CS723 Assignment One Final Report

Dylan Chamberlain, Franklin O’Sullivan, Bailey Clague

Table of Contents

Introduction

Mains power in New Zealand is specified to operate at 50Hz AC. This is subject to change according to the power network's balance of supply and load. A deviation in the 50Hz frequency can cause devastating effects on the components within the network, even leading to a total disaster of the power system. A low-cost frequency relay (LCFR) is a device designed to accurately detect abnormal power frequency on behalf of an individual consumer. It reacts quickly to disconnect or reconnect loads to maintain a stable network frequency. By deploying hundreds of thousands of these devices in homes across New Zealand, their simultaneous action can help keep the overall power system network stable.

Initial Design

A key component in the LCFR design was the use of an embedded operating system such as FreeRTOS [1] (Free Real Time Operating System) to meet the mandatory hard timing requirements. These requirements were implemented to ensure that the LCFR would respond to changes in the frequency or absolute rate of change of frequency before damage could be caused to the power system. The use of FreeRTOS allowed us to schedule tasks that would run periodically, checking the current frequency and rate of change of frequency values against a configurable threshold, managing loads that needed to be disconnected or reconnected and updating outputs such as a set of LEDs (Light Emitting Diodes) and a VGA (Video Graphics Array) display. FreeRTOS further allowed us to receive interrupts from external events and process the data into a queue that could be read at a convenient time. Data integrity was enforced through the use of mutex semaphores. An initial design of the LCFR can be seen in Figure 1.

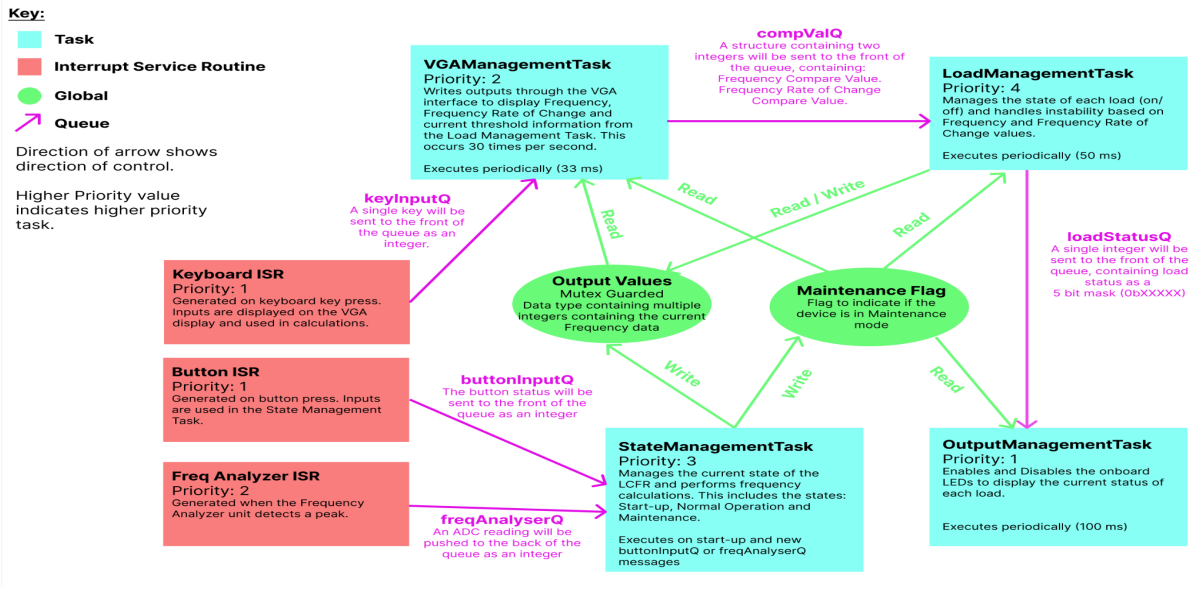


Figure 1: Initial design of the LCFR

Implementation

The final implementation of our system features four tasks, three interrupts and a single queue. We split the functionality of the system into these four tasks because we want to ensure that each major part of the system is able to occur within a reasonable amount of time. Each of the tasks completes one of the key features we require for our design. Communication between the tasks is largely achieved through the use of global variables. These variables are not fit to be used in a queue because we only ever want to use the most recent value. For this reason, the variables are protected with mutex semaphores. We found that the highest priority task should be related to the core functionality of the LCFR, with lower priority tasks being used for more “nice to have” features. A current design representation of the LCFR can be seen in Figure 2. Further instructions on how to run the system can be found in *Appendix A.*

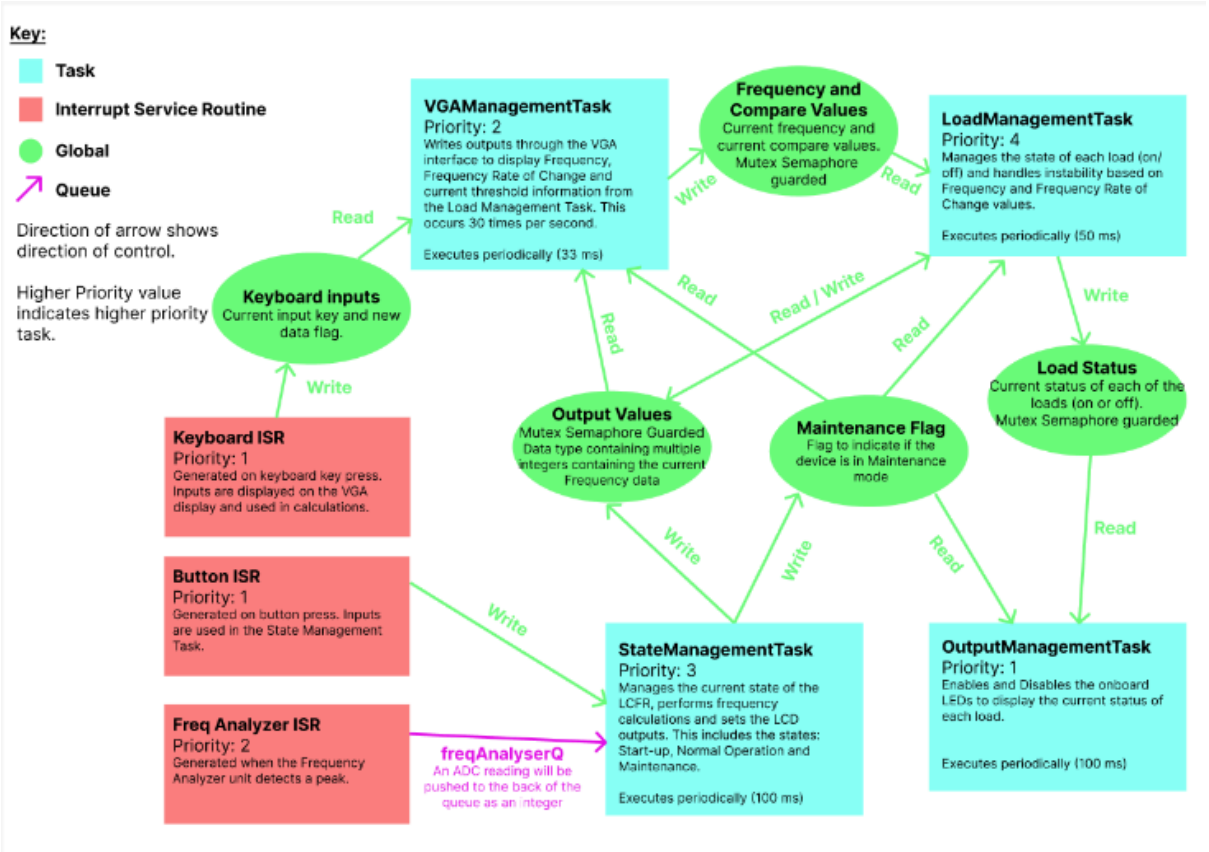


Figure 2: Design representation of the current LCFR implementation

Tasks

Load Management Task - Priority 4

The load management task reacts to undesirable power system behaviour. If the system is unstable, the load management task disconnects loads. Loads disconnect in order of lowest priority. Once the system is stable. The load management then reconnects the loads in reverse order.

The LCFR requires that the first load disconnects from the system within 200 ms of a detected deviation. Because of this, the load management task is the highest priority. We needed to ensure that loads could be shed as quickly as possible. If the deviation remains, a load is shed every 500 ms. Once the system is stable, this task reconnects loads every 500 ms until the operation returns to normal.

The load management task executes periodically every 50 ms. This is frequent enough to handle the loads effectively whilst not hindering the operation of other tasks.

State Management Task - Priority 3

The state management task handles inputs from the frequency analyser unit and buttons. Based on these inputs, the task performs calculations to work out the current frequency and current rate of change of frequency. This data is stored in a circular buffer, using a mutex semaphore, for use in the VGA management task. The state management task makes further comparisons with this data to determine whether the system is currently stable or unstable. A sample diagram of the state transitions can be seen in Figure 3. If the state machine enters the load management state due to instability in the frequency or rate of change of frequency, a flag will be set, notifying the load management task that loads should be managed.

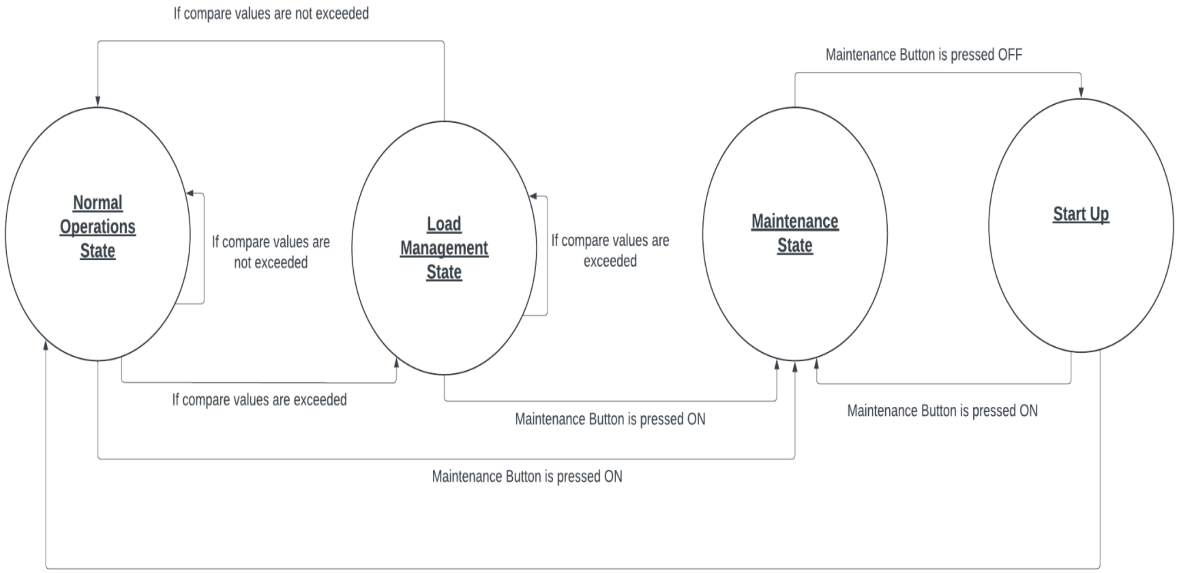


Figure 3: Simplified state transition diagram

If a button press is detected on key 1, the LCFR enters its maintenance state. This state occurs when the LCFR should not manage loads, allowing users to make any changes they wish.

The LCD is also updated and printed within this task based on the current state we are operating in. The screen allows the user to see if we are in “Normal Operation”, “Load Managing”, “Maintenance” or “Start-up”.

This task is periodic, running every 100 ms. This is frequent enough for data to be processed without impeding the other tasks.

VGA Management Task - Priority 2

This task writes to the VGA display. It displays two charts depicting the current frequency and rate of change, of frequency over time. The task retrieves the values for these charts from the circular buffer. It further shows some useful statistics such as the compare values used in the load shedding calculations, the system uptime, current system stability, the five most recent initial load shedding times, the minimum load shedding time, maximum load shedding time and the average of the five most recent times. The VGA task is further responsible for handling input from the connected keyboard. The keyboard inputs are displayed through the VGA, and the inputs configure the compare values used in the load-shedding calculations.

This task is periodic, running every 15 ms under ideal conditions. The reason for this is that the VGA display needs to be refreshed every 33ms at minimum to ensure the screen does not flicker. This is the second lowest priority task. While having a display that clearly shows the correct data is important, it is more important to manage the loads and ensure that system stability is maintained.

Output Management Task - Priority 1

This task is designed to handle all the outputs besides the VGA display. When a load is enabled, the corresponding red LED illuminates. If the LCFR needs to disable this load, the red LED will be disabled, and the corresponding green LED will be illuminated in its place. This task is the lowest priority task because it is more important to ensure that the loads are being disconnected when instability occurs. This task occurs periodically to ensure that the LEDs are updated frequently enough to display to the user visually which loads are currently enabled or disabled.

Interrupt Service Routines

Frequency Analyser Unit

The frequency analyser unit generates new data points at a frequency of approximately 50Hz meaning new data points are available approximately every 20ms. We must capture this data correctly and immediately to ensure our relay is operating with the most recent frequency readings. Therefore, we use the interrupt service routine to send this data using the frequency analyser queue (freqAnalyserQ).

Keyboard Input

The keyboard interrupt is used to take input from the PS-2 keyboard. Each time a key is pressed, the interrupt gets the code for that key and passes it to a variable. For each new key press, a flag is also set to notify the VGA management task that a new input has been recorded. This input is later translated into an ASCII value and used by the VGA management task.

Maintenance Button

The maintenance button interrupt is a hardware interrupt that is caused when the user presses one of the keys. This ISR reads the value of the buttons and passes it to the state management task before clearing the interrupt flag.

Global Variables

As seen in the conceptual design diagram, global variables (in green) helped us to communicate effectively between the tasks. We elected to use a mutex semaphore when writing to these global variables to ensure data integrity. A mutex semaphore ensured that no other tasks could access or modify the data in use until all changes had been completed. This meant that our system could reliably use these global variables to enact state changes and load-managing decisions, as well as display correct and current information on the monitor through the VGA functionality.

Free RTOS Features

Semaphores

The main way we have interacted with the free RTOS feature is through the use of semaphores throughout our tasks to protect our global variables. We had the following semaphores in our project.

Output Values Semaphore

The output value semaphore is declared as “outputValues_semaphore” and is a mutex semaphore. This semaphore is accessed in the state management task where the critical section writes the frequency and rate of change values calculated to the circular buffer. The semaphore is used to ensure mutual exclusion and therefore that values are not being written/pushed to the circular buffer in other tasks at the same time. This helps to prevent deadlock and race conditions. The shared resource of the circular buffer is then accessed again in the VGA task to draw the graph that is displayed on the monitor.

Load Status Semaphore

Our load status mutex semaphore was used in the load management task and our output task. In the load management task when the mutex semaphore was accessed the switchEnabled and loadDisabled variables were written to. These variables determined which of the green and red LEDs were to light up, dependent on the load management decisions made in the state. These variables need to be protected by a critical section and mutual exclusivity as we needed to ensure no race conditions or deadlock occurred as the

LEDs showed the user the status of the loads. The mutex semaphore was then accessed again in the output state where the values in the critical section were updated then, the LEDs were set too to display the status of the loads.

Drop Load Semaphore

The drop load semaphore is a mutex semaphore that sets flags in its critical section. Specifically, the flags set when the semaphore is accessed in the state management task to set the load_drop flag which tells us when a load needs to be dropped (load_drop = 1), no load needs to be dropped or added (load_drop = 0) or when a load needs to be added back (load_drop = -1). The value that flag is set in the critical section is a result of the comparisons of the current frequency and rate of change to the given compare values. We also look at the timing constraints and if a load needs to be added back or dropped due to stability or instability. This flag is then accessed again in the load management task where we identify which load is to be dropped or added and then communicate that with the variables controlling the LEDs.
The first load dropped flag is accessed in this semaphore and helps us determine the time it takes for that first load to be shed to ensure we meet the timing requirements. In the load management task when the first load is dropped we write to this flag by accessing the semaphore. Then in the state management task when the relay first goes into normal operation we write the flag to zero by accessing the semaphore.

Queues

To properly store frequency values to then retrieve and calculate the rate of change and compare with the compare values we implement the free RTOS queue. Within our code, the queue was called FreqAnalyserQ and upon the frequency analysis unit interrupt being triggered, the interrupt would send the frequency value to the back of the queue using the xQueueSendToBackFromISR() function. pdFalse was an argument to this function meaning no context switch occurred. We did not need a context switch as this interrupt did not need to trigger a task, as the task where the frequency data was retrieved from the queue occurred periodically. Although the risk of using a “queue send” in an interrupt is present, we want to keep the interrupt as calculation and action light as possible. The frequency value that was sent to the queue is relatively small and is a light operation for the interrupt to perform without worry of the interrupt taking too much time.

Task Utilities

Another free RTOS feature we made use of was the xTaskGetTickCount() function to get a start and end time value to calculate the total time that was taken to not only respond to the frequency event and ensure we responded within the 200ms time but also to ensure that after 500ms we could calculate to either shed the next load or add the load back. When the first load is identified to be shed we get the tick count then when the load is shed we stop the tick count. Likewise, with identifying if we need to restore or shed due to the 500ms timing requirement, we get the tick count every time a load is shed then use the current time in an if statement to ensure when we hit the 500ms we act appropriately.

Resolved Issues & Implementation Problems

When we were initially designing the LCFR we thought it would make sense to use queues to communicate between tasks. Once work began on development, we found that this was not an effective way to pass data, and instead of helping to simplify our system, it almost seemed to hinder us. We found that the queues were an effective method of transferring data when multiple data points were used in calculations. In our case, all of the data we were transferring was instead the most recent values of a variable and it was intended that only these most recent values were to be used in the calculations. Because of this, we elected to swap the majority of our queues with global variables.

Another issue we encountered during the implementation of our design was in the VGA task. We thought initially that we would need to draw to the display every 33 ms in order to display a clear image. When we got to programming, we instead found out that we only needed to write new values to the display, with all old values persisting. This simplified the use of the VGA a lot and allowed it to take up less of the system time.

A further issue that we addressed in the development of our project was related to multitasking. There were some cases in which a load was dropped within 200 ms but we received a reading that it had taken 57 seconds. We found that this error was caused by an integer overflow. We assumed that the timer would be started in the state management task when instability was detected and stopped in the load management task when the first load was dropped. We instead found that the timer was starting multiple times, often leading to an end time smaller than the start time and therefore a negative time taken to drop the load. This issue was resolved by ensuring that the timer could only be started once.

Limitations of the current design

Our current design has all of the tasks running periodically. Although this provides us with certainty and predictability of code, it also does provide a few drawbacks. Resource usage and the lack of event driven flexibility means that even though on a small scale, like what we are currently faced with, the system works. On a larger scale with more loads present or a greater quantity of data, this solution could be susceptible to things such as context switching. A task may not be able to complete its operation before being interrupted, possibly never being able to complete its execution. This would have a significant impact on the operation of the relay and its capabilities.

Future Improvements

A possible future improvement for this project would be to allow the LCFR to communicate through the Internet. This would allow for mobile management of the LCFR and the loads it controls. Internet connectivity could be achieved using an external unit similar to the Frequency Analyser Unit. Internet communications could be handled externally, with only useful data and flags reaching the LCFR.

To communicate with the external internet connection component, we could expect it to generate two types of interrupts. One interrupt will request new data. When this interrupt occurs, it will be able to read the current values of global variables such as the current frequency, current rate of change of frequency, both of the current compare values, as well as, all of the timing data currently being displayed on the VGA display. This would allow the mobile or web app to display all of the data from the VGA display in a more user-friendly manner.

The second type of interrupt that the internet connection component would be able to generate is the post interrupt. In this interrupt, the component will be able to notify the relay of any changes that have been requested on the app. One of these could be to change the compare values for the frequency or rate of change of frequency. Another option would be to change the priority of loads, allowing loads to be disconnected and reconnected in a different order.

A prototype implementation of these interrupts can be seen in Figure 4, although a full implementation would require many complex changes and the creation of more global variables or queues.

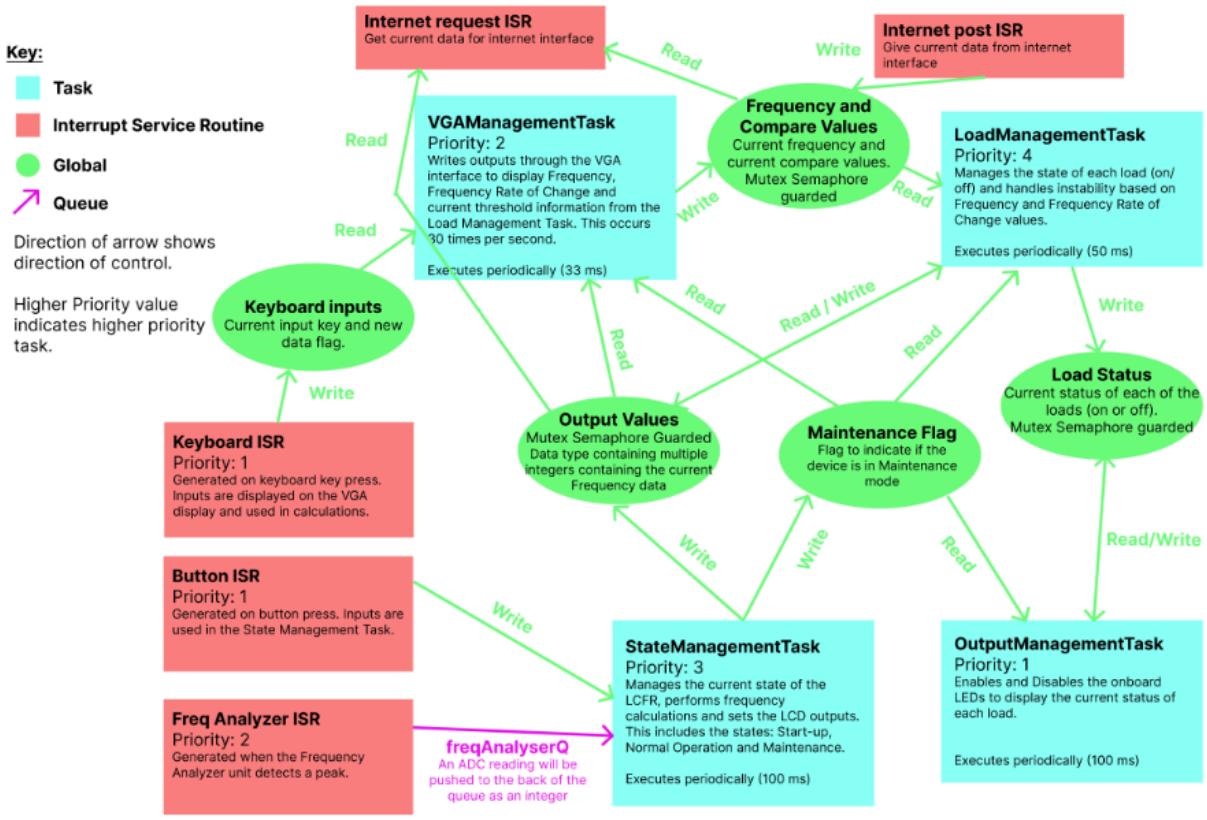


Figure 4: Prototype connections for Internet interrupts

The data that is sent and collected through these interrupts will be displayed on a mobile or web app. At this point, it can be expected that the apps will look very similar to the VGA display, as can be seen in Figure 5.

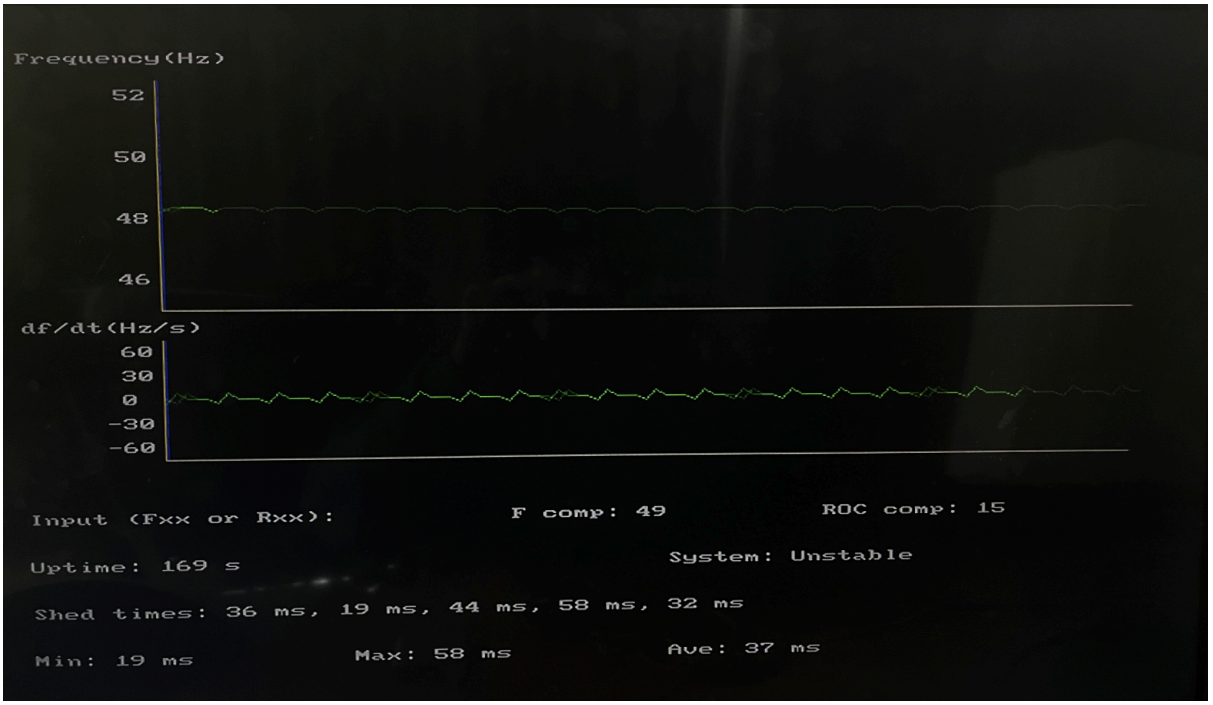


Figure 5: Current VGA output

The mobile app will be a modified version of this display. This is due to the difference in aspect ratios. Both applications will further be able to display data about the system as a whole. This is because the apps will have access to data from LCFRs located in many homes across New Zealand. The apps will be able to display charts showing where in the country usage is the highest and in which places instability exists. This will allow for a better understanding of the country's power networks while allowing companies to see space for improvements or missed inefficiencies.

The keyboard interface currently being used for the LCFR is not user-friendly and can cause issues if used incorrectly. One major improvement for this project would be to implement a better interface that can cater to the user's needs without being complex or causing problems. One way this could be implemented is through the use of a mobile interface. Data could be written to the LCFR through Bluetooth or NFC (Near Field Communication). A mobile interface will be easy for all people to use and simplifies maintenance of the device.

We could have also improved our design by creating a timing task that set up and ran timed responses for each task and ensured that ran properly and only when needed. For example, the VGA could have been on this timer so the refresh rate was fixed.

Project Timeline

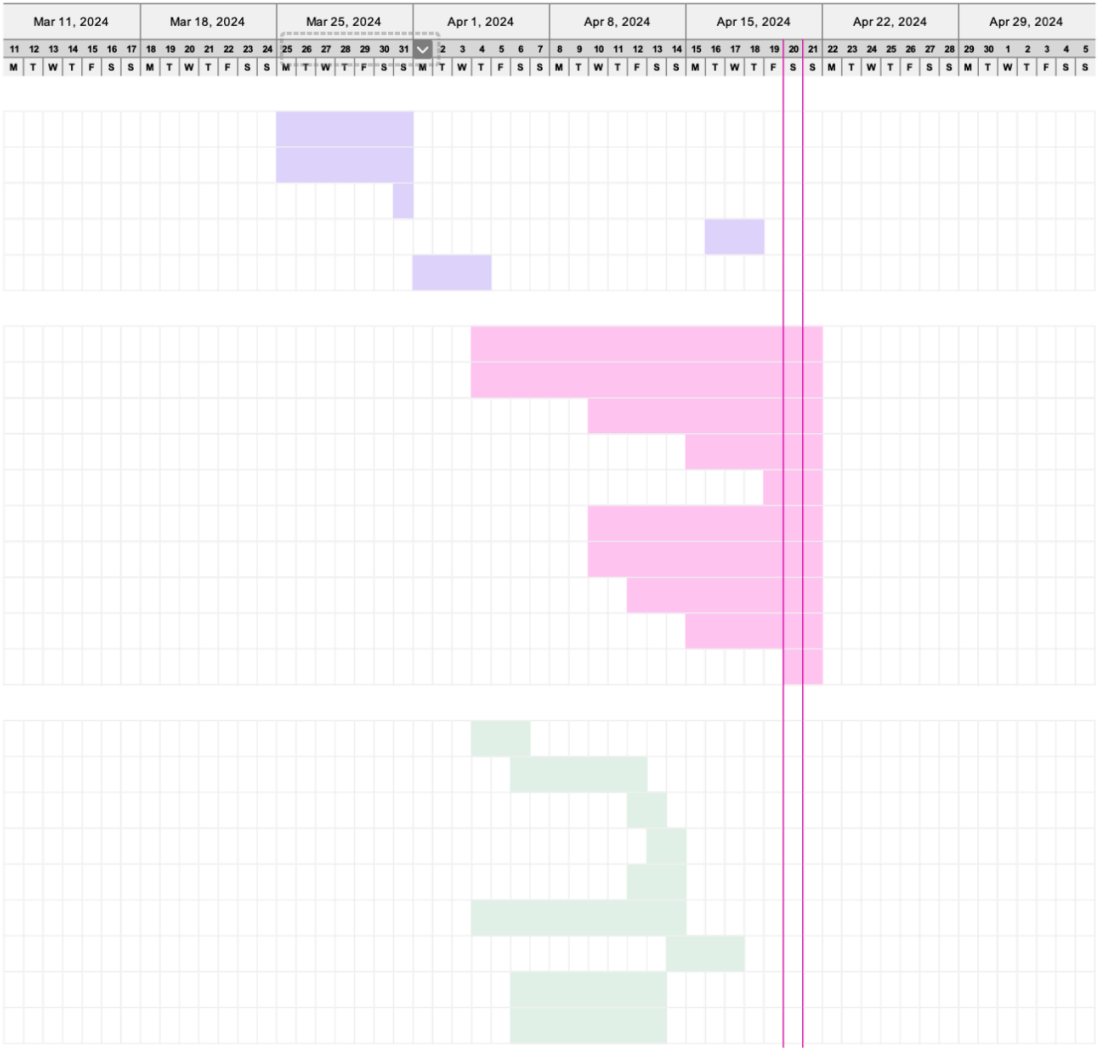
723 Assignment 1

Team #2

Project start: 15/03/2024

Display week: 1

TASK	ASSIGNED TO	PROGRESS	START	END
Planning and Documentation				
Initial Design Meeting	Everyone	100%	25/03/2024	31/03/2024
Figma Diagram Design	Everyone	100%	25/03/2024	31/03/2024
Submitt Initial Design	Everyone	100%	31/03/2024	31/03/2024
Slides for Demo	Bailey & Franklin	100%	16/04/2024	18/04/2024
Gantt Chart	Bailey & Franklin	100%	01/04/2024	4/04/2024
Report				
Initial Template	Franklin	100%	04/04/2024	21/04/2024
Task Definition	Bailey & Franklin	100%	04/04/2024	21/04/2024
Task Reasoning	Bailey	100%	10/04/2024	21/04/2024
FSM and Diagrams	Bailey	100%	15/04/2024	21/04/2024
Final Figma Design	Dylan & Franklin	100%	19/04/2024	21/04/2024
Design Plan Summary	Dylan	100%	10/04/2024	21/04/2024
ISR's	Bailey & Dylan	100%	10/04/2024	21/04/2024
RTOS Features	Bailey	100%	12/04/2024	21/04/2024
Report Conclussion	Dylan	100%	15/04/2024	21/04/2024
Proof Read and Submitt	Franklin	100%	20/04/2024	21/04/2024
Code				
Implement Switches and ISR's	Franklin	100%	04/04/2024	6/04/2024
Implement VGA management task	Franklin	100%	06/04/2024	12/04/2024
Semaphore integration	Bailey	100%	12/04/2024	13/04/2024
Implement Timing conditions	Dylan	100%	13/04/2024	14/04/2024
Load Management Task	Bailey	100%	12/04/2024	14/04/2024
Output Task	Dylan	100%	04/04/2024	14/04/2024
Timing calculations	Franklin	100%	14/04/2024	17/04/2024
State Management Task	Bailey & Dylan	100%	06/04/2024	13/04/2024
Refactor and Tidy up	Everyone	100%	06/04/2024	13/04/2024



The above Gantt chart [2] shows the breakdown and allocation of work throughout the project. This plan was made and modified based on discussions in team meetings. Work was assigned to group members with an expected completion time. On average, we would work about 2 hours per day. Most deadlines were met as planned, with only a few having to be modified or changed.

Conclusion

FreeRTOS allowed us to design a reliable, low-cost solution to manage power system stability in New Zealand. Our solution takes advantage of FreeRTOS features to process frequency and rate of change of frequency values to control the disconnection and reconnection of multiple loads. Whilst there is space for many future changes or developments, the current state of the project, the design was able to satisfy all of the timing requirements while offering additional features for usability. By deploying hundreds of thousands of these devices in homes across New Zealand, their simultaneous action can help keep the overall power system network stable.

References

1. FreeRTOS. “FreeRTOS - Market Leading RTOS (Real Time Operating System) for Embedded Systems with Internet of Things Extensions.” Accessed April 20, 2024. https://www.freertos.org/index.html.
2. Atlassian. “Gantt Charts.” Atlassian. Accessed April 21, 2024. https://www.atlassian.com/agile/project-management/gantt-chart.

Appendix A - Instructions for running code

Running the LCFR on a DE2-SoC board is very simple, given all the correct files and software are present.

Required software:
Quartus Prime (Standard or Lite) 18.1
Eclipse development tools for Quartus Prime

Required hardware:
DE2-SoC Board
DE2-SoC Power cable and adapter
USB Blaster cable
VGA Cable
VGA to HDMI
HDMI Cable
PS-2 Keyboard
Monitor with VGA connectivity

1. Connect the DE2-SoC to your PC (Personal Computer) using the USB-Blaster cable. Make sure that the board is powered and turned on. When the power button SW18 is pressed, the LED D1 should illuminate.
2. Download the given assignment zip file and move the contents into an appropriate folder.
3. Open the Quartus Prime Programmer and select the “freq_relay_controller.sof” file, found in the “A1” folder. Ensure the programming switch SW19 is set to RUN, then press “Start”.
    a. If programming is successful, the programmer tool will show a green bar with “Success” in the top right corner of the window.
4. Open Eclipse Development tools for Quartus Prime under tools.
5. Once Eclipse is opened, select your appropriate workspace directory, then click “ok”
6. Select “File”, “Import”
7. Once the popup has opened, select General > Existing Projects into Workspace > Next.
8. Then navigate to “A1” folder and click “Finish.”
9. Right-click on the project BSP navigate down to Nios II and click “Generate BSP”
10. Once this is complete clean and build the project by navigating to Project in the menu bar and then clicking clean, then build.
11. Once this is complete you are ready to run the software on the Nios II hardware. Simply right click on the project in the left hand column and find Run As then Nios II hardware.
12. Once this is complete your board and vga should light up.
13. At any given time to enable a load use switches SW0 to SW6
14. At any given time if you would like to enter maintenance mode click push button KEY1.
15. The LCD and VGA should be responsive to the current state and operating mode.
16. When you would like to change the frequency compare value use the keyboard and type FXX where XX is the two digit frequency compare value. eg. F07 would give a compare value of 7Hz. Then hit enter.
17. When you would like to change the rate of change compare value simply type RXX where XX is a two digit number, eg. R12 would give a rate of change compare value of 12 Hz/s. Then hit enter to confirm the value.