2021

Development of a flow injection micro analysis system using an ARM microcontroller with an interactive web-based interface

Andrewartha, William Ross

Andrewartha, W.R. (2021) 'Development of a flow injection micro analysis system using an ARM microcontroller with an interactive web-based interface', The Plymouth Student Scientist, 14(1), pp. 108-144.
http://hdl.handle.net/10026.1/17335

The Plymouth Student Scientist
University of Plymouth

*The Plymouth Student Scientist, 2021, **14**, (1), 108-144*

Development of a flow injection micro analysis system using an ARM microcontroller with an interactive web-based interface

William Ross Andrewartha

Research Advisor: Dr Paul Davey, School of Engineering, Computing and Mathematics University of Plymouth, Drake Circus, Plymouth, PL4 8AA

Abstract

Monitoring the state of the environment is increasingly important in today's changing world. Flow Injection Analysis techniques are useful for measuring a wide range of solutions in a cost-effective and controlled manner. These include seawater samples from which the levels of iron present can be quantified. During the analysis, a wide variety of devices need to be controlled precisely to ensure accurate and repeatable results. Such devices include peristaltic pumps, switching valves, six-port valves and solenoid valves.

The system is simple to configure by using a bespoke web-based interface to configure and control an STM32 ARM microcontroller. The routine can be visualised to check that it looks correct, and then the information is sent in a JSON format to the microcontroller. The microcontroller then sends the appropriate control signals to the devices following the programmed routine, allowing users to analyse substances such as iron with minimal training.

Keywords: Flow injection, solutions, peristaltic pump, valves, ARM, STM32, microcontroller, web, internet, interactive, API, JSON, JavaScript Object Notation, C++, programming, automation, software design.

Introduction

Background

Flow injection analysis involves the combination of millilitre level fluid streams under highly repeatable conditions. It achieves this by using peristaltic pumps [1], switching valves, solenoid valves and six-port valves. In order to automate the process to save time and money, a controller can be used to control the devices at precise times to ensure accurate and repeatable experimental results.

Research Procedure

1. Develop a microcontroller-based system that can operate a peristaltic pump, switching valves and solenoid valves according to a pre-programmed timing routine.
2. Create a web-based interface to allow the user to interact with the system and adjust the timings of the analytical cycle.
3. Produce a comprehensive user manual covering the operation of the system and its components, as well as troubleshooting steps and design philosophy for the system.
4. Design and simulate a PCB including device monitoring functionality.

Research

As stated in the Research Procedure, a critical function of the controller is its ability to interface with and control a wide variety of lab equipment, using standard connectors where possible. Standard connectors allow the devices to be swapped around, allowing the controller to adapt to the different device requirements of the various experiments. For example, one experiment may require three pumps and one valve, while another needs one pump and three valves. The controller needs to be flexible to adapt to the changing device configuration with minimal setup time.

Concept Design

Concept 1 – Control using proprietary connections on a computer

Concept 1 controls the pumps and valves using the manufacturer provided proprietary software on a computer. By looking at the manual for the pump [1], the manufacturer, Gilson Inc, provides a proprietary interface called GSIOC. GSIOC allows for full control and monitoring of the pump; however, the software is currently only available for Microsoft ® Windows, requiring additional drivers to be written for use with other platforms. The valves are similar, with limited support for non-windows operating systems. This system would, therefore, be specific to the pumps and valves that it was designed for, requiring additional work to integrate valves that are not of the same manufacturer. Also, the cost of a Windows ® PC is high, with the required interface cards adding further costs. Due to these limitations, this concept will not be developed further.

Concept 2 – Control using standard electronic connectors
Concept 2 controls the pumps and valves by using standard interface connectors, which are present on their control unit. Standard connectors allow for much greater interoperability than proprietary ones and prevent the system from being tied to a particular operating system. This allows pumps and valves from different manufacturers to be used in the future with minimal software reconfiguration. This provides a more open system without licencing fees or other recurring costs. Also, the use of standard connectors means that the system can be configured to suit the particular requirements of the analysis being performed. This allows the control to be much more flexible and useful to run different analyses. This concept will be developed in this research.

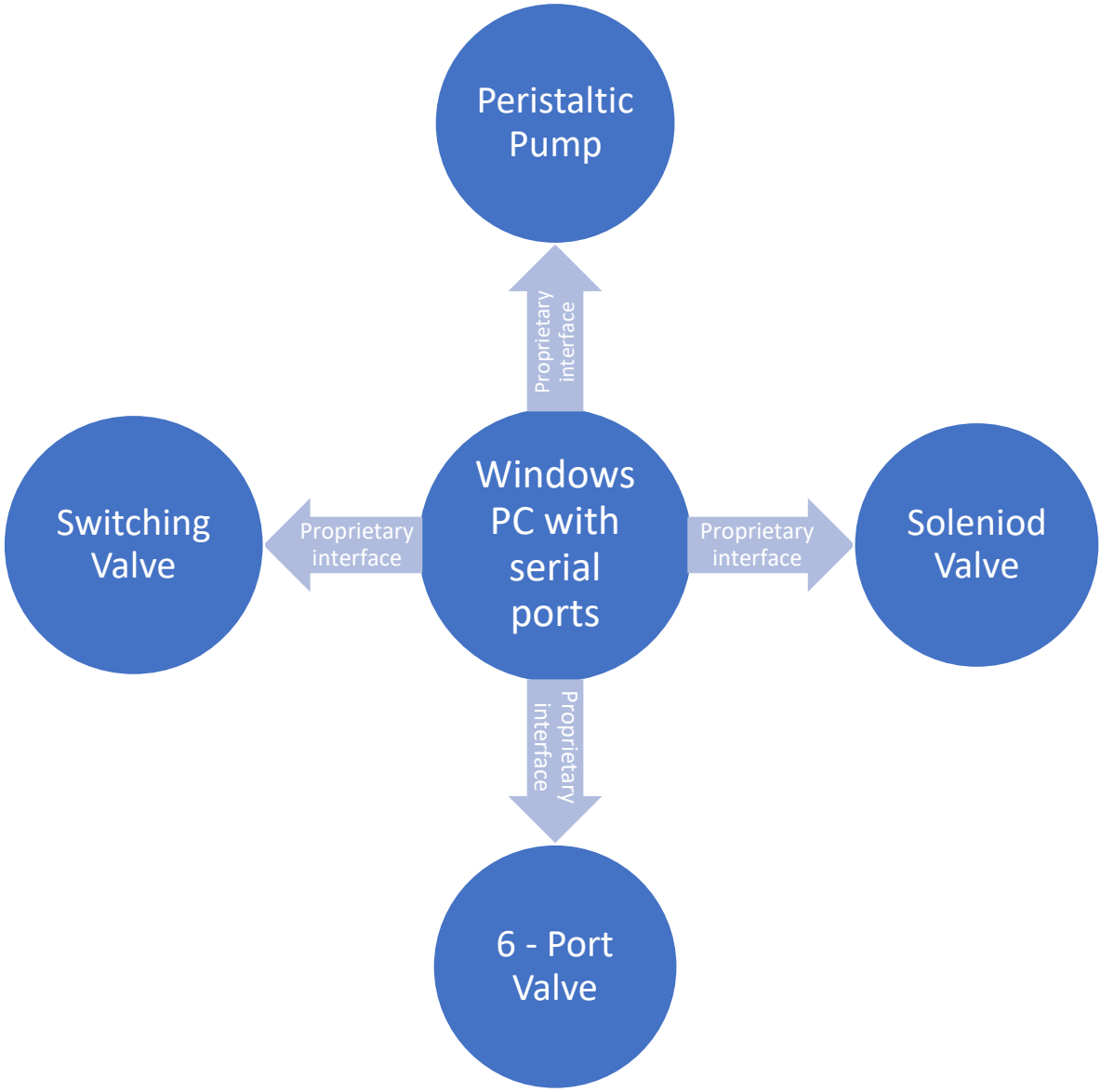

Figure 1: Concept 1: Using proprietary interfaces to control devices

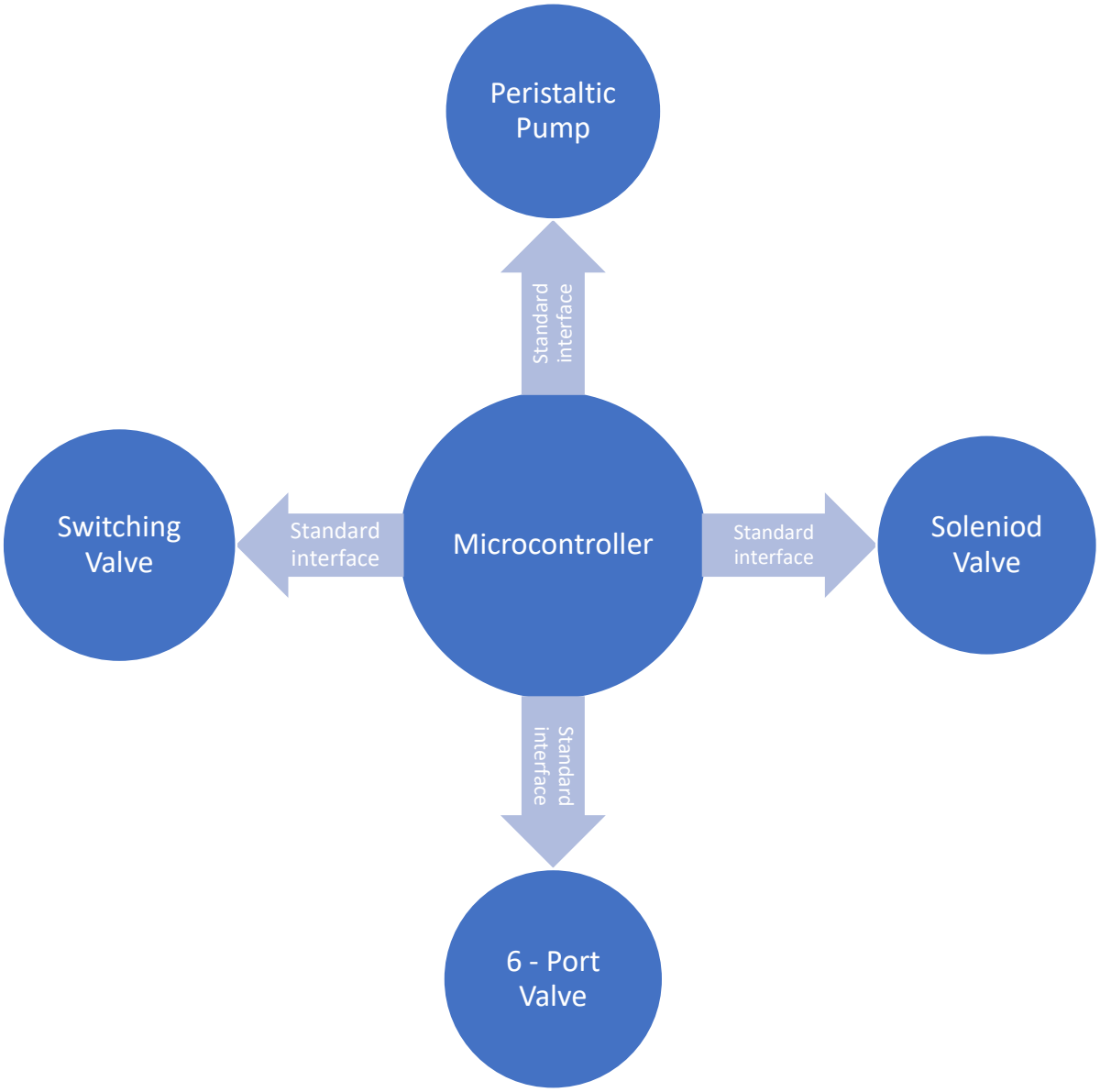

Figure 2: Concept 2: Using standard interfaces to control devices

Optioneering

Processor Optioneering

The STM32F429ZI microcontroller was selected as the processor for the system, based on the processor selection table shown below.

Table 1 contains a class weight column ranging from 0 – Unnecessary to 10 – Essential for each feature. Each processor is then individually scored from 0 – Poor to 5 - Best for each feature and multiplied by the class weight to get an overall feature score. The feature scores are then summed together, and the processor with the largest score is the one that is to be selected for the research.

Table 1: Processor comparison

Feature	Feature Weighting	STM32F429ZI		ATmega32u4		Raspberry Pi Zero	
		Weight	Score	Weight	Score	Weight	Score
Maximum Clock Speed	3	3	9	1	3	5	15
Maximum Storage (EEPROM)	2	3	6	2	4	5	10
Maximum Memory (RAM)	3	4	12	1	3	5	15
Number of ADC's	2	5	10	3	6	0	0
ADC Resolution	3	4	12	3	9	0	0
Number of Hardware Timers	5	5	25	2	10	1	5
Number of USART Interfaces	5	5	25	2	10	2	10
Number of I^2C Interfaces	2	4	8	2	4	2	4
Number of SPI Interfaces	2	5	10	2	4	2	4
Number of Onboard Ethernet Connectors	5	5	25	0	0	1	5
Development Board Cost[1]	3	1	3	2	6	5	15
Ease of use	3	3	9	5	15	6	18
Total		145		74		101	

Offerings from the leading manufacturers in the microcontroller industry, ARM and Microchip were considered. This includes the STM32F429ZI and Raspberry Pi Zero

[1] Cost referenced to the development board of each microcontroller at RS Components Ltd and Pimoroni Ltd. Prices correct as of 25/05/2020

powered by ARM cores, and the ATmega32u4 from Microchip, powered by an AVR core.

The primary concern for the processor was how many hardware timers are present and whether the system can operate in real-time, given that precise timing is a critical part of the research. Also, a built-in ethernet interface was desirable in order to build a web-based interface to control the system. The rationale for deciding on a web-based interface will be explained in the Web Interface Software Design section.

In addition, the processor used should support software threads with priorities so that the devices can be controlled at precise time intervals. A state machine could be used in the absence of threads, but they can simplify the microcontroller code significantly, making them a desirable feature.

Furthermore, support for a wide variety of hardware interfaces, such as USART, SPI and I^2C is desirable in case the web interface is impractical, meaning another method of communication can be used if required. Finally, a low cost was also a desirable factor so the device can be made as cheaply as possible.

The STM32F429ZI has ethernet support inbuilt into the microcontroller, with only a small amount of extra circuitry required to support an RJ45 jack connection weighing heavily in its favour. It also supports software threads, has a wide variety of additional interfaces, though at the highest cost.

The ATmega32u4 does not have inbuilt ethernet support, but an external SPI to ethernet adapter could be used in order to support an ethernet interface. Because it is an 8-bit device, the performance compared to the other 32-bit processors is significantly lower. Then again, it does benefit from comprehensive software support and is easy to program, with many libraries available, as it is used in the Arduino® Leonardo [2].

The Raspberry Pi Zero does not have an ethernet connector; however, due to the more powerful processor, as well as a USB OTG connector, it can emulate an ethernet interface over serial [3]. It supports threads and some additional interfaces but has only one hardware timer, which is a significant limitation for research that depends on precise timing.

Device Interface Optioneering

As discussed in the Background section, the purpose of the system is to interface with existing lab devices to control them precisely. Therefore, the controller must have the appropriate hardware to interface with the devices control system so that the devices can be controlled without damaging the processor.

The interface between the controller and the lab equipment should be as simple as possible to increase reliability and to keep the cost of the controller low. Some equipment, like a solenoid valve or peristaltic pump, only need one contact to supply power to the device to turn it on. Other devices, like switching valves, six-port valves, photomultiplier tubes, and photo spectrometers, have more complex interface requirements.

Switching Relays
Electrical relays were chosen to be the primary interface between the microcontroller and devices because they provide galvanic isolation between the device and microcontroller and can switch high voltages safely. They can be used to control solenoid valves, peristaltic pumps, switching valves and six-port valves.

If a typical routine switches each relay 35 times over the routine cycle, and that three routines are run per day, 175 cycles occur per day. Given that each relay switches 1 A of current and there is a unity power factor, the relay has a lifespan of 700,000 cycles [4]. This means that the relays should last for 4000 days, or just under 11 years, which is acceptable for the controller.

Device Inputs
Some devices such as a mass spectrometer, six-port and switching valves can assert a signal in the case of a fault [5]. An input interface circuit is needed to allow such inputs to be read by the microcontroller, so that the appropriate action can be taken. Also, the circuit needs to ensure that the microcontroller is protected from electrostatic discharge to prevent damage by using an optocoupler. The Inputs section describes the development of the interface circuit.

Interface Connectors
In order to maintain existing compatibility with existing lab devices, standard screw terminal connectors were chosen. Specialised connectors for each type of device were considered to assist with connecting the devices. However, this would have made the system less flexible to changing needs. Therefore, standard connectors were chosen to allow the controller to adapt to changing configurations readily. Figure 3 shows the interface connector chosen.

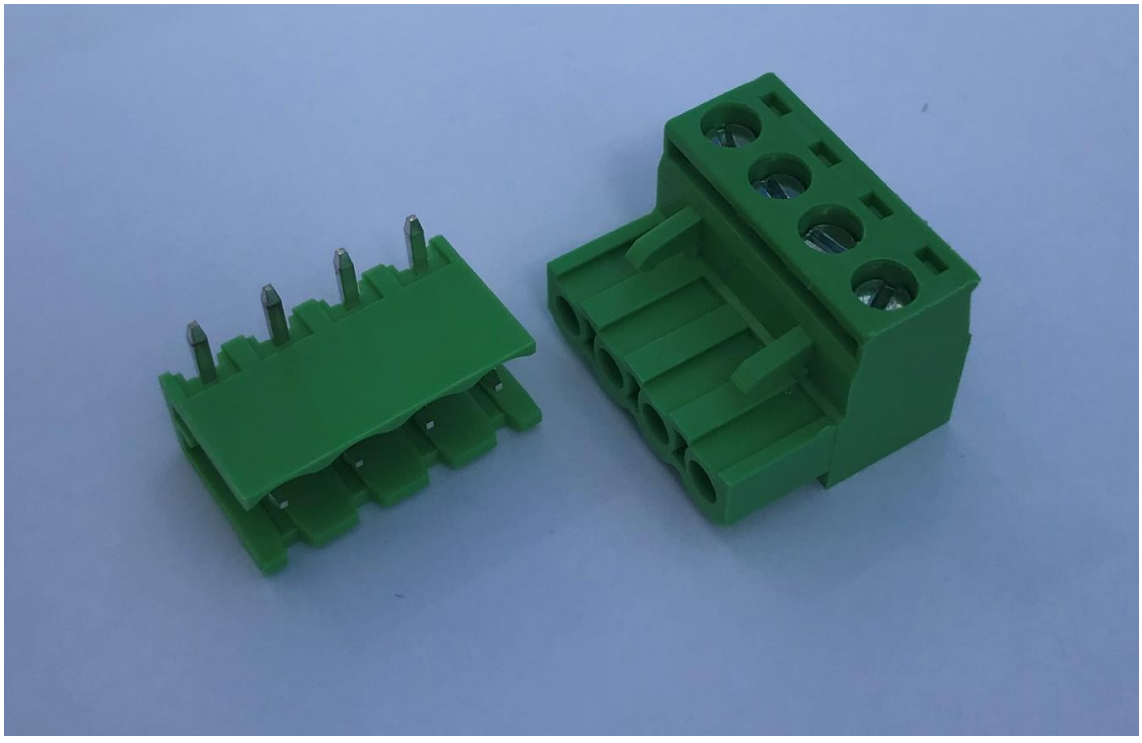

Figure 3: A standard interface connector

Photomultiplier Tube

A photomultiplier tube measures the amount of light generated by a chemical reaction and can be used to determine when it is complete. Figure 4 shows an example of a photomultiplier tube that could be used with the system. To operate, it requires a high input voltage of 1.1 kV, meaning that a suitable input conversion circuit is required to interface with the microcontroller.

The circuit could convert the signal into a digital form so that it can be processed by the microcontroller digitally. Alternatively, the circuit needs to convert the signal into a range of 0 v to 3.3 v so that the onboard ADC can sample the signal to digitise it. Unfortunately, the author could not access a photomultiplier tube during development, so the physical implementation of this part was not able to be completed.

Figure 4: A photomultiplier tube

Research Enclosure

The research enclosure was chosen to ensure that it would be able to withstand the conditions in the lab. Dr Antony Birchill was consulted to ensure that the enclosure would be suitable. The enclosure is made from ABS material to resist corrosion from chemicals used in the lab, while the rubber feet prevent damage from any spilt

chemical. Also, the size was essential to ensure that it could hold all the electronics required. The removable end panels are a useful feature to aid assembly.

For this research, the maximum control voltage is 24 v. This means that a case capable of withstanding Band 1 signals, less than 50 v is suitable.

Microcontroller Software Design

C/C++ is the language used for the control software on the microcontroller. Keil is the IDE used to develop the code for this research. Also, the STM32 HAL, as well as mbed-os, is used to reduce the development time by taking care of the microcontroller-specific features and providing a consistent interface to perform standard functions.

The use of mbed-os comes with a performance trade-off as well as increasing the size of the compiled program, requiring a larger amount of ROM on the microcontroller to store the program. However, due to the high-performance microcontroller used, and the fact that there are no power limitations in the system, the benefits of mbed-os such as threading and the reduction in development time, particularly with the HTTP server, outweigh the performance costs of using it for this research.

mbed-os version 5.15 was chosen to be used throughout the research. The entire library for the STM32F429ZI microcontroller was exported to Keil from the online compiler [6]. Keil provides more comprehensive debugging features than the online compiler, such as the ability to single-step through lines of code to identify software bugs. Also, the use of the online compiler was impractical due to a limited internet connection. The use of the new mbed ide [7] was considered but ruled out due to the software still being in beta at the time of starting the software development.

Software device class hierarchy

To provide a consistent interface and to simplify the management of devices, a base device class was created. The base device class defines the primary methods that can be used with a device. The child classes then specialise these for the specific requirements of the device. Figure 5 shows the relationship between the classes.

The class constructor takes the names of the control pin(s) used for the device as well as a unique ID that can later be used to control each device individually.

The class also provides an internal state variable that can be adjusted through an interrupt-safe API to keep track of the state of the device.

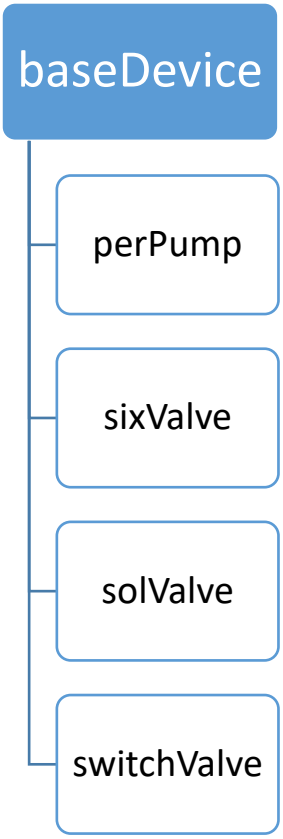

Figure 5: The device class hierarchy

Pin Name Abstraction

In order to simplify the connection and configuration of devices, the microcontroller-specific pin names are abstracted. Instead, a simple numbering system is used when configuring the system. This has been accomplished by using an array to hold the pin names for the digital inputs and outputs. It also means that additional device interfaces can be added easily by adding more pin names to the array. The interface numbers are simply the position of the pin name in the array, starting at one so as not to confuse non-technical users who might find an interface zero a bit strange.

Code 1 shows the definition of two arrays of pin names used for digital inputs and outputs. Here there are eight digital outputs and four digital inputs.

```
//Define an array to hold the pins used for the digital
outputs

const array<PinName, 8> digitalOutputs = {PF_13, PE_9, PE_11,
PF_14, PE_13, PF_15, PG_14, PG_9};


//Define an array to hold the pins used for the digital inputs

const array<PinName, 4> digitalInputs = {PA_7, PD_14, PD_15,
PF_12};
```

Code 1: The pin name definitions

Base Device Array

Due to the input and output pins being known at compile-time, as described in the section on Hardware Design, the maximum number of devices on the system can be calculated. This is possible since each device requires at least one input or output pin in order to operate. Code 2 shows the calculation of the maximum number of devices based on the number of inputs and outputs.

The number of actual devices on the system may be less than this due to some requiring more than one pin but cannot be more. This calculation allows an array to be created to hold all of the possible device objects. An array was used rather than a vector to prevent memory fragmentation that would be caused by having a variable-length structure to hold the device objects.

```
//Calculate the number of digital inputs and outputs defined

const short maxDevices = digitalOutputs.size() +
digitalInputs.size();
```

Code 2: The maximum number of devices calculation based on the number of inputs and outputs defined

The array is initialised in an empty state, as shown in Code 3. It is populated according to the configuration specified by the device configuration file, as explained in the Device Configuration JSON section.

```
//Create an array of baseDevice* which will be populated at
run-time by calling configDevices()

array<baseDevice*, maxDevices> devices = {};
```

Code 3: The creation of the devices array

File Format

JSON stands for JavaScript Object Notation and is the file format used to hold the routines and device information in the system. It is comprised of ASCII characters that can easily be parsed into JavaScript objects. Data is stored in name-value pairs and is separated by commas, with curly brackets, { } denoting objects and square brackets, [ ] denoting arrays [8].

The controller configuration for devices and routines is stored in a JSON file format in order to pass the information between the microcontroller and web interface easily. Name-value pairs allow the code to be more readable, assisting with debugging but with a significant performance cost due to the data needing to be held in C++ strings for parsing. However, this performance cost only occurs when the device or routine configuration is updated and needs to be parsed. Once the parsing is complete, the performance returns to normal as the data is stored in C++ structures that use fewer resources.

An alternative file format such as CSV, Comma Separated Value file format was considered due to superior performance. However, this would mean that a custom parser would be needed to extract the information from the file, which would need extensive testing to ensure that it would cope with the variety of information in the file. Besides, the file would also need extra parsing on the web interface side, in order to display device and routines information. Furthermore, if the file needs to contain additional information in the future, such as a new device type requiring three pins to operate, more data can be added without having to change the existing parser. This is possible due to the name-value pairs in JSON, which is possible in a format such as CSV but is more complex to implement. The name-value pairs mean extra elements can be ignored for the existing devices that do not need them.

In order to parse the JSON on the microcontroller, the JSON parser library written by Samuel Mokrani was used [9]. Writing a new parser library was considered but dismissed due to the development time and complexity.

Device Configuration JSON

Figure 6 shows the devices file structure, consisting of an array of device objects, each containing the required properties for the device. This includes a unique ID, the name of the device, the type of device and the interface pins used.

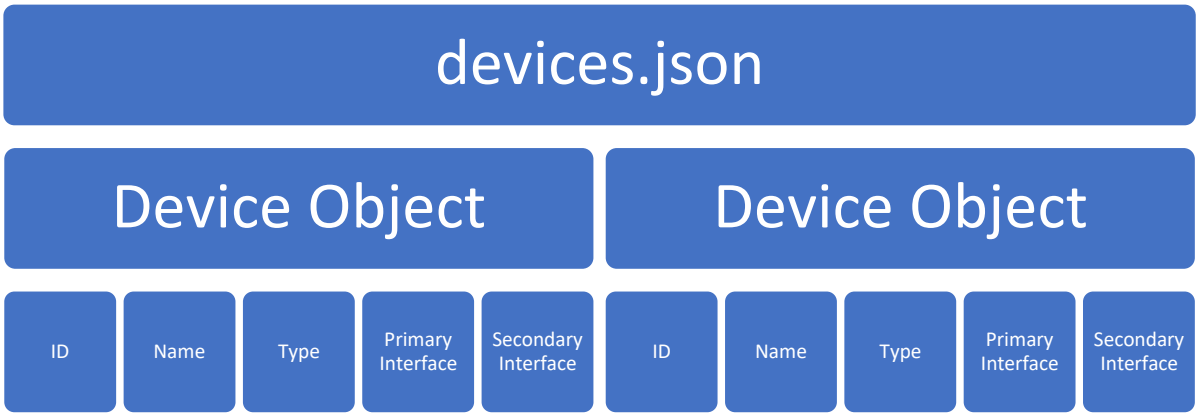

Figure 6: The devices.json file structure

The code in Section 1 of the supplementary file parses the JSON file and extracts the required information. It then creates the appropriate device objects in the base device array, described in the Base Device Array section, and initialises them ready for use.

Timing Structures

In order to store routine configuration data, a new structure was created. This comprises the device ID, start time, stop time and the state of the device. The timings are that the device starts at the start time and stops at the start time of the next timing block, or the stop time at the end of the routine. Start time and duration were considered parameters for the timings, but this would have made it more

challenging to check for gaps in the timing information, leading to the devices being in an undefined state. Also, the duration can be easily calculated by taking the start time from the stop time.

Code 4: shows the definition of the timing structure.

```
typedef struct {
    uint16_t devID;
    uint16_t startTime;
    uint16_t stopTime;
    uint16_t devState;
} deviceTimes;
```

Code 4: The structure to hold the routine timing information

Figure 7 shows a visual representation of the timing structure, containing the device ID, start time, stop time and state.

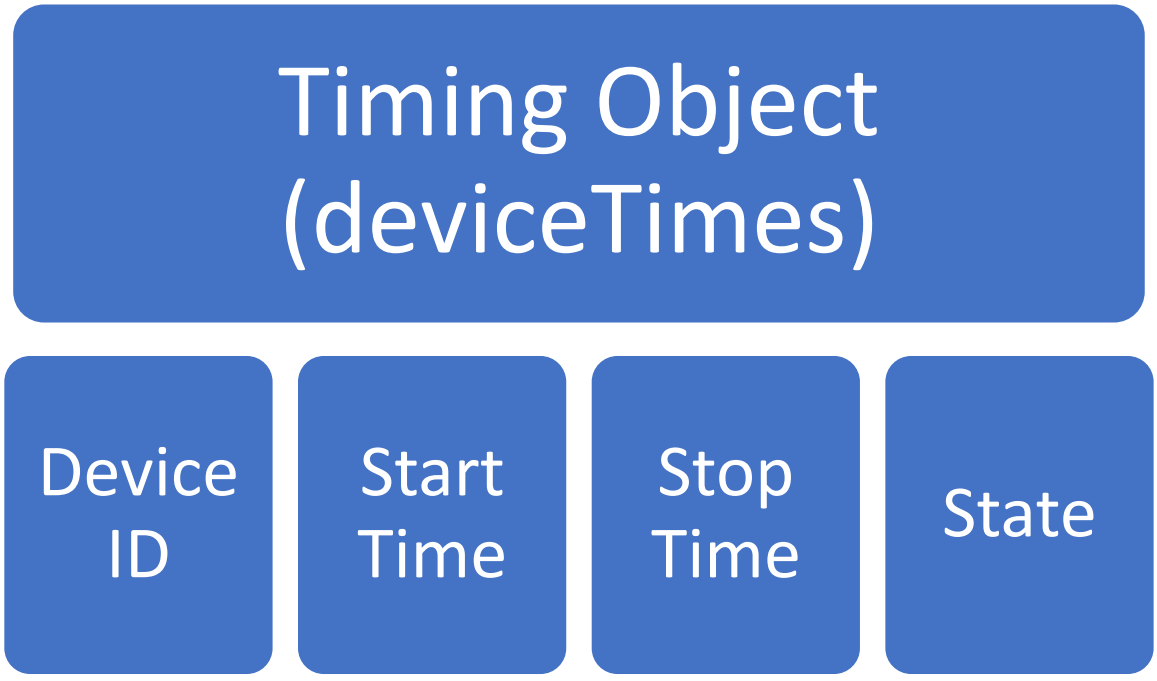

Figure 7: The timing object structure

Directly parsing the JSON was considered to reduce the timing system's memory requirements, but, as discussed in the File Format section, there is a performance impact when parsing JSON, which might affect the timing performance. When the JSON is parsed, it can be checked for errors to ensure that all the required data is

present before the routine is run. The pre-checking ensures that there are no syntax errors with the routine, preventing wasted effort.

Due to the reasons above, a C++ standard vector is used to hold the variable-length routine information. An array could be used, but as its length cannot be changed after it is initialised, it would have to be initialised to a size larger than the most extensive routine possible, consuming valuable memory that may not be used.

Code 5 shows the initialisation of the routine vector. It is populated according to the configuration specified by the routine configuration file, as explained in the next section.

```
//Create a vector to store device times for the routine
std::vector<deviceTimes> routine;
```

Code 5: The routine vector

Routine Configuration JSON

Figure 8 shows the routine configuration file structure, consisting of an array of routine objects. Each routine object contains an ID, Name and an array of timing information, as described in the Timing Structures section.

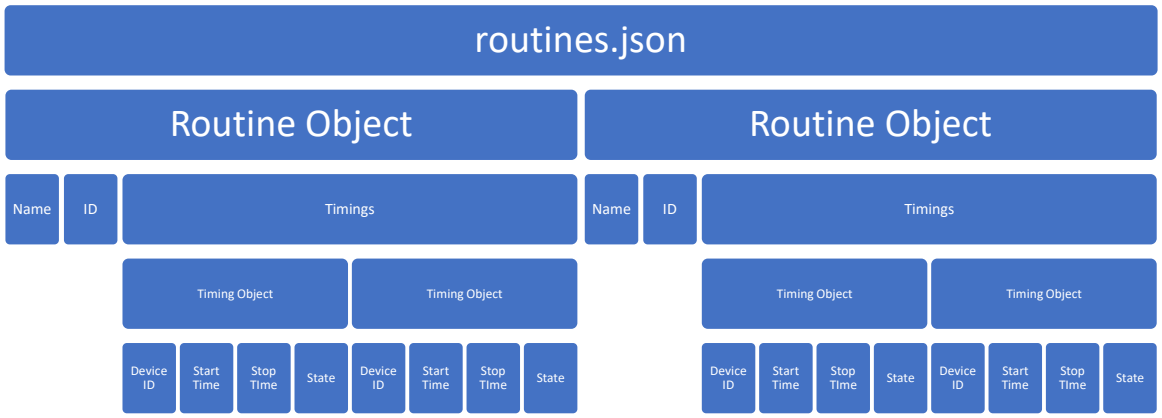

Figure 8: The routines.json file structure

The code in Section 2 of the supplementary file parses the JSON file and extracts the required information. It then inserts the appropriate timing objects in the routines vector, described in the previous section, and initialises them ready for use.

Routine Timing

Precise timing is critical in this research to control devices at precise time intervals to ensure accurate and repeatable results are obtained. The current time resolution of

the system is one second, which could be increased to 100 ms if needed for short routines.

The relays used can open and close their contacts within 10 ms. However, this is the maximum switching speed and will shorten the lifetime of the relays. Also, mechanical lab devices require time to change from one state to the next, making any greater time precision redundant. If there is a requirement for increased precision in the switching time of devices, a different interface such as a MOSFET would need to be considered.

In order to ensure that the timings are precise, a mbed-os EventQueue running on the highest priority thread is used. EventQueues [10] are extremely useful as they allow for scheduling of events at precise time intervals, using a hardware timer for accuracy.

Routine Operation

In order to run a routine, the previously parsed and stored data has to be read and used to control the connected devices.

Code 6 shows a function to run a routine in a blocking way, which is useful for testing the operation of it. The function initialises a local variable to zero and then determines the length of the routine, as described in Section 3 of the supplementary file. It then enters a while loop, checking each second if there is a device that needs to change state. Once the routine has finished, the devices are reset, and the function returns.

However, there are severe limitations to this approach, including a loss of timing precision if the thread calling the function is not the highest priority. The fact that it is a blocking function also means that thread starvation can occur if it is run on the highest priority thread. Furthermore, due to the non-deterministic nature of the code preceding the thread sleep command, there will be a further loss of timing precision, as the delay does not take into consideration the execution time of the preceding code.

```
void runBlockingRoutine(void) {

    //Seconds since starting routine
    uint16_t elapsed = 0;

    //Get routine duration
    uint16_t duration = routineDuration();

    //While the routine has not finished
    while (elapsed < duration) {
        //Loop through routine and change state if required
        for (deviceTimes n : routine) {
            //If a device needs to change state
            if (elapsed == n.startTime) {
                //Loop through the devices array
                for (int i = 0; i < devices.size(); i++) {
                    //If the device ID matches the specified ID
                    if (n.devID == devices[i]->getID()) {
                        //Change the state of the device
                        devices[i]->changeState(n.devState);
                    }
                }
            }
        }
        //Increment elapsed
        elapsed++;

        //Delay for a second before starting again
        thread_sleep_for(1000);
    }

    //Reset devies to default state
    resetRoutineDevices();
}
```

Code 6: The blocking function to run a routine

To overcome the issues with running the routine in a blocking function, an EventQueue is utilised, as shown in Code 7. The routine event queue runs on the highest priority thread, in order to ensure the timing is precise. The function is similar to the blocking one above, except that now it is called deterministically by the event queue, improving timing precision.

The calling function uses the function in Section 3 of the supplementary file to calculate the duration of the routine after it is loaded and then calls the run routine function on the event queue every second. The routine then runs like before, except once it has finished, the function removes itself from the event queue using the ID obtained when it was put on to the event queue.

```
//Routine Thread
Thread routineThread(osPriorityRealtime);

//Create eventqueue
EventQueue routineQueue;

//startRoutine - Responsible for running the routine EventQueue
void startRoutine(void) {
    //Start event queue on thread
    routineQueue.dispatch();
}

//Seconds since starting routine
uint16_t routineElapsed = 0;

//Routine duration in seconds
uint16_t duration = 0;

//ID of the EventQueue task generated by calling runRoutine every second
int routineEventQueueID = 0;

//Get routine duration
uint16_t duration = routineDuration();
```

```
void runRoutine(void) {
    //If the routine has not finished
    if (elapsed < duration) {
        //Loop through routine and change state if required
        for (deviceTimes n : routine) {
            //If a device needs to change state
            if (elapsed == n.startTime) {
                //Loop through the devices array
                for (int i = 0; i < devices.size(); i++) {
                    //If the device ID matches the specified ID
                    if (n.devID == devices[i]->getID()) {
                        //Change the state of the device
                        devices[i]->changeState(n.devState);
                    }
                }
            }
        }

        //Increment elapsed time
        routineElapsed++;

    }
    else {
        //Reset devies to default state
        resetRoutineDevices();

        //Stop the function from being called again as the routine
has finished
        routineQueue.cancel(routineEventQueueID);

        //Reset the eventQueueID
       routineEventQueueID = 0;
    }
}
```

Code 7: The event queue way of running a routine

Webserver

In order to interact with the system, a web-based interface is used. To achieve this, an HTTP server runs on the microcontroller to handle the requests from the client. The client-side web interface development is discussed in the Web Interface Software Design section.

To ensure that the webserver is responsive and does not impact other tasks, it runs in a separate thread with a normal priority. The thread consists of an infinite loop, in which the server listens for GET requests from the client and takes the appropriate action in response to them.

Section 5 of the supplementary file details the reconfiguration of the microcontroller in response to the user interface changing the device configuration. As noted in the Web Interface Software Design section, the web client has more processing power than the microcontroller and so is responsible for creating the device and routine configuration files. These are then sent to the microcontroller, which then reconfigures to the new configuration.

Sending requests to the microcontroller to change the configuration was considered and is definitely possible, but with the frequent requests to change the device or routines and the input validation needed, poor performance was observed. Due to this, all of the device and routine manipulation and validation happens on the web client. The complete configuration file is then sent to the microcontroller, which reconfigures to match the new file.

Device Status Monitoring

The device status monitoring system will track the state of all of the devices and ensure that they are all operating normally. It will also have the ability to shut down the system if a fault is detected, to protect the lab equipment.

The priority device for monitoring is solenoid valves, as it is challenging to determine manually if they are working or not. Next are the Switching and Six-Port valves, as, during testing, a controller failed, and a significant amount of time was spent trying to determine why the valve was no responding to commands. Finally, the peristaltic pump is the easiest to test, though monitoring would be useful to ensure that that it has genuinely stopped if the emergency stop is activated.

To do this, a monitoring circuit will need to measure the critical parameters of the lab equipment. The system can then notify the user and shut down the system if a device is operating outside normal parameters.

The Solenoid Valve Monitoring section discusses a circuit to monitor the voltage and current of a solenoid valve. Two analogue input pins can be used to monitor the current and voltage. An additional function could be added to the base device class to facilitate the configuration and reading of the values. A separate thread could then continuously measure and monitor the device and shut down the system if a fault is detected.

Web Interface Software Design

Introduction

The goal of using a web-based interface is to reduce the initial setup time of the system, as nearly all modern computers have a web browser installed. Also, users are familiar with web interfaces and so should find it more intuitive than using a custom piece of software that has to be installed. In addition, web browsers are actively maintained and generally well supported.

To develop the interface, modern web techniques along with HTML CSS and JavaScript code were used. The interface communicates with the HTTP server running on the microcontroller to service the requests from the user, as discussed in the Webserver section previously.

Typically, web servers are more powerful than the client, meaning that they handle the heavy processing work, for example, running databases or storing files. However, in this case, the microcontroller has limited processing power, and the webserver thread is a low priority, to ensure precise routine timing, as discussed in the Routine Operation section previously.

As the client computer is more powerful than the microcontroller, it is better suited for creating and validating device and routine configurations that are then sent to the microcontroller to be run. Writing a custom database function to manage the devices and routines on the microcontroller was considered but dismissed due to development time and complexity as well as poor performance during initial experiments.

The jQuery JavaScript library was used to assist with adding interactivity to the web interface. The library has useful features such as the “each” function for looping through data, and the “getJSON” function for getting JSON data from a web server. The interface could be written without using this library, using pure JavaScript. The main challenge is the manipulation of JSON for the devices and routines. A future improvement to the system would be the removal of the jQuery library to improve performance as many functions of the library are not used but still stored on the microcontroller.

As noted in the Future Development section, an improvement to the system would be the addition of a single board computer such as a Raspberry Pi. This would then host the webserver and improve the user experience by having additional processing power for data analysis. The single-board computer could and act as the webserver while the microcontroller board handles all of the real-time tasks. Serial communications over USART could then control the board.

To assist with understanding the web interface, there is a demonstration video, https://www.youtube.com/watch?v=7ggpQW3ySN4. The demonstration starts at 2:50.

Home Page

Figure 9 shows the homepage upon first loading the web interface. The top bar allows for navigation, while the rest of the page allows for routines to be selected, tested and run.

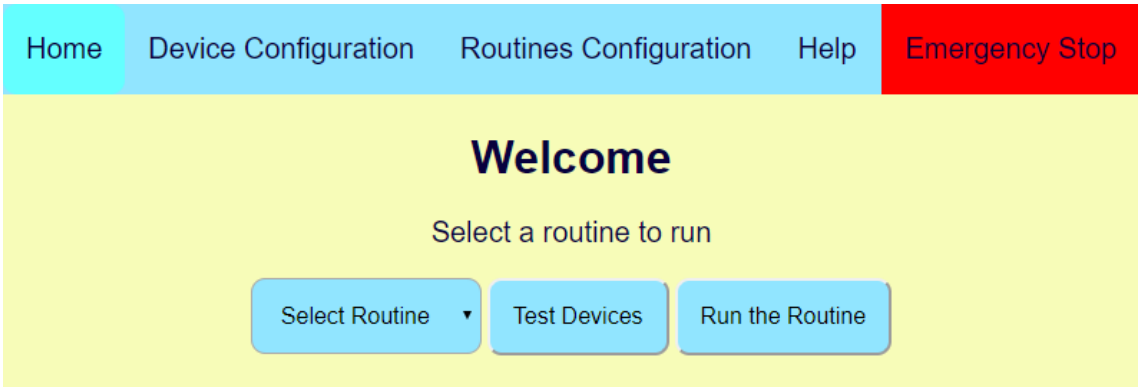

Figure 9: The web interface homepage

Selecting Routines

The routine to be run can be selected using the dropdown box populated from the routines.json file described in the Routine Configuration JSON section. The routine names are put into the dropdown for selection, and the ID is added for reference. Code 8 shows the function used to populate the dropdown. After ensuring that it is empty, the code inserts a default option called “Select Routine” to prompt the user to select a routine. It then gets the routine JSON file from the microcontroller, parses it, and then appends the created HTML to the dropdown to display the routines to the user.

Testing Routine Devices

After a routine has been selected, the “Test Devices” button tests all the devices used in the routine to ensure that they are working correctly. Once the button is pressed, the devices are tested sequentially. Upon finishing, a popup appears informing the user that the devices were all successfully tested, as shown in Figure 10 or that there was an error, as demonstrated in Figure 11. At the moment, the testing is manual, with the user confirming the correct operation of the devices. Future improvements include integrating with the device monitoring system discussed in the Device Status Monitoring section so that the testing process can be automated.

```
function populateRoutines(ddID) {
    //Ensure that the dropdown is empty
    $('#' + ddID).empty();

    //Set default option as select device, and make it disabled
    $('#' + ddID).append('<option selected="true" disabled>Select
Routine</option>');
    $('#' + ddID).prop('selectedIndex', 0);

    //Define a temporary variable to hold the HTML
    var opHTML = '';

    //Get the routine information
    let routines = getRoutines();

    //Parse the JSON
    var response = $.parseJSON(routines.json);

    //Loop through the response and fill in the dropdown
    $.each(response, function (i, item) {
        //Generate the dropdown html
        opHTML += '<option class=' + item.routineID + '>' +
item.name + '</option>';
    });

    //Append the html to the dropdown
    $('#' + ddID).append(opHTML);
}
```

Code 8: The JavaScript to populate the dropdown

Figure 10: A successful device test

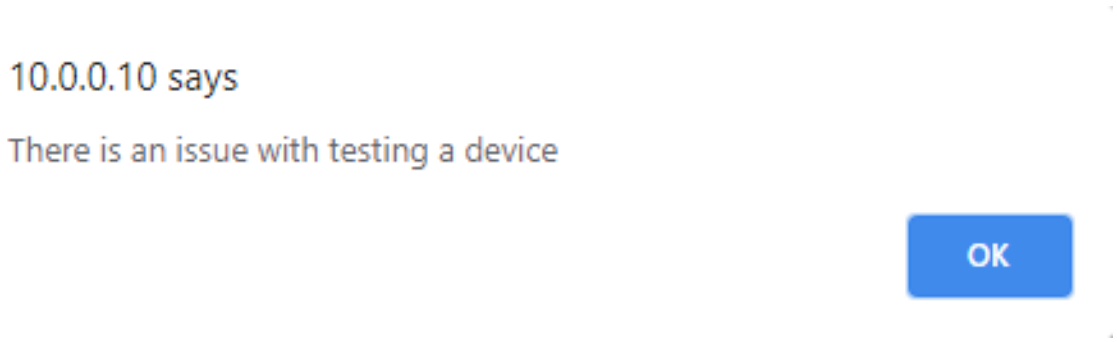

Figure 11: An unsuccessful device test

The code in Section 4 of the supplementary file runs when the test button is pressed. It gets the routine ID and then sends this to the server to test the devices. It then awaits a response indicating that all devices were tested, or that an error occurred. A popup is then generated to inform the user.

Running a Routine

The “Run Routine” button works in a very similar way to the “Test Devices” button described in the previous section, except that it starts the selected routine running rather than testing the configured devices.

Device Configuration

Figure 12 shows the device configuration page of the system. The navigation bar is still at the top, while the main page shows all the currently configured devices and provides options for adding, updating and deleting them.

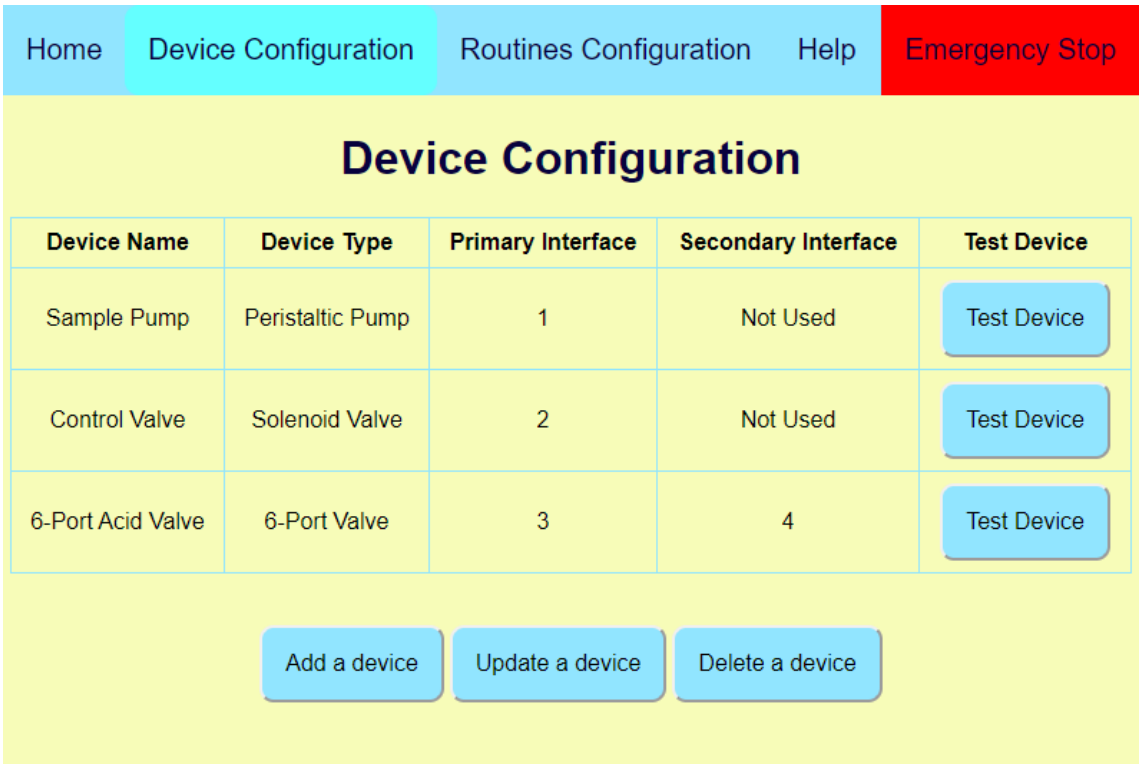

Figure 12: The device configuration page showing three configured devices

Viewing Devices
As shown in Figure 12, the device configuration page shows all of the currently configured devices. The name, type and interfaces used by the device are all shown.

Testing Devices
As shown in Figure 12, each configured device has a test button to allow its operation to be tested quickly. Upon clicking on a test button, a popup will appear, asking which state the device should be changed to. Once the option is selected, the device will change to the requested state, similar to the routine device test described in the previous section.

Adding Devices
Upon clicking the “Add a device” button, a form pops up to enter the required information, as shown in Figure 13. Once the fields have been populated, the “Add device” button can be clicked to add the device to the system.

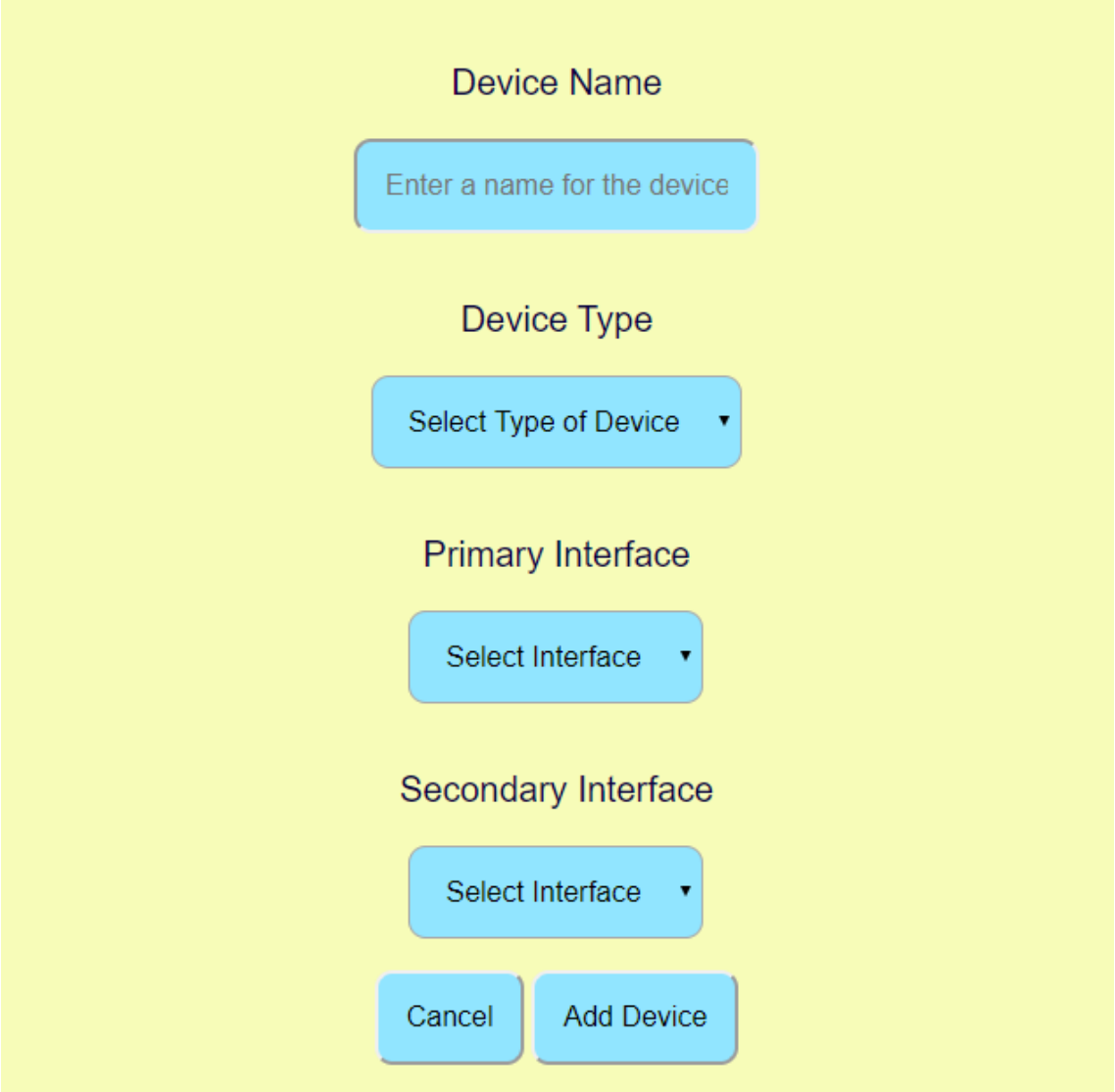

Figure 13: The form to add a device

As discussed in the Web Interface Software Design section, the client is more suitable for managing the configuration than the server, so an Indexed DB is used to hold the device and routine configuration information. Upon the configuration changing, the database is exported in a JSON format and sent to the microcontroller. The microcontroller then reconfigures to match the updated configuration.

Code 9 shows the function to add a device to the system, while the code in Section 5 of the supplementary file shows the functions to send the configuration to the microcontroller and reconfigure it to match the updated configuration.

```
function addDevice(formObject) {

    //Get the form data, validity checked by HTML5 required
attribute and range restrictions when the form is submitted
    let newDevice = [
        {
            devName: formObject.devName.value,
            devType: formObject.devType.value,
            devPin1: formObject.priInter.value,
            devPin2: formObject.secInter.value
        }
    ];

    //Start a database transaction
    let transaction = db.transaction(["devices"], "readwrite");

    //Start an object store request
    let objectStore = transaction.objectStore("devices");

    //Add the device to the database
    let objectStoreRequest = objectStore.add(newDevice[0]);

    //Update board configuration
    updateDeviceConfig();
}
```

Code 9: The function to add a new device to the system

Deleting Devices

Clicking on the “Delete a Device” button brings up the form shown in Figure 14. It consists of a dropdown to select the device to delete, and then a button to confirm the deletion.

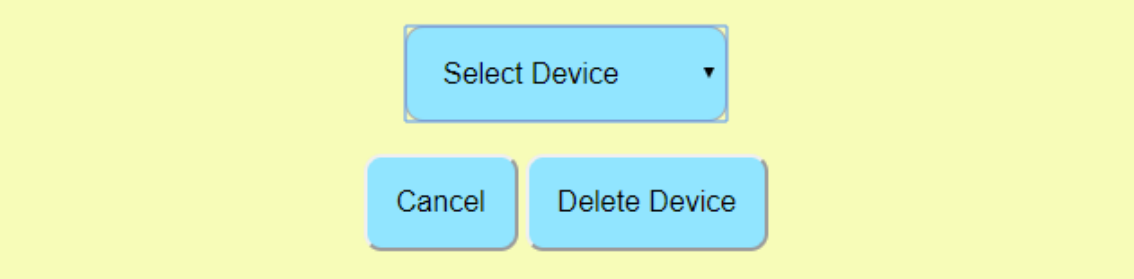

Figure 14: The form to delete a device

Code 10 shows the function to delete a device from the system; the only difference to adding a device is the fact that a record is removed from the database.

```
function deleteDevice(formObject) {

    //Get the dropdown data which contains the ID of the
device to delete
    let delDevice = formObject.devID.value

    //Start a database transaction
    let transaction = db.transaction(["devices"],
"readwrite");

    //Start an object store request
    let objectStore = transaction.objectStore("devices");

    //Add the device to the database
    let objectStoreRequest = objectStore.delete(delDevice);

    //Update board configuration
    updateDeviceConfig();
}
```

Code 10: The function to delete a device from the system

Routine Configuration

Figure 15 shows the routines configuration interface, with the navigation bar at the top as always. There is a dropdown to select a routine to visualise, and buttons to

add or remove a routine. Finally, the bottom row allows for the configuration of device timings.

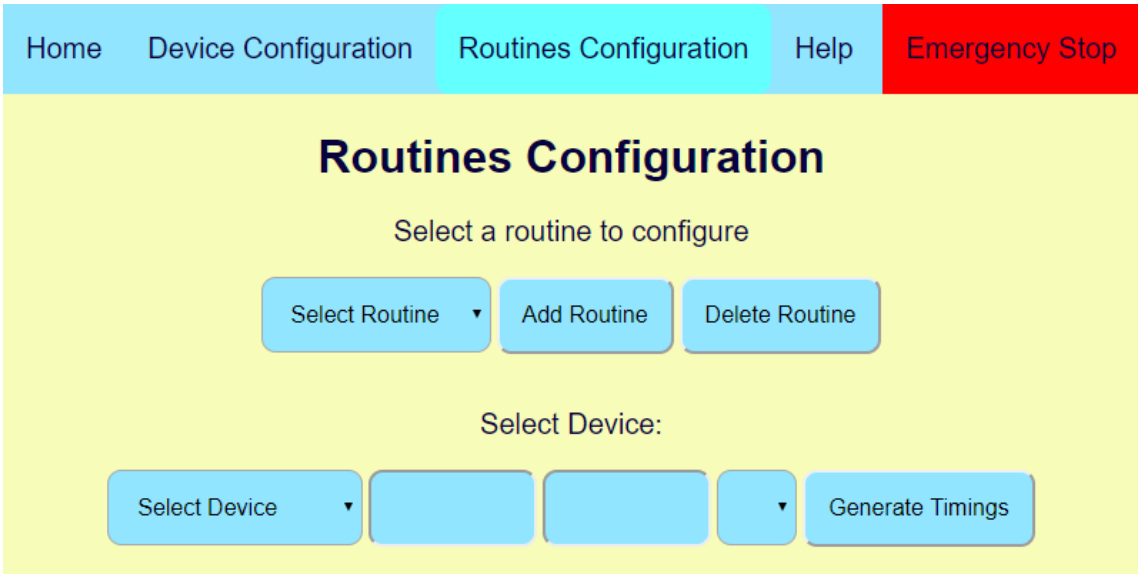

Figure 15: The routines configuration interface

Visualising Routines

By selecting a routine from the dropdown box, it will be loaded for editing and visualised on-screen, as shown in Figure 15. The visualisation provides a clear overview of the state of all devices during a routine and simplifies the process of debugging. It also helps to catch simple errors, such as a pump being turned on when a control valve is closed. This prevents damage to equipment and prevents wasted effort.

The horizontal bar chart by Richard Ramsay [11] inspired the look of the visualisation. The visualisation consists of the name of the device used in the routine, followed by a row of HTML span elements [12]. The span elements display the state of the device and are colour coded for easy identification. Also, if the mouse hovers over them, the tooltip displays the precise timings that the device will be in the state, as shown in Figure 17.

The code in Section 6 of the supplementary file shows the function to generate the routine visualisation HTML, given the routines JSON as a parameter. After parsing the JSON, the function sorts the timings array by start time [13] and then creates a list of unique devices. It then gets the name of the devices, before looping through the timings array and returning the generated HTML.

Editing Routines

Once a routine has been selected, it can be edited by using the inputs shown at the bottom of Figure 16. This allows a device to be selected, the timing defined, and the state selected. Upon clicking the “Generate Timings” button, the timing information is added to the routine. The visualisation can be refreshed to reflect the updated timings. Although this works reasonably well, as shown in the video demonstration, it is not particularly intuitive. An improvement could be to add a right-click menu to the

timing visualisation, to add a timing step. Also, the visualisation could be made resizable by dragging the edges, which would be a more natural way to edit routines.

Adding Routines

Adding a routine is very similar to the process of adding a device, described in the section Adding Devices. In this case, only the name needs to be specified when creating a routine, with the timings added to the routine by using the inputs detailed in the previous section. As mentioned before, once any changes are made, the database updates and the updated configuration is sent to the microcontroller.

Deleting Routines

Again, deleting a routine is very similar to the process of deleting a device, described in the Deleting Devices section. As shown before, the routine needs to be selected before pressing the delete button. The board then receives the new configuration.

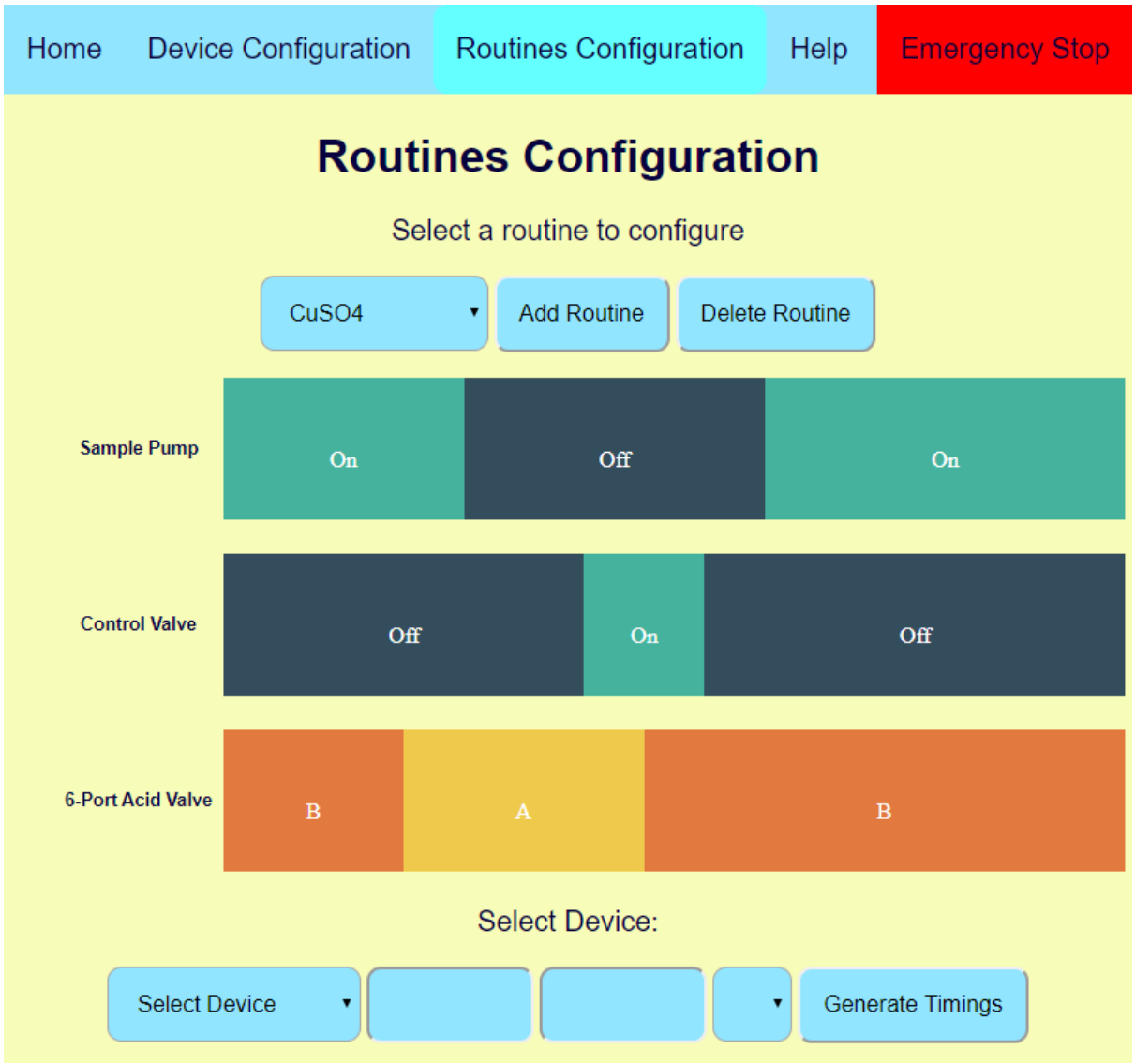

Figure 16: A routine visualised

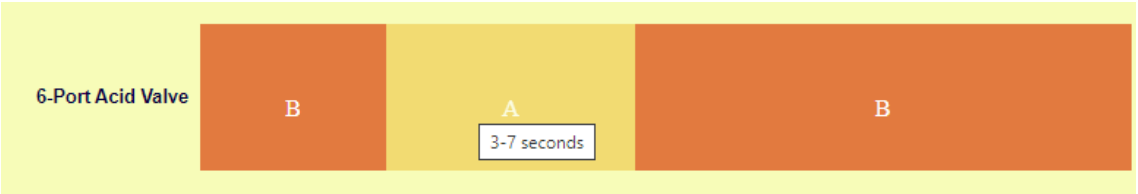

Figure 17: The tooltip showing the timing of the device

Emergency Stop

To stop all of the devices in an emergency, an emergency stop button is located on the top navigation bar. Figure 18 shows the top navigation bar, with the emergency stop button located on the bar's far-right. It is coloured red to stand out in what is likely to be a stressful situation such as a chemical leak, spill or equipment malfunction. Once pressed, all devices are stopped immediately.

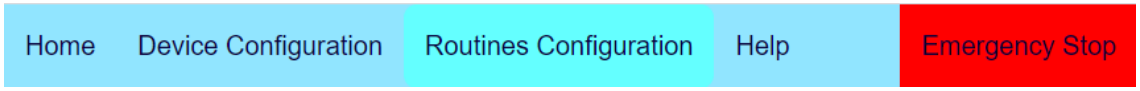

Figure 18: Navigation bar

Hardware Design

Relays

The relay interface boards break out three pins each, + 3.3 v power, ground and signal. All of the power and ground connections are wired together, as shown in Figure 19. The signal wires each connect to a separate GPIO pin on the microcontroller, allowing each to be controlled individually.

Currently, the relays are located within a 3D printed PLA enclosure to allow them to be tested with LEDs. While this is suitable for testing, it is not acceptable in a finished product. As discussed in the Future Development section, the production of a PCB would significantly enhance the performance and reliability of the research by eliminating the majority of the connections and flexible wire that is prone to coming loose. Unfortunately, due to a lockdown, importing a PCB board was not possible at the time of completing the research.

Inputs

As discussed in the Future Development section, the ability of lab devices to signal an action from the controller would significantly enhance the research. It would also allow for the detection of errors in equipment, improving the safety of the system.

Figure **20** shows the schematic for such a circuit, allowing an external input to interface with the system through an optocoupler to avoid potential electrostatic damage to the microcontroller.

Figure 21 and Figure 22 show the device input circuit built on a breadboard, with an LED to show the state of the input.

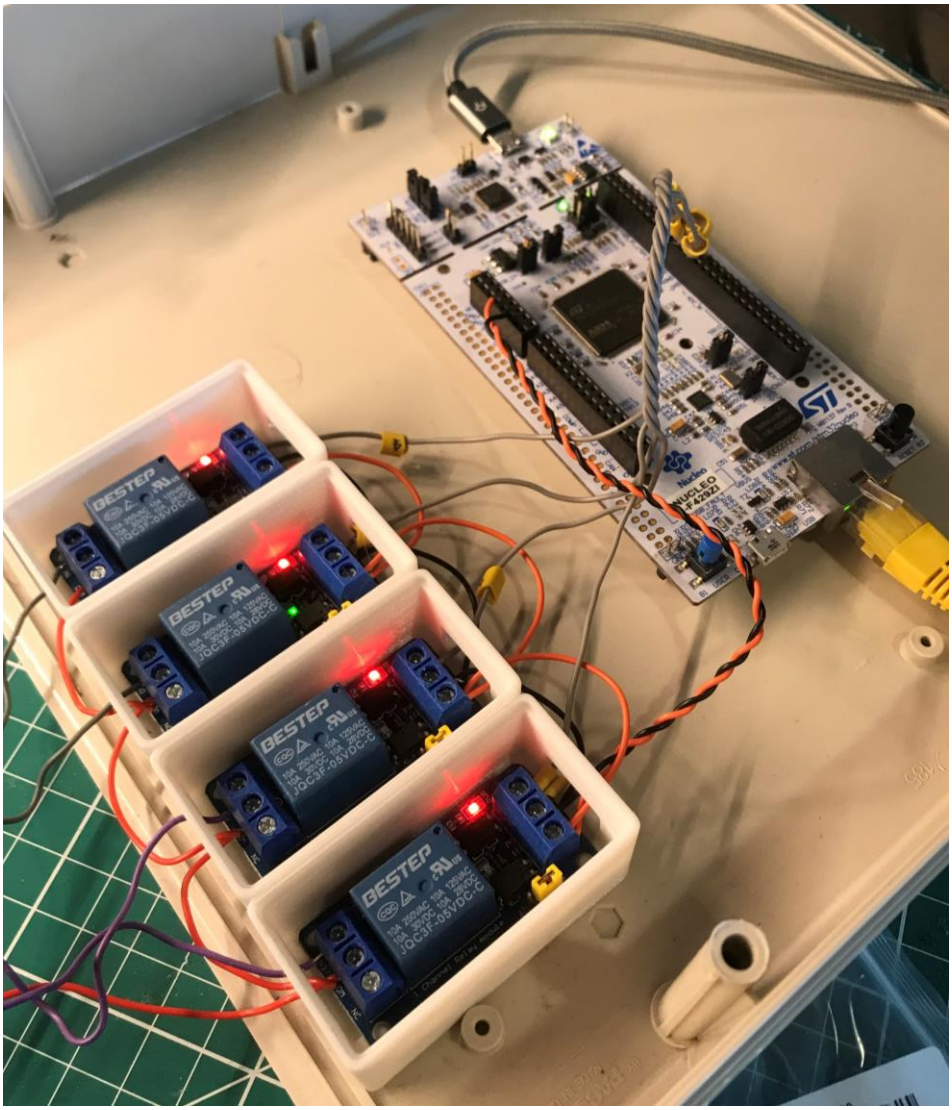

Figure 19: The relays connected to the microcontroller

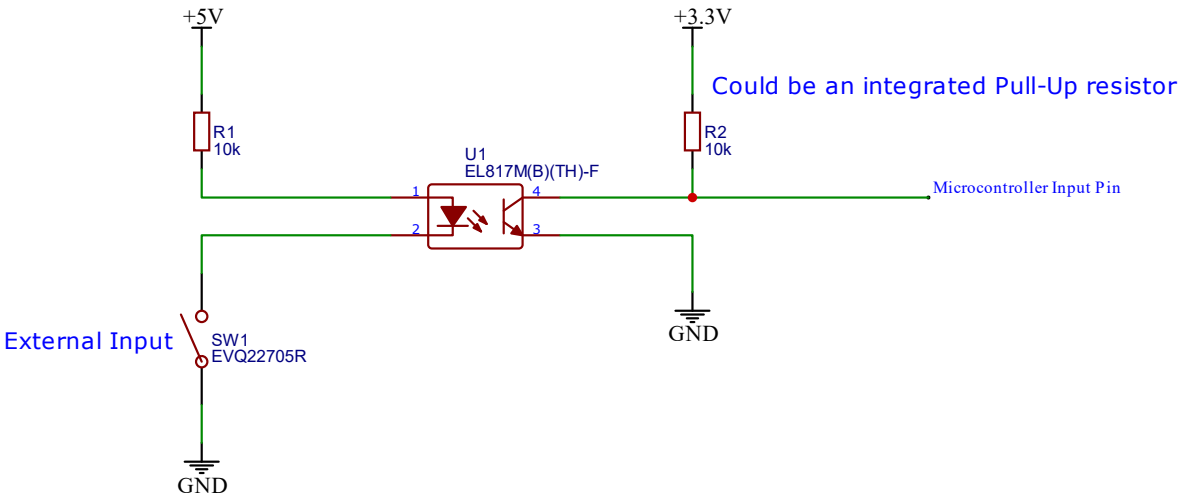

Figure 20: The device input circuit with optocoupler

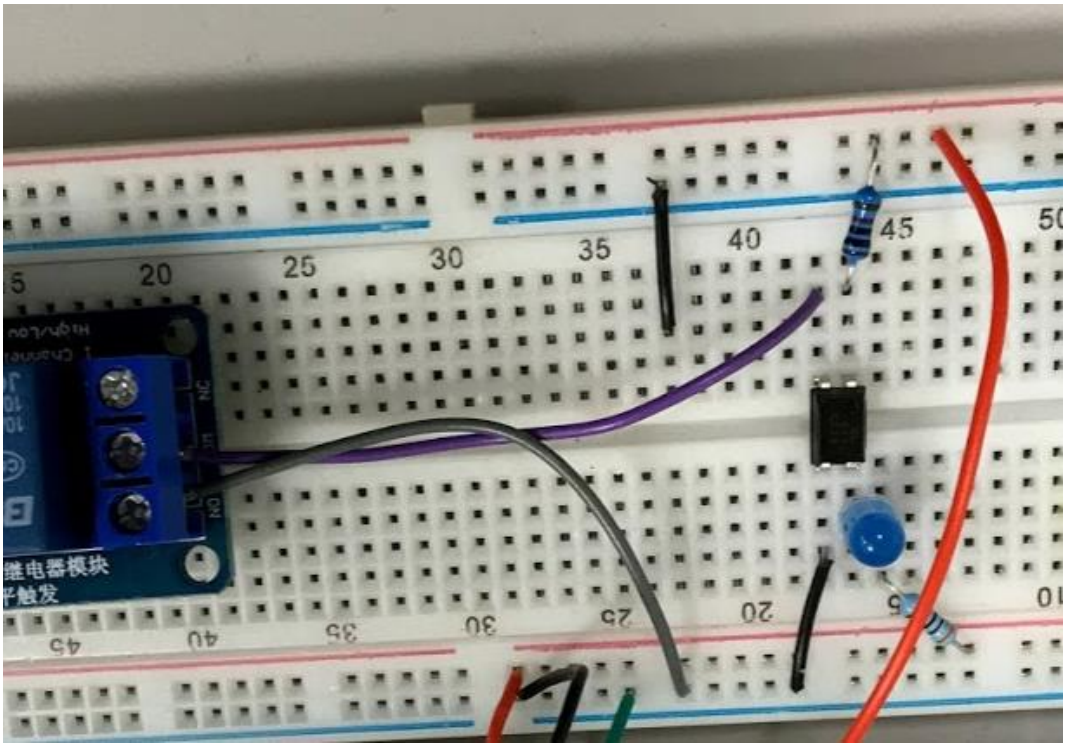

Figure 21: The device input circuit built on a breadboard in the off-state

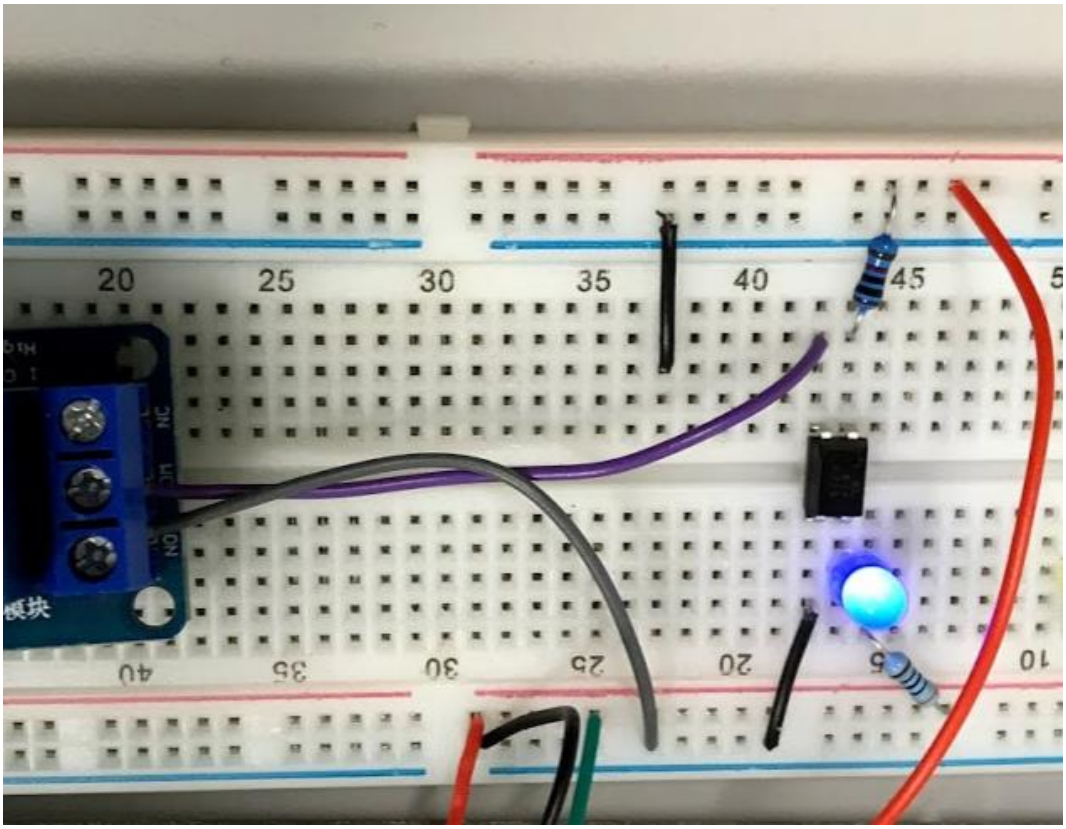

Figure 22: The device input circuit built on a breadboard in the on-state

Solenoid Valve Monitoring

As discussed in the Testing and Future Development sections, the addition of a device monitoring function would significantly enhance the research.

Figure **23** shows a circuit to monitor a solenoid valve. The voltage and current are measured and converted to a suitable range for the ADC on the microcontroller to sample. The circuit works with supply voltages of 24 v DC and 500 mA of current.

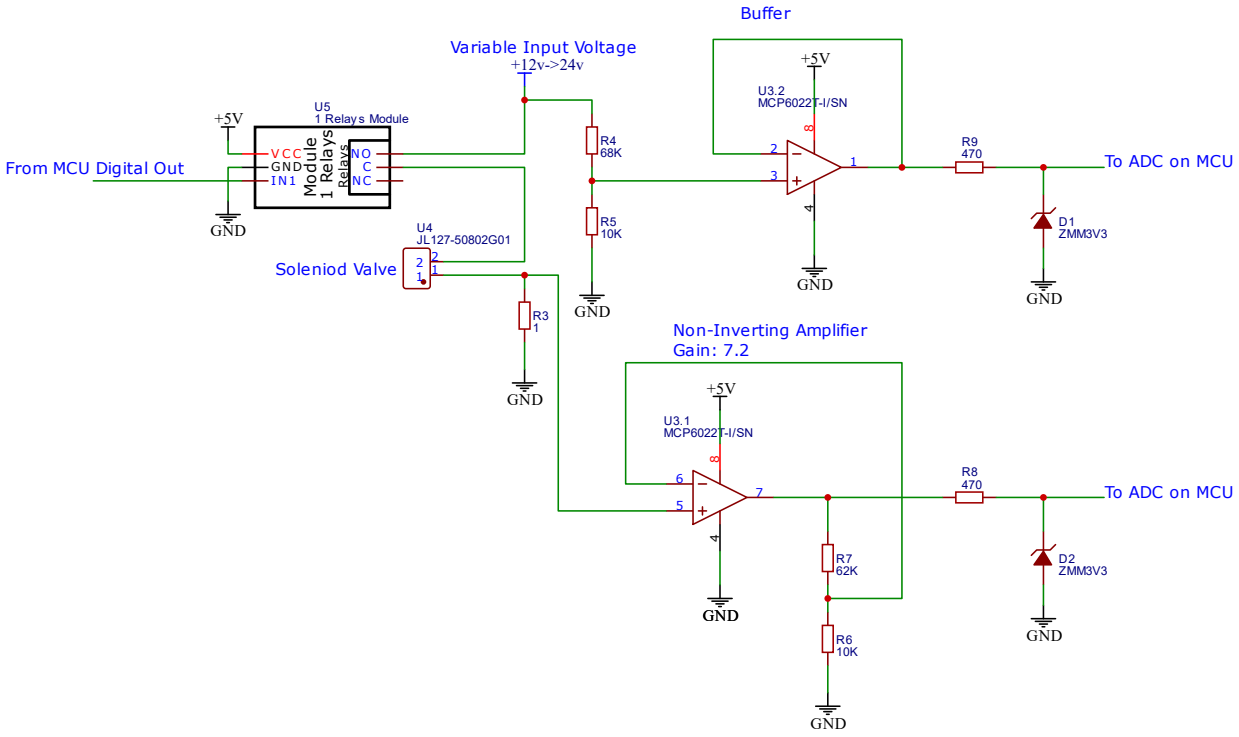

Figure 23: Solenoid valve monitoring circuit

Testing

On Thursday, 12th March 2020, the system had its first test in controlling the lab devices. The purpose of the test was to ensure that the system could control a variety of devices successfully. The test results were encouraging, with a six-port valve, solenoid valve and peristaltic pump all being successfully controlled by the system, as shown in Figure 24.

Further work was also suggested as desirable additions to the research, with a monitoring system to detect device failures. This is an excellent idea to improve safety and save time troubleshooting.

Figure 24: Testing the controller with a six-port valve, solenoid valve and peristaltic pump

The production of a PCB would significantly enhance the reliability of the research by eliminating most of the flexible wire and connectors. It would also allow a smaller case to be used, reducing the cost of the enclosure.

The use of a different enclosure is required if high voltages are to be used. If a PCB reduces the size of the electronics, then a smaller and cheaper case could be used. Also, a case that could withstand high voltages would be beneficial as it would allow new devices to be added in the future, without worrying about the enclosure not being up to the task. Additionally, the case would have extra certifications, such as an IP rating and fire-resistance rating. This would increase confidence in the case to withstand harsh conditions.

Conclusion

This research has clearly shown that lab devices can be automated, saving time while increasing the accuracy and repeatability of measurements. The results from the controller are encouraging, showing that the lab devices can be effectively automated with low-cost components. This is of great value to many laboratories, where finances and time are limited. Of particular interest is that once programmed, there are no ongoing licensing fees or recurring costs, and the system works on any computer with a USB port, making deployment simple.

Flow injection analysis is used in many laboratories in order to analyses a range of chemical species, such as trace metals, nutrients and organic compounds. Each analysis has its own devices and routine, and the ability of the controller to be reconfigured quickly and efficiently is of significant value. The user guide helps new users start using the system quickly and efficiently.

The research has provided an excellent opportunity to explore the capabilities and limitations of modern STM32 ARM processors in real-time applications. Additionally, the development of the web interface has been of significant academic value.

Future Development

There is some scope for future development and improvement. For example, adding additional processing power via a Raspberry Pi or other single board computer would be beneficial for firstly hosting the web interface. Although the microcontroller hosts the interface well, there are some limitations, such as not being able to serve large files, such as device manuals or datasheets which would be useful to store on the controller. Also, the additional processing power available would be useful for running analysis on the data collected, such as plotting graphs or calculating statistics.

In addition, the webserver could then be removed from the microcontroller, instead communicating to the SBC over a serial interface. This would allow the microcontroller to be replaced by a cheaper model without ethernet, such as the STM32F446. It does not have an ethernet connector and has fewer GPIO pins, but it is £9.02 cheaper than the STM32F429[2]. Other improvements include the ability to interface with a photomultiplier tube, as this would allow the progress of the reaction to be monitored and provide useful data for analysis.

Also, as touched on in the Device Status Monitoring section and the Testing section, the implementation of device inputs and monitoring would significantly enhance the research by automating the process of testing devices. The continuous monitoring of devices is excellent from a safety perspective, as the whole system can be shut down if there is a fault with a device. Device inputs would allow the controller to integrate much more effectively with other lab devices, saving time by reducing the required amount of supervision.

Acknowledgements

I would like to thank Dr Antony Birchill for his support and for agreeing to be the client for the project.

The mentor support provided by Dr Paul Davey has been invaluable throughout the project and was greatly appreciated.

Finally, I would also like to thank all members of my family for their continued support.

[2] Cost refers to the development board of each microcontroller at RS Components Ltd. Prices correct as of 25/05/2020

Glossary

ADC – Analogue to Digital Converter

API – Application Programming Interface

ARM – Advanced RISC Machines Ltd

ASCII – American Standard Code for Information Interchange

CSV – Comma Separated Value

GSIOC – Gilson Serial Input Output Channel

GPIO – General Purpose Input Output

HAL – Hardware Abstraction Layer

HTML – HyperText Markup Language

HTTP – HyperText Transfer Protocol

IDE – Integrated Development Environment

Indexed DB – Indexed DataBase

JSON – JavaScript Object Notation

MOSFET – Metal Oxide Semiconductor Field-Effect Transistor

RISC – Reduced Instruction Set Computer

ROM – Read-Only Memory

SBC – Single Board Computer

USART – Universal Synchronous/Asynchronous Receiver/Transmitter

USB OTG – Universal Serial Bus – On The Go

References

[1] Gilson Incorporated, “User’s Guide Peristaltic Pump,” [Online]. Available: https://gb.gilson.com/pub/static/frontend/Gilson/customtheme/en_US/images/docs/MINIPULS3_UG_LT801121-17.pdf. [Accessed 15 October 2019].

[2] Arduino AG, “Arduino Leonardo,” [Online]. Available: https://www.arduino.cc/en/Main/Arduino_BoardLeonardo. [Accessed 10 October 2019].

[3] Adafruit Industries, LLC, “Ethernet Gadget | Turning your Raspberry Pi Zero into a USB Gadget,” [Online]. Available: https://learn.adafruit.com/turning-your-raspberry-pi-zero-into-a-usb-gadget/ethernet-gadget. [Accessed 08 October 2019].

[4] Songle Relay , “Songle Relay SRD,” [Online]. Available: https://www.switchelectronics.co.uk/pdf/SRDsongle.pdf. [Accessed 09 May 2020].

[5] VICI AG International, “Two Position Microelectric Valve Actuator,” [Online]. Available: https://www.vici.com/support/tn/tn421.pdf. [Accessed 12 October 2019].

[6] ARM MBED, “Using the Online Compiler,” [Online]. Available: https://os.mbed.com/docs/mbed-os/v5.15/quick-start/online-with-the-online-compiler.html. [Accessed 23 May 2020].

[7] ARM MBED, “Mbed Studio,” [Online]. Available: https://os.mbed.com/studio/. [Accessed 22 May 2020].

[8] W3 Schools, “JSON Syntax,” [Online]. Available: https://www.w3schools.com/js/js_json_syntax.asp. [Accessed 20 May 2020].

[9] S. Mokrani. [Online]. Available: https://os.mbed.com/users/samux/code/MbedJSONValue/. [Accessed 07 March 2020].

[10] ARM Ltd, “mbed EventQueue,” [Online]. Available: https://os.mbed.com/docs/mbed-os/v5.15/apis/eventqueue.html. [Accessed 22 May 2020].

[11] R. Ramsay, “Horizontal Stacked Bar Chart,” [Online]. Available: https://codepen.io/richardramsay/pen/ZKmQJv. [Accessed 15 April 2020].

[12] W3 Schools, “HTML Tag,” [Online]. Available: https://www.w3schools.com/tags/tag_span.asp. [Accessed 22 May 2020].

[13] M. Asadi, “Sorting a JSON array according one property in JavaScript,” [Online]. Available: https://medium.com/@asadise/sorting-a-json-array-according-one-property-in-javascript-18b1d22cd9e9. [Accessed 01 May 2020].

[14] RS Components Ltd, “STMicroelectronics STM32 Nucleo-144 MCU Development Board NUCLEO-F429ZI,” [Online]. Available: https://uk.rs-online.com/web/p/processor-microcontroller-development-kits/9173775/. [Accessed 15 October 2019].

[15] RS Components Ltd, “Parallax Inc 27115,” [Online]. Available: https://uk.rs-online.com/web/p/power-management-development-kits/8430834. [Accessed 15 May 2020].

[16] RS Components Ltd, “RS PRO, , ABS Project Box, White, 291 x 264 x 111mm,” [Online]. Available: https://uk.rs-online.com/web/p/instrument-cases/2374476/. [Accessed 12 March 2020].

Appendices are provided as a supplementary file in the download area.