JSON response from the server program, *fastcgi.c* contains a framework of helper functions. Most of the functions help to ensure that the generated output is in a valid JSON format, although only a subset of the JSON syntax is supported. Supporting the full syntax would overcomplicate writing the framework while being of little benefit. Modules can still respond with whatever format they like, using `FCGI_JSONValue` (aka. `FCGI_PrintRaw`), but lose the guarantee that the output will be in a valid JSON format.

Additionally, not all responses are in the JSON format. In specific cases, some module handlers will respond in a more suitable format. For example, the image handler will return an image (using `FCGI_WriteBinary`); it would make no sense to return anything else. On the other hand, the sensor and actuator modules will return data as tab-separated values, if the user specifically asks for it (e.g.: using `https://host/api/sensors?id=X&format=tsv`)

2.4.5 Server API - Cookies

The system makes use of HTTP cookies to keep track of who is logged in at any point. The cookie is a small token of information that gets sent by the server, which is then stored automatically by the web browser. The cookie then gets sent back automatically on subsequent requests to the server. If the cookie sent back matches what is expected, the user is 'logged in'. Almost all web sites in existence that has some sort of login use cookies to keep track of this sort of information, so this method is standard practice. In the server code, this information is referred to as the 'control key'. A control key is only provided to a user if they provide valid login credentials, and no one else is logged in at that time.

The control key used is the SHA-1 hash of some randomly generated data, in hexadecimal format. In essence, this is just a string of random numbers and letters that uniquely identifies the current user.

Initially, users had to pass this information as another key-value pair of the module parameters. However, this was difficult to handle, both for the client and the server, which was what precipitated the change to use HTTP cookies.

2.4.6 Client - JavaScript and AJAX Requests

JavaScript forms the backbone of the web interface that the clients use. JavaScript drives the interactivity behind the GUI and enables the web interface to be updated in real-time. Without JavaScript, interactivity would be severely limited, which would be a large hindrance to the learning aspect of the system.

To maintain interactivity and to keep information up-to-date with the server, the API needs to be polled at a regular interval. Polling is necessary due to the design of HTTP; a server cannot "push" data to a client, the client must request it first. To be able to achieve this, code was written in JavaScript to periodically perform what is termed AJAX requests.

AJAX requests are essentially web requests made in JavaScript that occur "behind the scenes" of a web page. By making such requests in JavaScript, the web page can be updated without having the user refresh the web page, thus allowing for interactivity and a pleasant user experience.

Whilst AJAX requests are possible with plain JavaScript, the use of the jQuery library (see Section 2.8.2) greatly simplifies the way in which requests can be made and interpreted.

2.5 Alternative Communication Technologies

This section attempts to explain the reasoning behind the communication method chosen. This choice was not trivial, as it had to allow for anyone to remotely control the experiment, while imposing as little requirements from the user as possible. These requirements can be summarised by:

1. A widely available, highly accessible service should be used, to reach as many users as possible
2. Communication between client and server should be fairly reliable, to maintain responsiveness of the remote interface
3. Communication should be secured against access from unauthorised persons, to maintain the integrity of the system

To satisfy the first criteria, remote control via some form of internet access was the natural choice. Internet access is widely established and highly accessible, both globally and locally, where it can be (and is) used for a multitude of remote applications. One only needs to look as far as the UWA Telelabs project for such an example, having been successfully run since 1994 [17].

Internet communications itself is complex, and there is more than one way to approach the issue of remote control. A number of internet protocols exist, where the protocol chosen is based on the needs of the application. Arguably most prevalent is the Hypertext Transfer Protocol (HTTP)[18] used in conjunction with the Transmission Control Protocol (TCP) - to distribute web pages and related content across the internet. Other protocols exist, but are less widely used. Even custom protocols can be used, but that comes at the cost of having to build, test and maintain an extra component of software that likely has no benefit over pre-existing systems.

As a result, being able to control the system via a web page and standard web browser seemed the most logical choice, which was why it was used in the final design. Firstly, by designing the system to be controlled from a web page, the system becomes highly accessible, given that where internet access is present, the presence of a web browser is almost guaranteed. Nothing else from the client is required.

Secondly, setup and maintenance for the server is less involved, given that there is a wide range of pre-existing software made just for this purpose. Many features of the web browser can also be leveraged to the advantage of the system — for example, communications between the client and server can be quite easily secured using Transport Layer Security (TLS, previously known as Secure Sockets Layer or SSL).

Thirdly, reliability of the communications is better guaranteed by using such existing technology, which has been well tested and proven to work of its own accord. While internet access itself may not always be fully reliable, the use of such protocols and correct software design allows for a fair margin of robustness towards this issue. For example, TCP communications have error checking methods built-in to the protocol, to ensure the correct delivery of content. Similarly, HTTP has been designed with intermittent communications to the client in mind[18].

2.5.1 Server Interface

Other options were explored apart from FastCGI to implement the server interface. Primarily, it had to allow for continuous sensor/actuator control independent of user requests, which may be intermittent.

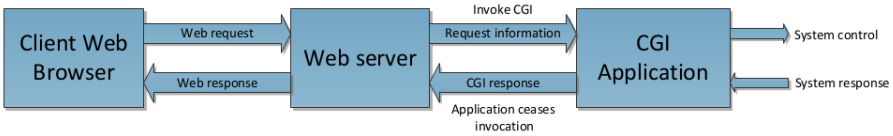

Figure 2.11: Block Diagram of a request to a CGI Application

Initially, a system known as "Common Gateway Interface", or CGI was explored. However, CGI based software is only executed when a request is received (Figure 2.11), which makes continuous control and logging over the sensors and actuators unfeasible.

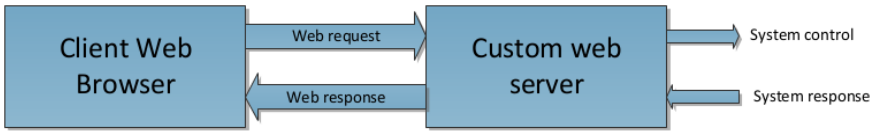

Figure 2.12: Block Diagram of a request to a custom web server

Another system considered was to build a custom web server (Figure 2.12) that used threading, integrating both the control and web components. This option was primarily discarded because it was inflexible to supporting extended services like PHP and TLS encryption. See Issue 6 on GitHub for more information.

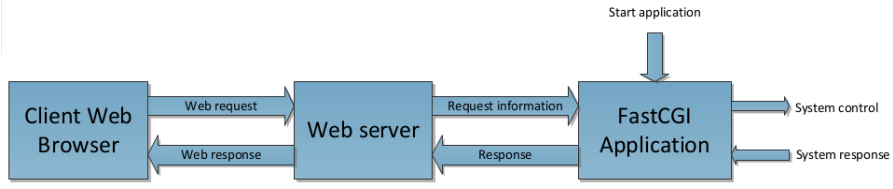

Figure 2.13: Block Diagram of a request to a FastCGI application

In comparison, FastCGI (Figure 2.13) can be seen as the "best of both worlds". As mentioned previously, it is a variant of CGI, in that it allows some software to respond to web requests.

The key difference is that with FastCGI, the program is continuously run independent of any web requests. This overcomes the issues faced with either using CGI or a custom web server; continuous control can be achieved while also not having to worry about the low-level implementation details a web server.

2.5.2 Recommendations for Future Work

1. A self-signed TLS certificate has been used, as it is free. It is equally secure as any, but users will get a security warning when accessing the web site. A proper TLS certificate signed by a trusted certificate authority should be used instead.
2. Consider expanding the framework of JSON functions to simplify creating a response.
3. Consider using X-Accel-Redirect along with UserCake (Section 2.3) to make a finer-grained access control system to information such as the system logs

2.6 BeagleBone Configuration

2.6.1 Operating system

The Beaglebone has been configured to use the Ubuntu operating system. The original operating system was Angstrom, which was unsuitable because it lacked a number of software packages required. Detailed instructions on how to install this operating system exist on the project wiki[1].

In particular, Ubuntu 13.04 running Linux kernel 3.8.13-bone28 was used, which is essentially the latest version available to date for this platform. Normally an older, more tested version is recommended, especially in a server environment. However, the BeagleBone Black is a relatively new device, and it was found that a lot of the drivers simply do not work well on older versions.

Specifically, there was much grief over getting the pins to function correctly, especially for PWM output. Lacking any great documentation, much trial and error was spent determining the best configuration. The BeagleBone Black uses what is termed a "device tree" [19, 20] and "device tree overlays" to dynamically determine what each pin does. This is because each pin can have more than one function, so a "device tree overlay" determines what it does at any one point. However, this also complicates matters, since what pins do essentially have to be loaded at runtime.

PWM control in particular took many hours to achieve, which was not helped by a lot of conflicting information available online. As a result, the primary tool used to correctly determine proper PWM control was the use of a cathode ray oscilloscope. Quite briefly, it was found that certain actions had to be performed in a very specific order to make PWM control available. The wiki goes into more detail on the issues found.

Getting the cameras to work on the BeagleBone was another major issue faced. After much testing, it was simply found that the cameras could only work on the latest version of the operating system. On anything else, only low resolution captures of around 352x288 pixels could be achieved.

Finally, it should be noted that USB hot-plugging does not work on the BeagleBone. This means that the cameras have to be plugged in before booting the BeagleBone. Upgrading to a newer kernel (when it exists) should solve this issue.

2.6.2 Required software

A number of packages are required to compile the code: `nginx spawn-fcgi libfcgi-dev gcc libssl-dev make libopencv-dev valgrind libldap2-dev mysql-server libmysqlclient-dev php5 php5-gd php5-fpm php5-mysqlnd`

These packages should be installed with the command `apt-get install`.

2.6.3 Required configurations

Many components need to be configured correctly for the server to work. In particular, these configurations relate to the web server, nginx, as well as logging software used, rsyslog. Executing *install.sh* as root should install all the required configuration files to run the server correctly.

2.6.4 Logging and Debugging

The function `Log` located in *log.c* is used extensively throughout the server program for debugging, warning and error reporting. This function uses syslog to simultaneously print messages to the `stderr` output stream of the program and log them to a file, providing a wealth of information about the (mal)functioning of the program. As discussed in Section 2.4.3, the logs may be also be viewed by a client using the server API.

For more low level debugging, ie: detecting memory leaks, uninitialised values, bad memory accesses, etc, the program `valgrind`[21] was frequently used.

2.7 Image Processing

The system contains two USB cameras, the Logitech C170[22] and the Kaiser Baas KBA03030 (microscope)[23]. The Logitech camera will be used to record and stream the can being pressurized to explode. The microscope will be used to measure the change in width in the can.

2.7.1 OpenCV

For everything related to image acquisition and processing we decided to use a library called OpenCV[24]. OpenCV uses the capture structure to connect with cameras, and stores the data in `IplImage` structures and the newer `CvMat` structure. As in C we cannot transfer the data straight to `CvMat` we need to convert from `IplImage` to `CvMat`. There are two main functions required for use with the camera. We need to be able to stream images to the user interface and use the microscope as a dilatometer, returning the rate of expansion of the can.

2.7.2 Image Streaming

The image streaming is done through the function file *image.c* and the header *image.h*. There are only 2 functions in *image.c*, both of which are externally accessible by the rest of the system.

The `Image_Handler` function handles requests from the server. The parameters required for taking the image, such as the camera ID, width and height are determined by calling `FCGI_ParseRequest` (see *fastcgi.h* and *fastcgi.c*) using the parameter string passed to the function.

The function `Camera_GetImage` in *image.c* is used to capture a frame on the camera from the ID given by `num`. As we cannot have 2 camera structures open at once, we use a mutex to ensure the function execute concurrently. We check to see if `num` is equivalent to the previous camera ID, if so we do not need to close the capture in order to recreate the connection with the new camera, which takes time. These considerations are currently redundant as the decision was made to only have one camera connected at a time, which was mainly due to power and bandwidth issues. However the code was implemented to allow for further development. If more than 2 cameras are ever connected, then the allowable upper bound for `num` will need to be increased to $n-1$ (where n is the number of cameras connected to the system).

After capturing the image we encode the `IplImage`, which passes back an encoded `CvMat`.

The image is then returned back to the web browser via `FCGI_WriteBinary`, where it can be displayed.

2.7.3 Dilatometer

The dilatometer algorithm is used to determine the rate of expansion of the can. The relevant functions are declared in *sensors/dilatometer.c* and *sensors/dilatometer.h*. When an experiment is started, `Dilatometer_Init` is executed. This creates all the necessary structures and sets the initial value of `lastPosition`, which is a static variable that stores the last edge found.

As the `Camera_GetImage` function in *image.c* is external, it can be accessed from *sensors/dilatometer.c*. This was done so that both the dilatometer and the image stream can gain access to the camera. The `IplImage` returned is converted to the `CvMat` structure `g_srcRGB`. This `CvMat` structure is then passed to a function, `CannyThreshold`. In this function, a series of steps are taken to extract an image containing only the edges. First we use `cvCvtColor` to convert the `CvMat` file to a grayscale image. The image is then blurred using the `cvSmooth` function, which we pass the parameters `CV_GAUSSIAN` and `BLUR`, so we use a Gaussian blur with a kernel of size `BLUR` (defined in *sensors/dilatometer.h*). The blurred file is then passed to the OpenCV Canny Edge detector.

The Canny Edge algorithm[25] determines which pixels are "edge" pixels through a series of steps. The algorithm applies the Sobel operator in the x and y directions using `KERNELSIZE` for the size of the kernel. The result of this gives the gradient strength and direction. The direction is rounded to 0, 45, 90 or 135 degrees. Non-maximum suppression is then used to remove any pixels not considered to be part of an edge. The pixels left are then put through the hysteresis step. If the gradient of the pixel is higher than the upper threshold (in our algorithm denoted by `LOWTHRESHOLD*RATIO`) then the pixel is accepted as an edge. If it is below the lower threshold (i.e. `LOWTHRESHOLD`) then the pixel is disregarded. The remaining pixels are removed unless that is connected to a pixel above the upper threshold (Canny Edge Detector). The defined values in the header file can be altered to improve accuracy.

The `CannyThreshold` function fills the `CvMat g_edges` structure with the current image edge (i.e. an image containing only pixels considering to be edges, see Appendix **??**). The code then finds the location of the line. It does this by sampling a number of rows, determined by the number of samples and the height of the image, finding the pixel/s in the row considered to be an edge. The algorithm then takes the average position of these pixels. The average position over all rows sampled then determines the actual edge position. The rows sampled are evenly spaced over the height of the image. If a row does not contain an edge, then it will not be included in the average. If a blank image goes through, or the algorithm has a low number of samples and does not pick up an edge, then the function will return false and the data point will not be recorded.

Once the edge is found, we will either return the position of the edge, if the `DIL_POS` ID is set. It needs to be noted that this will only show the change in position of one side of the can. If the `DIL_DIFF` ID is set then the value will be set to the difference between the current position and the last position, multiplied by `SCALE` and 2. We need to multiply by 2 as we are only measuring the change in width to one side of the can, however we must assume that the expansion is symmetrical. The scale will be used to convert from pixels to μm (or a more suitable scale). Currently the scale is set to 1, as the dilatometer has not been calibrated, thus we are only measuring the rate of change of pixels (which is arbitrary). The static variable, `lastPosition`, is then set to determine the next change in size. If the difference is negative, then the can is being compressed or is being depressurized. The rate of expansion can then be determined from the data set. As the system does not have a fixed refresh rate, however each data point is time-stamped. If the data is the edge position, then plotting the derivative of the time graph will show the rate of expansion over time.

2.7.4 Design Considerations

OpenCV

OpenCV was chosen as the image processing library primarily due to it being open source and widely used in image processing tasks. One thing to note however is the documentation for OpenCV for the language C is quite difficult to follow. This is mainly due to the fact that the source (despite originally being written for C) is now written primarily for use in C++, thus the documentation and some of the newer functionality is tailored more for C++. This caused some difficulty in writing the code for C as not all C++ functionality was available for C, or was included in a different or outdated fashion.

Memory Management

An initial problem I faced when coding in OpenCV was memory leaks. My simple program to take an image and save it to file was causing us to lose approximately 18Mb, which is unacceptable and would cause issues in the long term. After researching the issue I found that I was not properly releasing the structure dealing with storing the image for the data, `IplImage`. For example I was using:

```
1         cvReleaseImage(&frame);
```

When the correct release function is actually:

```
1         cvReleaseImageHeader(&frame);
```

Another thing to note was that releasing one of the `CvMat` structures (`g_srcRGB`) during the cleanup of the dilatometer module, a `NULL` pointer exception was returned and the program execution stopped. The reason for this is unknown, but the other `CvMat` structures appear to be released properly. For now I simply removed this release; however the cause should be looked into.

Dilatometer

The dilatometer code went through a few iterations. Originally we were informed by the Sensors Team that the camera would be watching the can, rather than object attached to the can. Thus my original algorithms were revolved around finding the actual width and change in width of the can.

Originally I designed the algorithm to find the edge of the can via the pixel thresholds. By finding the average position of the pixels below a certain threshold (as ideally you would have a dark can on a light background to create a contrast for the edge). This would already give a fairly inaccurate result, as it assumes a relatively sharp intensity gradient. Even with little noise the system would have accuracy issues.

To increase the accuracy in finding the edge, I considered the Canny Edge theorem. I wrote my algorithm to find all points above a certain threshold and take the average of these, considering this as an edge. I then scanned through the rest of the image until the next edge was found and do the same. The width of the can is found by taking the difference of the two locations. I also wrote an algorithm to generate these edges so I can test the algorithm. The function (`Dilatometer_TestImage`/, which is still located within *sensors/dilatometer.c*) generated two edges, with an amount of noise. The edges were created by taking an exponential decay around the edge and adding (and subtracting) a random noise from the expected decay. The edges where then moved outwards using a for loop. The results can be seen in Figure 2.14. From the graphs (Figure **??**) it can be seen how effective the algorithm was for a system with negligible noise, as it gave negligible percentage error. However with increasing levels of noise we notice a considerable increase in inaccuracy (Figure 2.15).

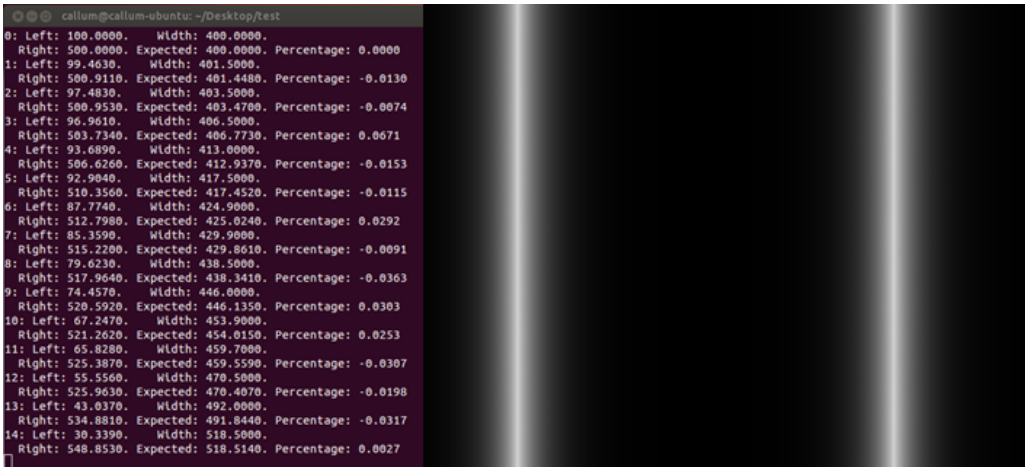

Figure 2.14: Output of canny edge algorithm applied to generated edges

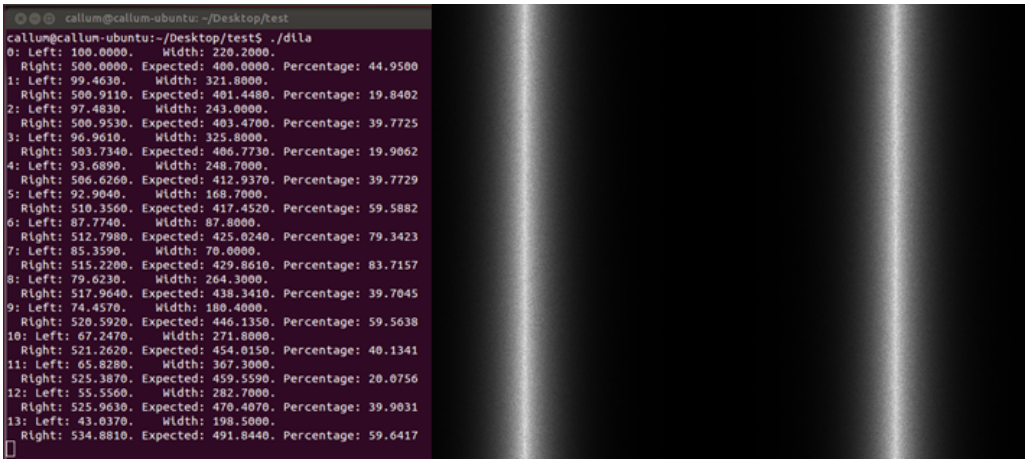

Figure 2.15: Output of canny edge algorithm applied to generated edges with generated noise

After the Sensors Team relayed that they were now attaching something to the can in order to measure the change position, I decided to simply stick with the Canny Edge algorithm and implement something similar to what I had in my previous testing. The images in Figure 2.17 shows the progression of the image through the algorithm. Figure 2.17 A shows the original image, whereas 2.17B shows the blurred (with a BLUR value of 5) gray scale image. Whereas figure 2.17C shows the image after going through the Canny Edge algorithm with a low threshold of 35. Figures 2.17D and 2.17E both have the same input image, however different input values. It can be seen how tweaking the values can remove outliers, as figure 2.17E is skewed to the right due to the outliers. From figure 2.17F it can be seen that despite there being no points in the edge in the top half of the image, the edge has still been accurately determined.

The testing done shows that given a rough edge with few outliers an edge can be determined, however there is an obvious degree of inaccuracy the greater the variance of the edge. The best solution to this however does not lie in software. If an edge was used that was straight even at that magnification with a good contrast then the results would be much more accurate (i.e. the accuracy of the dilatometer is currently more dependent on the object used than the software).

Interferometer

Earlier in the semester we were informed by the Sensors Team that instead of a dilatometer we would be using an interferometer. The algorithm for this was written and tested; it is currently

still located in the file *interferometer.c* and header *interferometer.h*. However development of the algorithm ceased after the sensors team informed us that the interferometer would no longer be implemented.

2.7.5 Further Design Considerations

- During testing we noted a considerable degree of lag between the image stream and reality. Further testing can be done to determine the causes and any possible solutions.
- A function to help calibrate the dilatometer should be created
- The algorithm should be tested over an extended period of time checking for memory leak issues caused by OpenCV.
- Possibly modify the code to allow the parameters used in the Canny Edge algorithm to be modified in real time so the user can try and maximize the accuracy of the results. The image with the edge superimposed on it can also be streamed to the client in the same manner as the image, so the user can have feedback.
- The algorithm can be improved to try and neglect outliers in the edge image; however this is not as necessary if the original object used gives a sufficiently smooth and straight edge.

2.7.6 Results

Figure 2.16 shows an image obtained from one of two dilatometers used in the system setup with collaboration between all teams. The image is of a white Lego tile attached to the can. This image was successfully streamed using the server software, and results of the dilatometer readings were monitored using the same software. Unfortunately we were unable to maintain a constant value for a stationary can, indicating that the algorithm needs further development. Due to a leak in the can seal we were unable to pressurize the can sufficiently to see a noticeable change in the edge position.

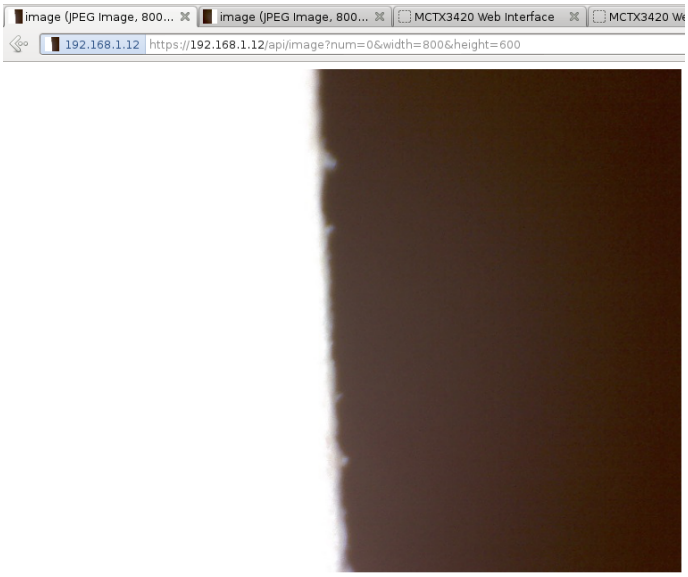

Figure 2.16: Microscope image of actual Lego tile attached to can in experimental setup

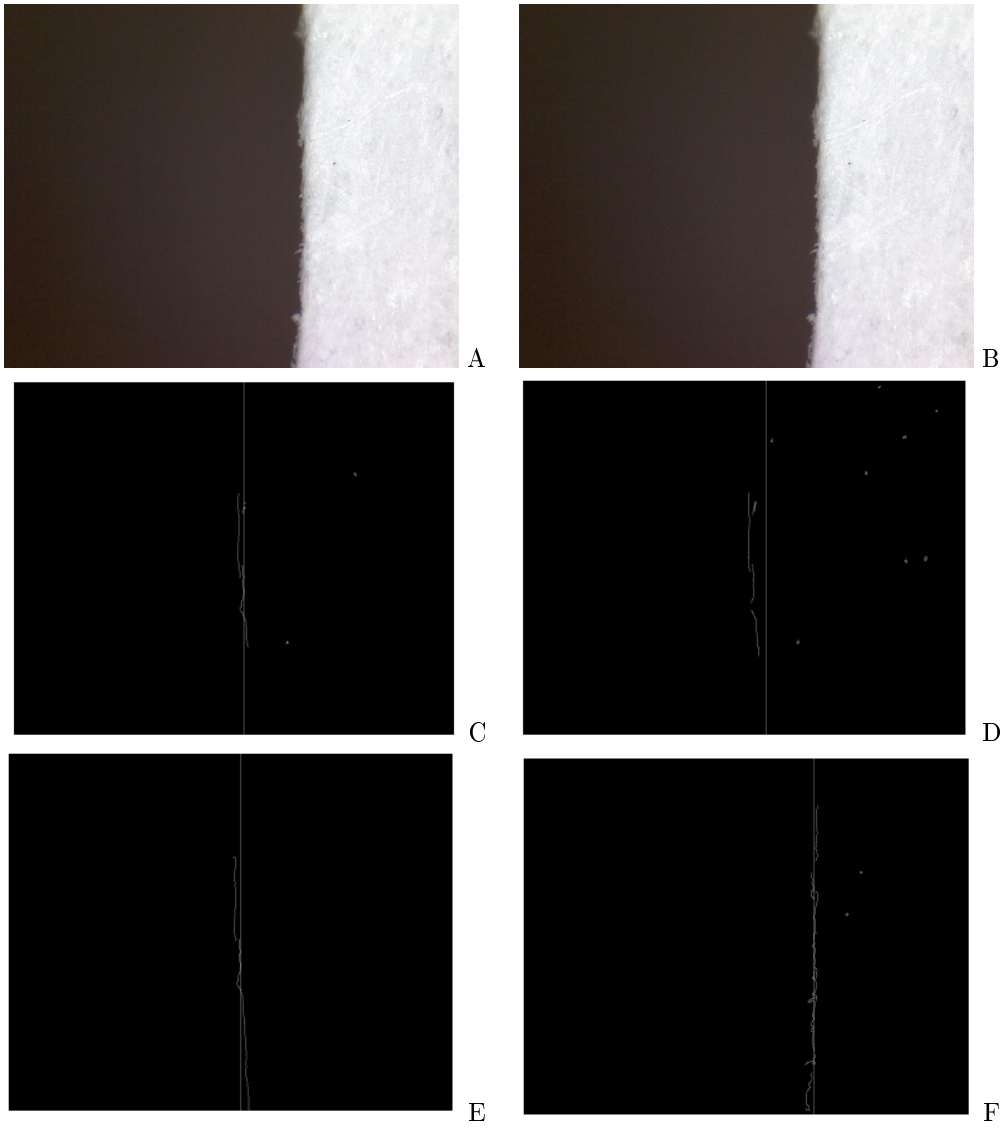

Figure 2.17: Canny Edge Algorithm in Action

2.8 Human Computer Interaction and the Graphical User Interface

2.8.1 Design Considerations

There are many considerations that are required to be taken into account for the successful creation of a Graphical User Interface (GUI) that allows Human Computer Interaction. A poorly designed GUI can make a system difficult and frustrating to use. A GUI made with no considerations to the underlying software can make a system inoperable or block key features. Without a well designed GUI the Human Computer Interaction becomes difficult and discourages any interaction with the system at all.

One of the key considerations made during the design of the GUI was the functionality it required. Originally this was limited to just allowing for simple control of the system including a start and stop and a display of system pressures however as the project progressed this was expanded to include a user login, limited admin functionality, graphing, image streaming and live

server logs. The addition of these features came as a result of changing requirements from the initial brief as well as logical progression of the GUI's capabilities. This gradual progression represents a continual improvement in Human Computer interaction for the system.

Ease of Use is the most important consideration of all to ensure that a GUI is well designed. Accessibility and user friendliness is a key aspect in web development. Burying key functionality inside menus makes it difficult to find and discourages its use. Making things obvious and accessible encourages use and makes the software quicker to learn which in turn means that the user is able to start doing what they want faster. However there are limits and care has to be taken to make sure that the user isn't bombarded with so many options that it becomes overwhelming for a first time user. Eventually a system of widgets in a sidebar was designed in order to satisfy the ease of use requirements by allowing functionality to be grouped and easily accessible.

Due to the limits of the Beagle Bone such as available memory and processing power it was important that the code, images and all libraries used were both small in size and efficient. This meant that careful consideration had to be made every time a library was considered for use. It also meant that where possible processing should be offloaded onto the client hardware rather than running on the server which already runs the server side code. This meant large libraries were ruled out and actions such as graphing were performed by the GUI on the client machine.

The final consideration is extensibility. An extensible software base code allows easy addition of new features. A good extensible interface makes it a simple case of simply dropping the extra code in in order to add extra features whereas a GUI that doesn't take this into account can require deleting and recoding of large chunks of the previous code. This means that the interface code must be structured in a coherent way and all conform to a "standard" across the GUI. Code must be laid out in the same way from page to page and where possible sections of code facilitating specific goals should be removed from the main page code. The latter was achieved through the use of the `.load()` JavaScript function allowing whole widgets to be removed and placed in their own separate files. This feature alone lets the developer add new widgets simply by creating a widget file conforming to the GUI's standard and then `.load()` it into the actual page.

2.8.2 Libraries used in GUI construction

These are libraries that we looked at and deemed to be sufficiently useful and as such were chosen to be used in the final GUI design.

jQuery

jQuery[26] is an open source library designed to make web coding easier and more effective. It has cross-platform and browser support all of the most common browsers. Features such as full CSS3 compatibility, overall versatility and extensibility combined with the light weight footprint made the decision to develop the GUI with this library included an easy one to make.

Flot

Flot[27] is a Javascript library designed for plotting and built for jQuery. This a lightweight easy to use library that allows easy production of attractive graphs. It also includes advanced support for interactive features and can support for IE < 9 . The Flot library provided an easy but powerful way to graph the data being sent by the server.

2.8.3 Libraries trialled but not used in GUI construction

These are libraries that were looked at and considered for use in the GUI software but were decided to not be used in the final product.

jQuery UI

jQueryUI[28] is a library that provides numerous widgets and user interface interactions utilising the jQuery JavaScript library. Targeted at both web design and web development the library allows easy and rapid construction of web application and interfaces with many pre-built interface elements. However this comes with the downside of being a larger library and provides many features that are unnecessary and is as such unfit for use in the GUI.

chart.js

chart.js[29] is an object orientated JavaScript library that provides graphing capabilities on the client side. The library uses some HTML5 elements to provide a variety of ways to present data including line graphs, bar charts, doughnut charts and more. It is a lightweight library that is dependency free however it is lacking on features compared to Flot and did not get used.

2.8.4 Design Process for the Graphical User Interface

As with any coding, following a somewhat strict design process improves efficiency and results in a better end product with more relevant code. Proper planning and testing prevents writing large amounts of code that is latter scrapped. It also provides a more focused direction than can be gleaned off of a project brief.

Producing test GUI's with simple functionality allows the developer to experiment and test features without investing a large amount of time and code in something that may not work or solve the required problem. The test GUI's can both functional and aesthetic. Throughout the project a large amount of test GUI's of both types were produced. Aesthetic test GUI's are great for experimenting with the look and feel of the software and allow the developer to experience first hand how the page handles. Functional GUI's on the other hand allow the developer to test out new features and investigate whether the client server interaction is functioning properly.

Whilst producing test GUI's a design document was drawn up. This document encompassed the design goals and specifications for the final Human Computer Interface and provided what was essentially a master plan. Include in the document were things such as what separate pages were to be included, the overall look of the system and what final functionality was desired.

Once a design document was completed a Master Template was created. Firstly a draft was created in PowerPoint using Smart Art and can be seen in Figure 2.18. After reviewing the draft and accepting the design a HTML template with CSS elements was produced. This template mimics the draft with some added features and improvements as seen in Figure 2.19. This was also reviewed and accepted and formed the base code for the GUI.

With the template completed functionality was then added. By copying the template exactly for each individual page the look of the software is kept the same throughout. Adding functionality is a simple case of substituting in functional code in the demonstration panels as well as adding the necessary JavaScript for the pages to function. Effort was made to keep as much functional code separated from the template itself and to load the code into the page from an external file in order to facilitate cleaner code with better expandability.

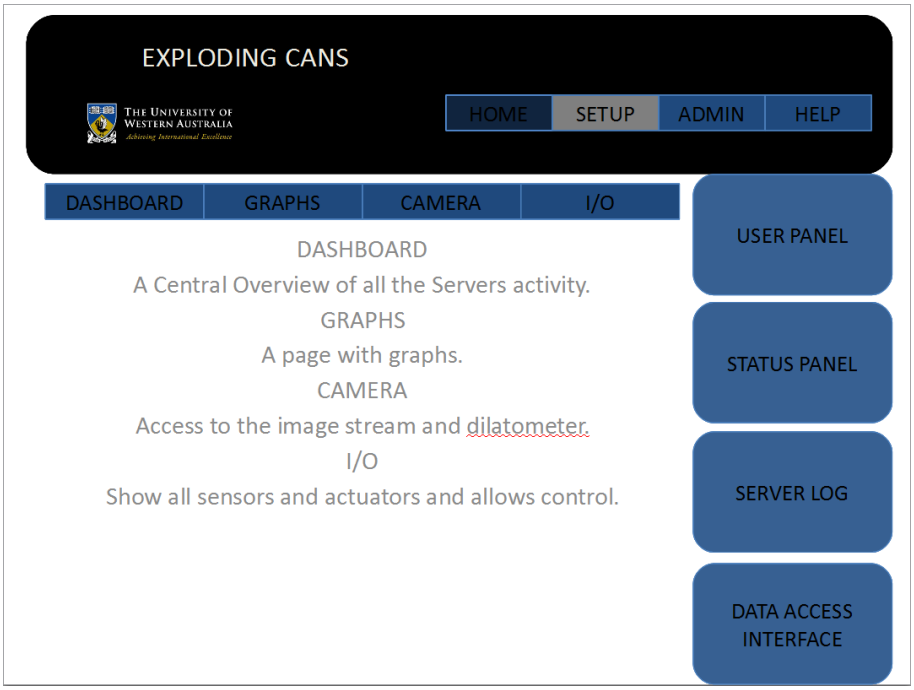

Figure 2.18: Draft GUI designed in Microsoft PowerPoint

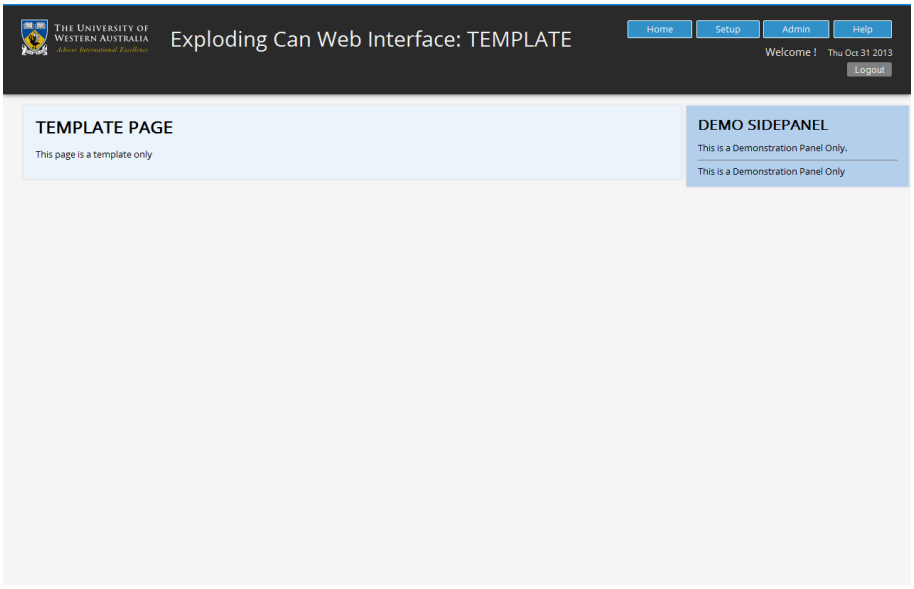

Figure 2.19: Screenshot of a GUI using templates to form each panel

2.9 GUI Design Process

2.9.1 Creation

The First iteration of the GUI was a relatively simple and almost purely text based. It held a graph, along with the basic image stream we had developed. It was formatted all down the Left hand side of the page.

Figure 2.20: First Test GUI

2.9.2 Testing

Secondly we decided to test the FastCGI protocol. Where FastCGI can be used to interface programs with a web server. This was the first test with the use of sensors and actuators theoretically collecting data from a server.

Figure 2.21: Testing GUI

This GUI was running over a free domain name which allowed us to play with control and command.

2.9.3 Iterations

After the basic testing of the initial GUIs we started playing with GUI design ideas which would be aesthetic, easy to use and reflect on UWA in a positive way. To do this we looked into how professional websites were made by opening their source code and investigating techniques into layout, structure and style. Then we went away and completed some GUI design trees, where

there would be a clear flow between pages.

2.9.4 Parallel GUI Design

During the GUI development phase, several GUIs were created. Some used graphical development software, while others used hard coded HTML, JavaScript, and CSS. Due to no organization within the group and a lack in communication a "final GUI" was made by several of the team members. Some of these are shown below.

2.9.5 GUI Aesthetics

Once we had decided on our core GUI design, we decided that, although not yet complete we would get Adrian Keating's opinion on the GUI design. While the GUI design was simple and functional Dr. Keating pointed out the design was bland. He encouraged us to release our artistic flair onto our GUI and make it more graphical and easy to use. Taking this into account we began work on another final GUI designing almost from scratch. We kept our GUI design flow, and worked largely on the look and feel of the GUI rather the functionality the GUI needed.

2.9.6 HTML Structure

The way our GUI works, in a nutshell, is that we use Basic HTML code to lay out what the page needs, then we have CSS(Styles) on top which lays out and formats the basic HTML code. We the put JavaScript files into the HTML code so that graphs and images and be streamed. In our GUI we have chosen to use JQuery to ask the server for information from the client and flot for graphing functionality.

2.9.7 Graphical Development VS Hard Coding

From the Multiple GUI we had accidentally created during the GUI design phase we noticed a large variety in the styles of GUIs that came out (Which shouldn't have happened) GUIs were created using HTML CSS and JavaScript being hard coded, from development software like Dreamweaver, and various Java based development platforms.

2.9.8 Final Design

The final concept consists of widgets and a navigation bar to the left. We decided for the maximum functionality we could get with the time remaining we would have pages for; Control, Graphs, Data, Data streaming, Pin debugging, and a help screen, shown below.

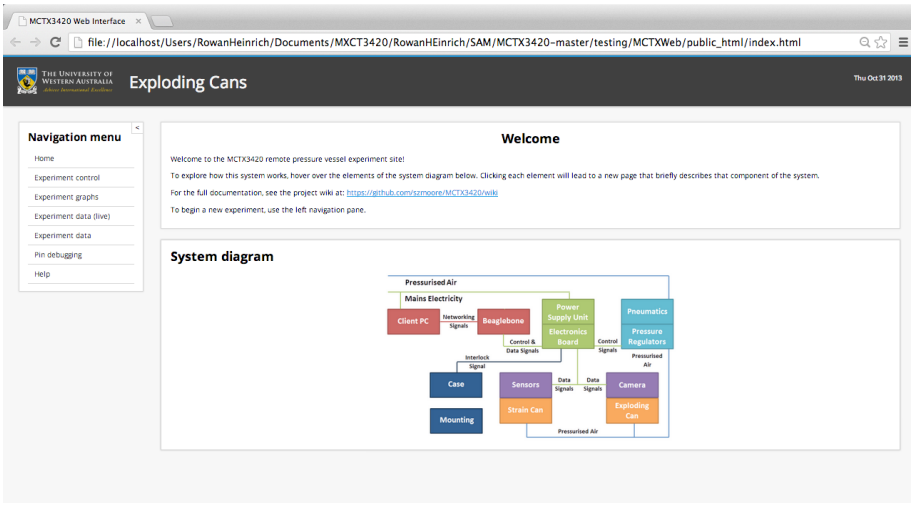

Figure 2.22: Final GUI

This is the "home screen" it shows the layout of the experiment, the subsystem and a welcome message.

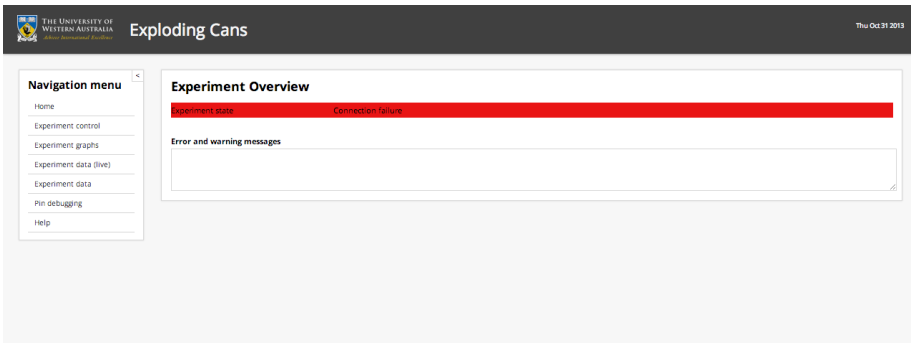

Figure 2.23: The Experiment (While disconnected from the server in the pic above) displays the Warnings and the experiment state to allow device use by only 1 student and avoid nasty conflicting control

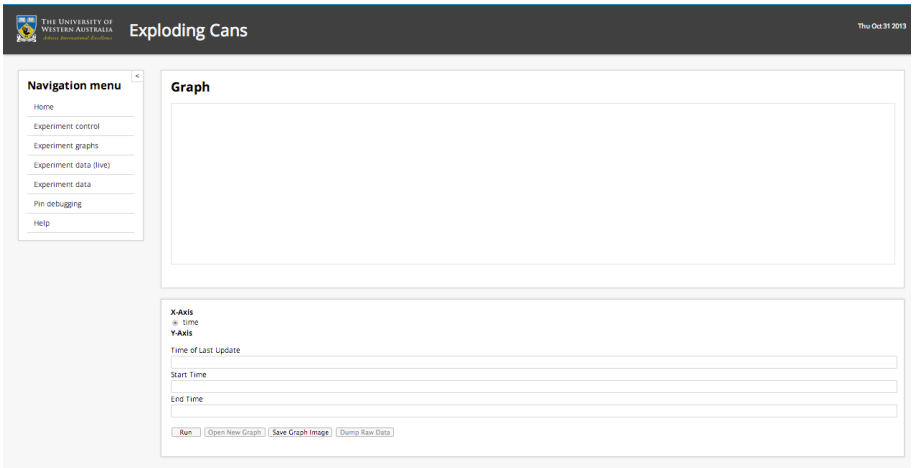

Figure 2.24: The Experimental Results page (also currently disconnected)

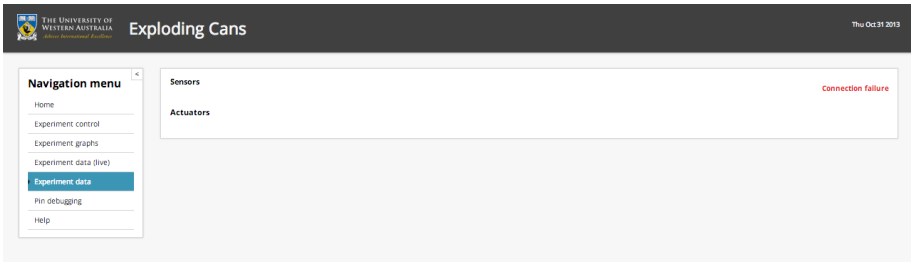

Figure 2.25: The experimental data page shows the start the sensors and actuators are reading, useful for checking the condition and measuring the experiment.

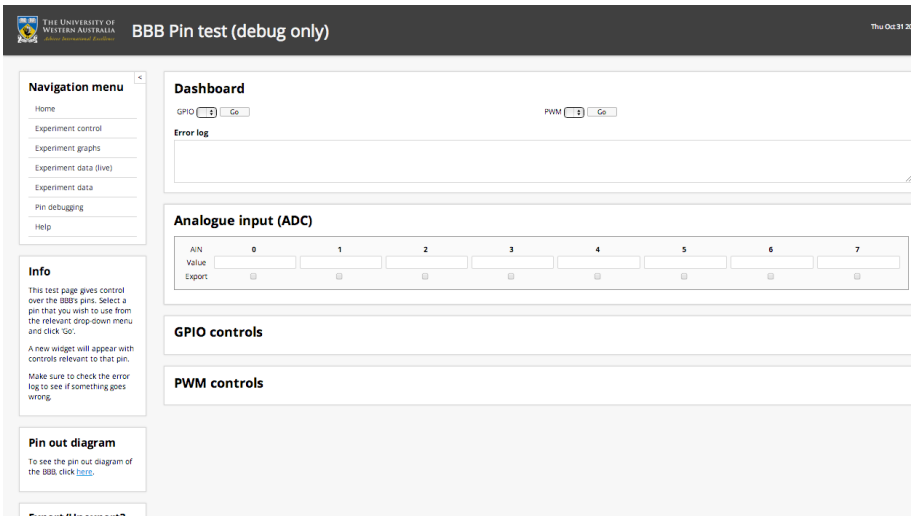

Figure 2.26: The BBB Pin test page is for the software team only so that we can test and debug the experiment we errors are found in the GUI or software.

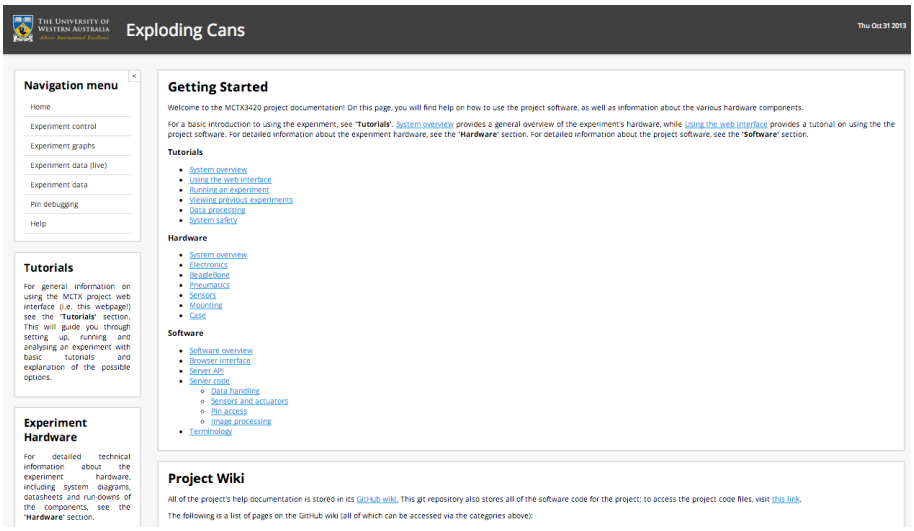

Figure 2.27: The help page, which links to the wiki information from all the teams and allows new users to look at all aspects of the project to be further developed and finished.

3. Conclusions and Recommendations

This report has described the work of the software team on the MCTX3420 pressurised can project during Semester 2, 2013 at UWA. In summary, we have succeeded in the following goals:

1. Design and implementation of a multithreaded process for providing continuous control over real hardware in response to intermittent user actions (Section 2.1, 2.2)
2. Design and implementation of a configuration allowing this process to interface with the *nginx* HTTP server (Sections 2.4, 2.6)
3. Use of image processing both for streaming images through the API and for use as a dilatometer (Section 2.7)
4. Design and implementation of a API using the HTTP protocol to allow a client process to supply user commands to the system (Section 2.4)
5. Design and implementation of the client process using a web browser based GUI that requires no additional software to be installed on the client PC (Section 2.4, 2.8)
6. Design and implementation of several alternative authentication mechanisms for the system which can be integrated with different user management solutions (Section 2.3)
7. Design and implementation of image streaming and image processing for use with a dilatometer (Section 2.7)
8. Partial design and implementation of a system for managing the datafiles of different users (Section 2.4.3)
9. Partial design and implementation of a user management system in PHP based upon UserCake (Sections 2.3, 2.4.5)
10. Integration and partial testing of the software with the overall MCTX3420 2013 Exploding Cans project involving extensive collaboration with a class of over 30 students (All sections)

We make the following general recommendations for further development of the system software (with more specific recommendations discussed in the relevant sections):

1. That the current software is built upon, rather than redesigned from scratch. The software can be adapted to run on a Raspberry Pi, or even a GNU/Linux laptop if required.
2. That more detailed testing and debugging of several aspects of the software are required; in particular:
   (a) The software should be tested for memory leaks by running for an extended time period
   (b) Any alternative image processing algorithms should be tested independently of the main system and then integrated after it is certain that no memory errors remain
3. That work is continued on documenting all aspects of the system.
4. That the GitHub Issues page[30] is used to identify and solve future issues and/or bugs
5. That members of the 2013 software team are contacted if further explanation of any aspect of the software is needed.

We would also like to make the following recommendations with regard to system hardware:

1. Care is given to protecting the BeagleBone from electrical faults (e.g.: overloading or underloading the ADC/GPIO pins, a power surge overloading the supply voltage)
2. A mechanism (possibly employing a high value capacitor) is included to allow a loss of power to be detected and the BeagleBone shut down safely

References

[1] J. Kruger, S. Moore, and J. Tan, "Mctx3420 project wiki." https://github.com/szmoore/MCTX3420/wiki, 2013.

[2] S. Moore, J. Tan, J. Kruger, C. Schofield, J. Rosher, and R. Heinrich, "Mctx3420 2013 git repository at github." https://github.com/szmoore/MCTX3420, 2013.

[3] S. Moore, J. Tan, J. Kruger, C. Schofield, J. Rosher, and R. Heinrich, "Mctx3420 git repository at ucc." http://git.ucc.asn.au/?p=matches/MCTX3420.git, 2013.

[4] GitHub, "Fork a repo." https://help.github.com/articles/fork-a-repo, 2013.

[5] "Mctx3420 2013 github contributions page." https://github.com/szmoore/MCTX3420/graphs/contributors.

[6] C. M. University, "C coding standards." http://users.ece.cmu.edu/~eno/coding/CCodingStandard.html.

[7] J. Kruger, "Safety systems - general outline." https://github.com/szmoore/MCTX3420/blob/master/notes/Safety%20Systems%20-%20general%20outline.docx, August 2013.

[8] Ohloh, "Mctx3420 project summary." http://www.ohloh.net/p/MCTX3420, 2013.

[9] "Beaglebone black (specifications)." http://beagleboard.org/Products/BeagleBone+Black/, 2013.

[10] Nginx.org, "Nginx http server." http://wiki.nginx.org/Main. v1.4.0.

[11] "Shadow man pages." http://linux.die.net/man/3/shadow.

[12] Wikipedia, "Lightweight directory access protocol." http://en.wikipedia.org/wiki/Lightweight_Directory_Access_Protocol.

[13] "Ldap man pages." http://linux.die.net/man/3/ldap. (API for libldab2-dev).

[14] MySQL.com, "Mysql: The world's most popular open source database." http://www.mysql.com/.

[15] T. et. al, "Usercake: The fully open source user management script." http://usercake.com, 2012.

[16] D. Eakins, "Lamp server: A brief overview." http://home.ite.sfcollege.edu/~daniel.m.eakins/media/Research_LAMP.pdf, 2012.

[17] J. Trevelyan, "10 years experience with remote laboratories." International Conference on Engineering Education Research, Olomouc, Czech Republic http://telerobot.mech.uwa.edu.au/Information/Trevelyan-INEER-2004.pdf, 2004.

[18] "Http specification (rfc2616)." http://www.w3.org/Protocols/rfc2616/rfc2616.html.

[19] P. Antoniou, T. King, and M. Porter, "Beaglebone and the 3.8 kernel." http://elinux.org/BeagleBone_and_the_3.8_Kernel, 2013. (Technical details of the Device Tree).

[20] Adafruit, "Introduction to the beaglebone black device tree." http://www.adafruit.com/blog/2013/07/29/tutorial-introduction-to-the-beaglebone-black-device-tree/, 2013. (Tutorial of the BeagleBone Black Device Tree).

[21] V. Developers, "Valgrind." http://valgrind.org/. v3.8.1.

[22] Logitech, "Logitech webcam c170 tehnical specifications." http://logitech-en-emea.custhelp.com/app/answers/detail/a_id/24412/~/logitech-webcam-c170-technical-specifications, 2013. (Driver for GNU/Linux is part of uvcvideo[31]).

[23] K. Baas, "Digital microscope: Study fine detail with 200x magnification." http://www.kaiserbaas.com/cameras/digital-microscope, 2013. (Driver for GNU/Linux is part of uvcvideo[31]).

[24] OpenCV, "Open source computer vision (opencv)." http://opencv.org/, 2013. v2.4.6.0 (we are using the C API).

[25] OpenCV, "Opencv documentation and tutorials: Canny edge detector." http://docs.opencv.org/doc/tutorials/imgproc/imgtrans/canny_detector/canny_detector.html.

[26] jQuery Foundation, "jquery: write less, do more." http://www.jquery.com, 2013. v1.10.1.

[27] FlotCharts.org, "Flot: Attractive javascript plotting for jquery." http://www.flotchargs.org, 2013. v0.8.1.

[28] jQuery Foundation, "jquery: User interface." http://jqueryui.com, 2013. v1.10.3.

[29] C. Organisation, "Chart.js." http://www.chartjs.org, 2013.

[30] "Mctx3420 2013 github issues page." https://github.com/szmoore/MCTX3420/issues?direction=asc&sort=updated&state=open.

[31] L. U. Project, "Usb video class (uvc) linux device driver." http://www.ideasonboard.org/uvc/, 2013.

Glossary

- **Server** — Refers to the MCTX3420 program that runs on the system and is responsible for controlling and querying hardware. “Server” is often also used to refer to a physical machine (computer or embedded device) that runs a Server program.
- **Client** — Refers to a program running on a computer that isn't part of the system. This program provides the user with an interface to the system; it will send commands and queries to the server as directed by a human user. “Client” is also often also used to refer to a physical machine that runs a Client program.
- **HTTP** — Hyper Text Transfer Protocol - The protocol used by web browsers and web servers to exchange information. A "web" server is technically called a HTTP server. A "web" client is something like a web browser (firefox, chrome, etc) which uses HTTP to query servers on the internet.
- **HTTPS** — HTTP itself involves sending plain text over a network, which can be intercepted and read by anyone on the network. The HTTPS protocol provides a layer of encryption to prevent eavesdropping on HTTP traffic.
- **API** — Application Programming Interface - A standard defined for programs to interact with each other. In our case, the "Server API" (discussed on this page) defines what the Client can request and give to the Server.
- **HTML** — Hypertext Markup Language - A language used by web browsers to display web pages. Static. HTML files are stored on a system that is running a HTTP server and transferred to web browsers when they are requested.
- **JavaScript** (not to be confused with Java) — A language that is interpreted by a web browser to produce HTML dynamically (which is then rendered by the browser) in response to events. It can also direct the browser to send HTTP queries (AJAX). The response can be interpreted by the JavaScript. JavaScript files are also stored on the server.
- **JSON** — JavaScript Object Notation - Text that can be directly interpreted as an Object in JavaScript.
- **CGI** — Common Gateway Interface - Protocol by which HTTP servers respond to requests by calling an external (seperate) program. The CGI program does not run continuously.
- **FastCGI** — Fast Common Gateway Interface - Protocol by which HTTP servers respond to requests by passing them to an external (separate) program. Differs from CGI because the external program runs continuously and separately from the HTTP server.
- **IP Address** — Internet Protocol Address - Identifies a device on a network
- **Hostname** — A human readable name of a device on a network. The hostname of the device is associated with its IP address.
- **Multithreading** — A technique by which a single set of code can be used by several processors at different stages of execution.
- **OpenCV** — A real time Image processing library
- **BBB** — the BeagleBone Black, ARM processor board acts as the client, and communicates with the server to send and request data for physically running the experiment.
- **nginx** — Used for website architecture which integrates efficiency with functionality
- **OpenMP** — Multiplatform memory processing: used for parallel tasks (not used in this project)
- **PThreads (POSIX Threads)** — A library used for thread management defining a set of c programing functions and constants.