MCTX3420

Contents

1. Introduction

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[**?**].

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 below, with control components in red, electronics in green, sensors in purple, pneumatics in blue, and experimental targets in orange.

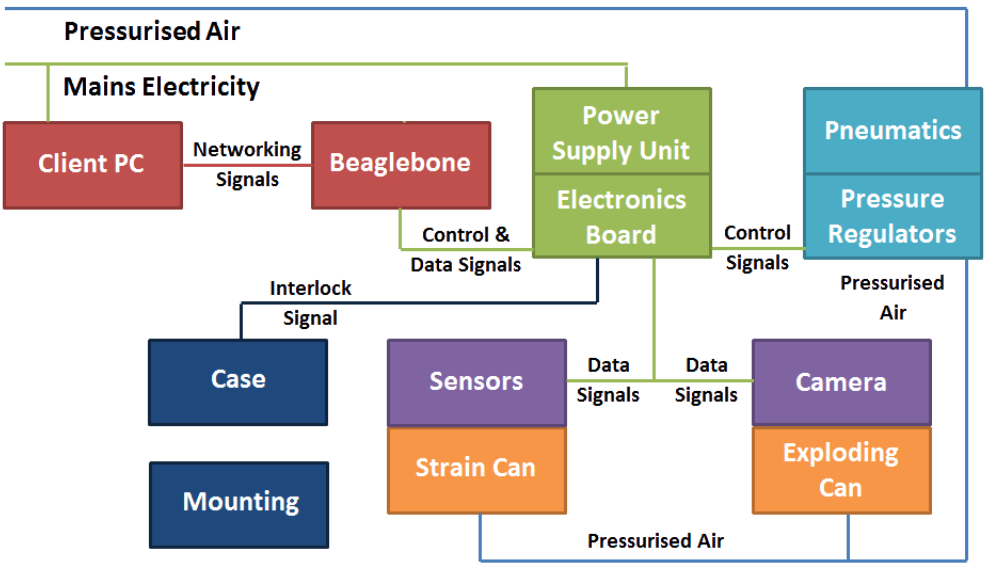

Figure 1.1: Block diagram of the physical system

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 vessels 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 systems 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 systems pneumatics feed the desired air pressure into the two pressure vessels being tested. Air is fed through a series of pipes from the laboratorys 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 cans 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 and GitHub website. 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[**?**].

The Git system ensures that work by different team members is tracked, 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 codebase.
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[**?**]. 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[**?**] 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 codebase. 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 projects end goals.

1.3.1 Communication

The primary time for collaboration was during the teams 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 groups repository for software work. The usefulness of GitHub was explained previously in the General Development Process section, 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 projects 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 teams 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**??**, 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 projects 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 timeframe. 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 vita 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 codebase). 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 teams 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 teams 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 BeagleBones 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[**?**], 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 Implementation

Figure **??** shows the earliest high level design of the software for the system created in the first week of the project. At this stage 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).

Figure **??** shows the revised diagram at the time of writing this report. 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 occured quite early in the project[1] is the switch from using multiple processes to running a single multithreaded process on the server. After performing some rudimentary testing it became clear that a system of seperate 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.

2.1 Hardware Interfacing

Figure **??** shows the pin out diagram of the BeagleBone black. There are many contradictory pin out diagrams available on the internet; Figure **??** was created by the software team after trial and error testing to determine the correct location of each pin.

The final specification of the pins and functions was chosen by the electrical team, although several earlier specifications were rejected after difficulties controlling the pins in software. These pins are identified in Table **??**.

2.1.1 Calibration Methods

Calibration of the sensors was done at a fairly late stage in the project and only a small number of test points were taken. With the exception of the microscope (discussed in Section **??**), 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.

2.2 Server Program

2.2.1 Threads and Sampling Rates

The Server Program runs as a multithreaded process under a POSIX compliant GNU/Linux operating system[2]. Each thread runs in parallel and is dedicated to a particular task; the three types of threads we have implemented are:

1. Main Thread**??** - Starts all other threads, accepts and responds to HTTP requests passed to the program by the HTTP server in the `FastCGI_Loop` function.
2. Sensor Thread**??** - Each sensor in the system is monitored by an individual thread running the `Sensor_Loop` function.
3. Actuator Thread**??** - Each actuator in the system is controlled by an individual thread running the `Actuator_Loop` function.

[1] about week 2
[2] Tested on Debian and Ubuntu

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 **??** shows a distribution of times between samples for a test sensor with the software sampling as fast as possible. Figure **??** shows the distribution when the sampling rate is set to 20Hz. Caution should be taken when interpreting these results, as they rely on the accuracy of timestamps recorded by the same software that is being time sliced by the operating system.

RTLinux is a version of the linux kernel that attempts to increase the predictability of when a process will have control[**?**]. It was not possible to obtain a real time linux kernel for the BeagleBone. However, testing on an amd64 laptop (figure **??**) showed very little difference in the sampling time distribution when the real time linux kernel was used.

2.2.2 Main Thread

The main thread of the process is responsible for transfering 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[**?**]. This configuration and the format of data transferred between the GUI and the server is discussed in more detail Section **??**.

Essentially, the main thread of the process responds to HTTP requests. The GUI is designed to send requests periodically (eg: to update a graph) or when a user action is taken (eg: 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 **??** gives an overview of this process.

2.2.3 Sensor Threads

Figure **??** 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. `DataPoint`s are continously saved to a binary file as long as the experiment is in process. An appropriate HTTP request (see section**??**) will cause the main thread of the server program to respond with `DataPoint`s 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 **??** 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.

2.2.4 Actuator Threads

Actuators are controlled by threads in a similar way to sensors. Figure **??** shows a flow chart for these threads. This is implemented in `Actuator_Loop`. Control over real hardware is seperated 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 syncronise 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.

2.2.5 Data Storage and Retrieval

Each sensor or actuator thread stores data points in a seperate 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.

Several alternate means of data storage were considered for this project. Binary files were chosen because of the significant performance benefit (see Figure **??**) and 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.

2.2.6 Authentication

The `Login_Handler` function is called in the main thread when a HTTP request for authentication is received. 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. 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 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 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 feasable to employ one of these to maintain a list of users authorised to access the experiment. UserCake is recommended, as it is both minimalistic and open source, so can be modified to suit future requirements.

   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.

2.2.7 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[**?**] 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 (eg: 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 deinitialised. 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 a `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.

2.2.8 Performance

Figure **??** shows the CPU and memory usage of the server program with different numbers of dummy sensor threads. This gives an idea of how well the system would scale if all sensors were run on the same BeagleBone.

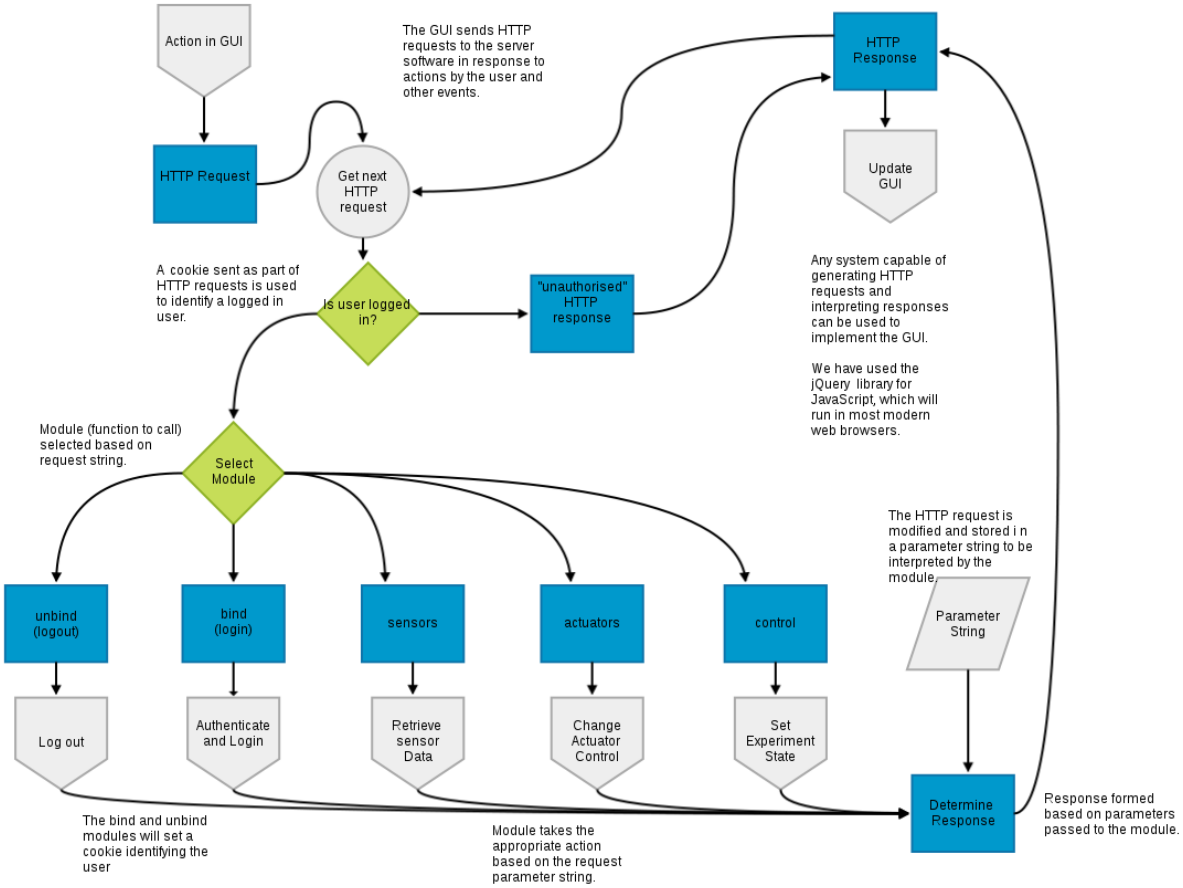

Figure 2.1: Server overview

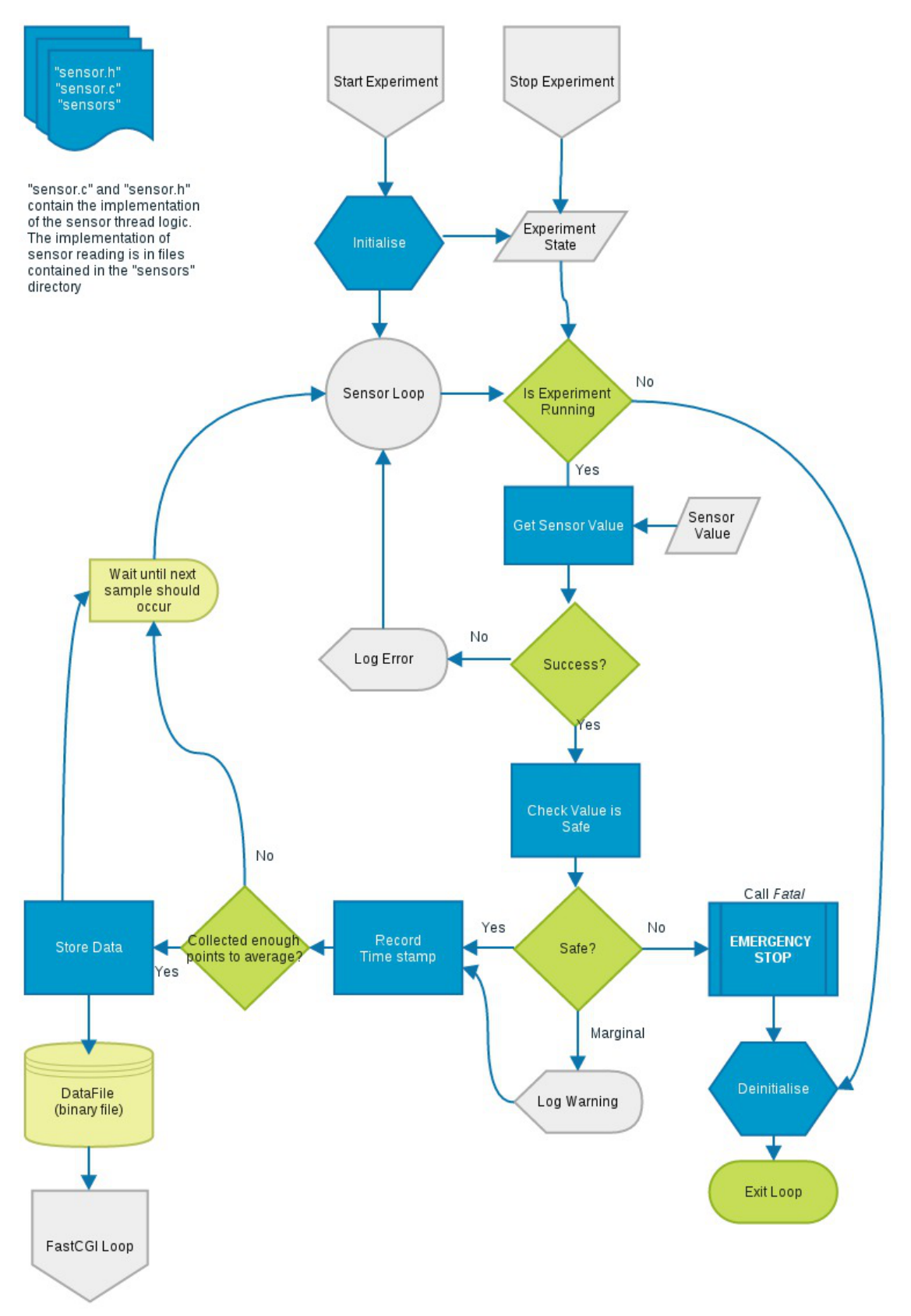

Figure 2.2: Flow chart for a sensor thread

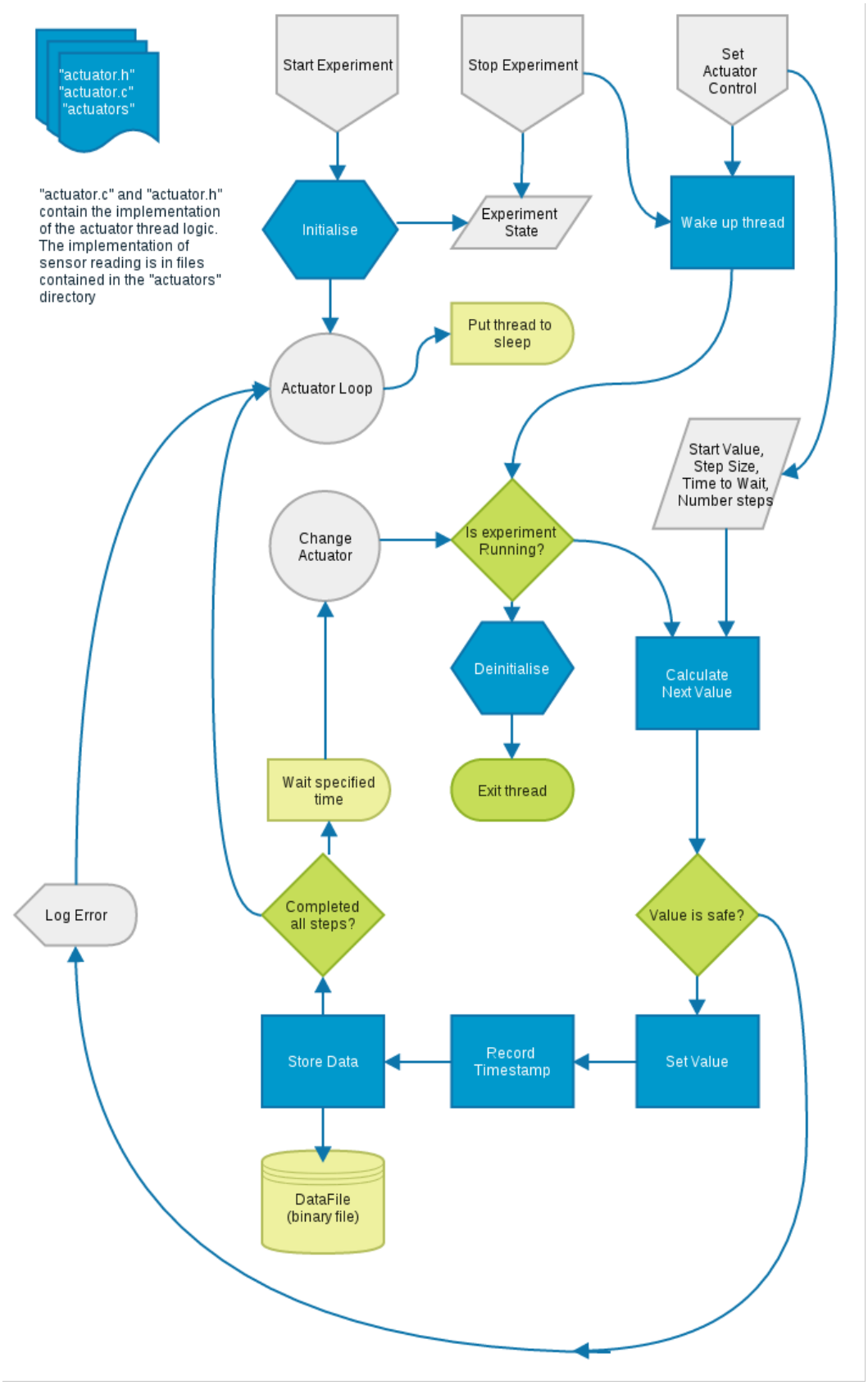

Figure 2.3: Flow chart for an actuator thread

2.3 Image Processing

2.4 Client Program

2.4.1 Human Computer Interaction

2.4.2 Interaction with API

3. Approach

3.1 Goals

3.2 Team Organisation

3.3 Development Process

3.4 Resources

3.5 Cost Calculation

4. Results

4.1 Results

4.1.1 Control of System

4.1.2 Design of GUI

4.1.3 Security and User Management

4.2 Recommendations

4.2.1 Approach and Integration

4.2.2 Hardware Control

4.2.3 Detect Loss of Power

4.2.4 Detect Program Crashes

4.2.5 Image Processing

4.2.6 GUI Design

4.2.7 BeagleBone/Server Configuration

4.2.8 Security and User Management

4.2.9 Debugging and Testing

4.3 Conclusions

This report has described the work of the software team on the MCTX3420 pressurised can project during Semester 2, 2013 at UWA.