MCTX3420

Contents

1. Introduction

Abstract

1.1 Objectives

1.2 Constraints

1.3 System Overview

1.3.1 Pneumatics

1.3.2 Electronics

1.3.3 Sensors

1.3.4 Mounting

1.3.5 Case

1.3.6 Software

The remainder of this report will focus on the software design.

2. Approach

2.1 Goals

2.2 Team Organisation

2.3 Development Process

2.4 Resources

2.5 Cost Calculation

3. Design Implementation

3.1 Overview

Figure ?? shows a high level block diagram of the software design.

The main subsystems to consider are:

1. Server Program A process running on the BeagleBone is responsible for overseeing control of the experiment. The program runs under a GNU/Linux operating system[1].
2.

3.2 Server Program

3.2.1 Naming Convention

Unless otherwise noted, the following naming convention applies:

3.2.2 Threads

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; all threads share the same memory and rescources. The three types of threads we have implemented are:

1. FastCGI (main) Thread - A single thread which accepts and responds to HTTP requests passed to the program by the HTTP server
2. Sensor Thread - Each sensor in the system is monitored by a single thread
3. Actuator Thread - Each actuator in the system is controlled by a single thread

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

[1]Debian or Ubuntu
[2]Tested on Debian and Ubuntu

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 showed very little difference in the sampling time distribution when the real time linux kernel was used.

3.2.3 Safety Mechanisms

3.2.4 Sensors

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 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 a user can safely close the GUI or even shutdown their computer and return to the experiment later without losing any data.

As the chart 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.

3.2.5 Actuators

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 to make it easy to compare the user settings

3.2.6 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.

3.2.7 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 on the beaglebone.

2. Lightweight Directory Access Protocol (LDAP) LDAP is a widely used data base for storing user information. A program that uses LDAP for authentication can query an LDAP server over a network; the LDAP server will respond indicating if the user and password match those stored in its database.

   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.

3. MySQL Database BLARGH

   MySQL databases are vulnerable to many different security issues. Care should be taken to ensure that all these issues are addressed before deploying the system.

3.2.8 Server API

3.2.9 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.

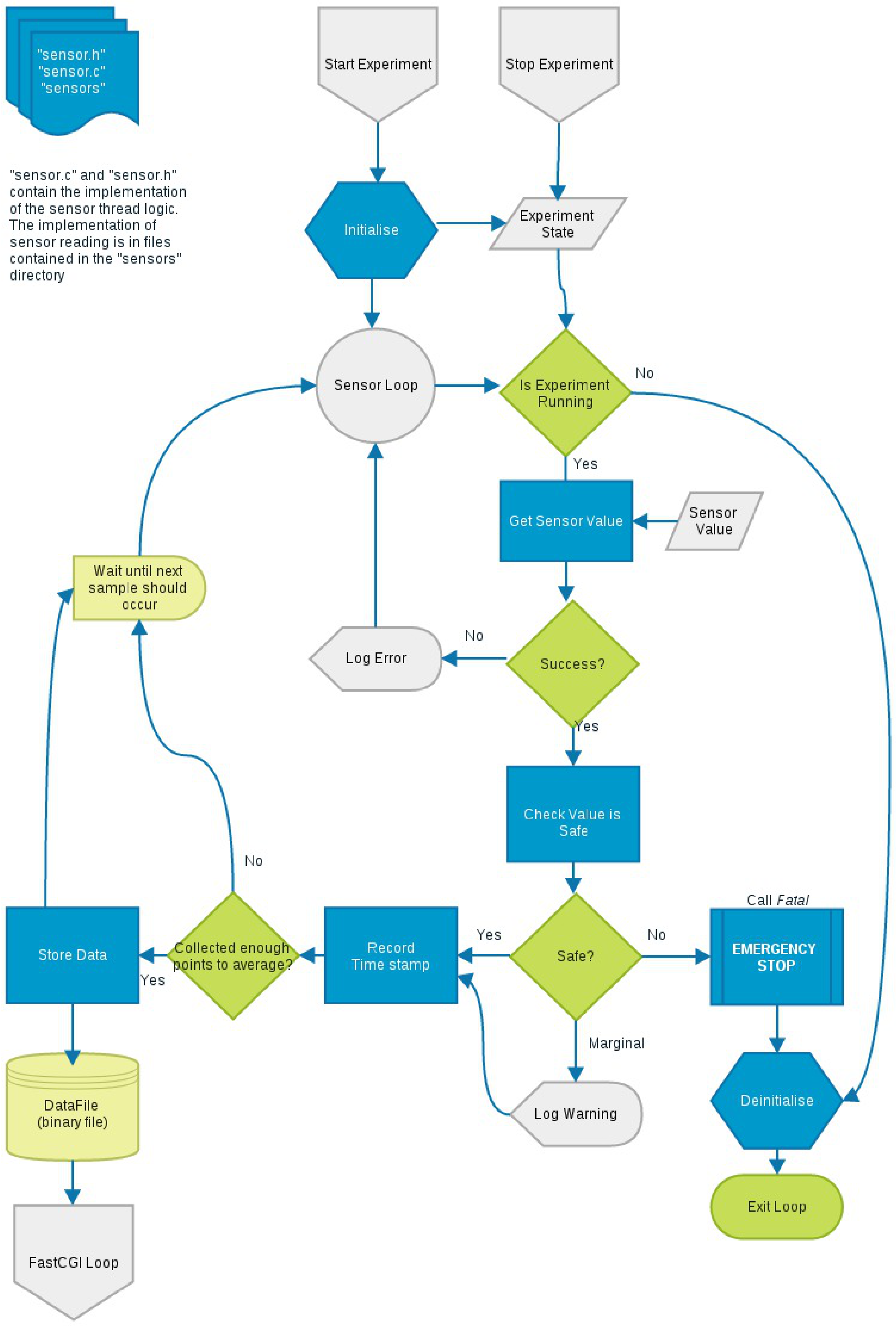

Figure 3.1: Flow chart for a sensor thread

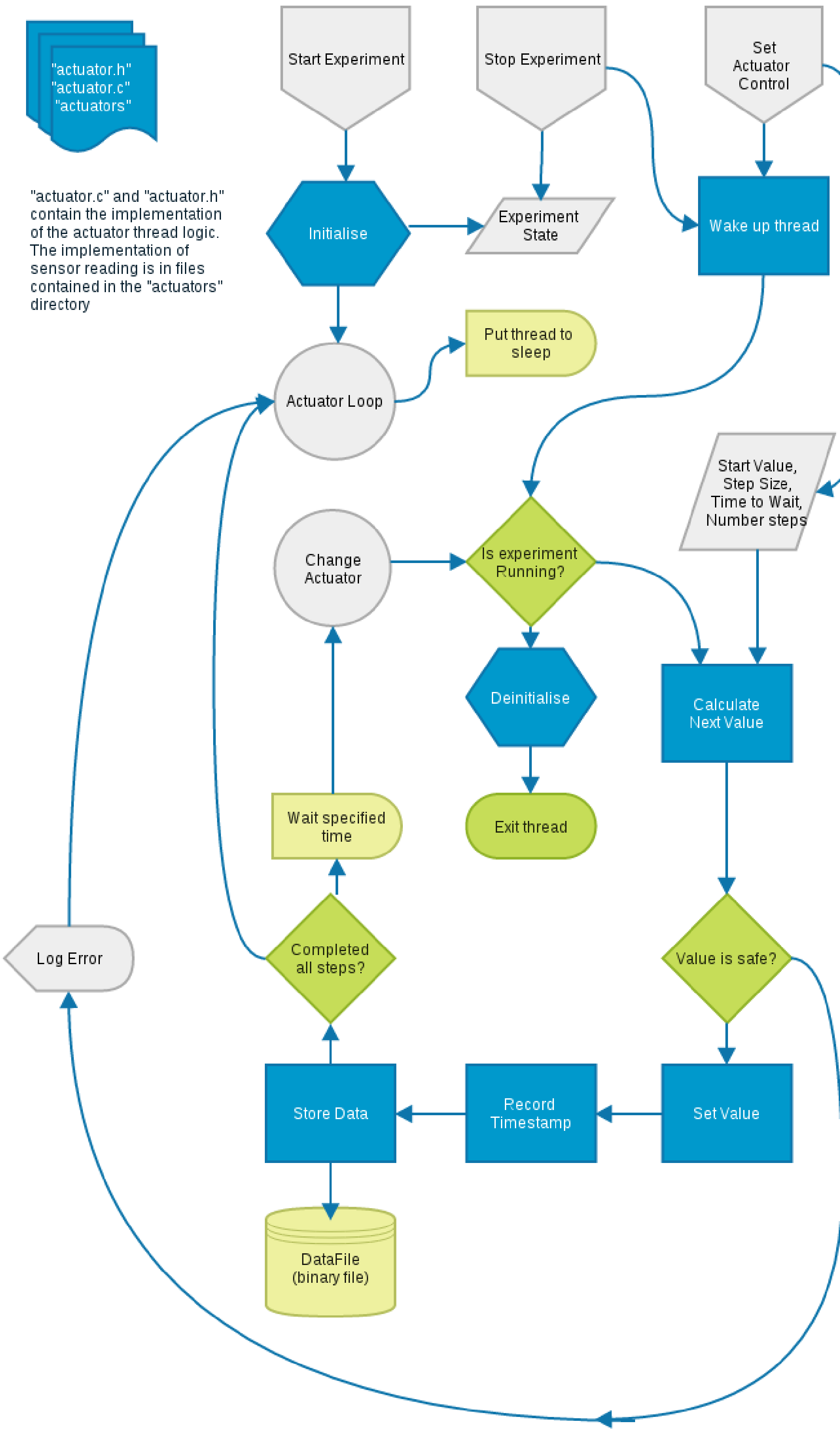

Figure 3.2: Flow chart for an actuator thread

3.3 Image Processing

3.4 Client Program

3.4.1 Human Computer Interaction

3.4.2 Interaction with API

4. Results

4.1 Results

4.2 Conclusion

4.3 Recommendations