Final Design Report

COMPSYS 723 Assignment 1 - Team 12

Hao Lin
UPI: hlin784

Chamith Nanayakkara
UPI: cnan194

Instructions for running this project on a DE2-115 board

Folder Structure:

documentation:

- Brief and system spec document
- A copy of this report

system

- System SRAM object (.sof)
- SOPC Builder file (.sopcinfo)

software

- customAPI - contains all functions for interacting with board & peripheral components
- freeRTOS - freeRTOS system files
- RTOS - contains all custom define tasks and global variable
- Main.c

Program a de2-115 board with the .sof file and build an empty eclipse project with the .sopcinfo. Move all folders under software into the software folder of the project. Set the 1u timer as the timestamp timer in the BSP editor. Compile and run with a VGA display and p/s2 keyboard connection. If the numerical displays of frequency and ROC are unreadable, the rate at which the "snapshots" are updated may be changed in 'service_VGA.h'.

P/s2 Keyboard Input commands

All inputs are inputted and then registered via ENTER key. Ensure ENTER is pressed to clear previous input before inputting a new command. After entering one of the threshold setting commands (and have registered it) the next input will expect a decimal number and set the threshold value to that input. Any invalid input will not change the threshold value.

Available Inputs (case-sensitive)

- ENTER (key) - register command and clear carriage for new input.
- freq - began setting frequency threshold. Next set of inputs determines the value.
- rate - began setting the rate of change threshold. Next set of inputs determines the value.
  Example:
  freq + ENTER + 49.6 + ENTER - will set the frequency threshold value to 49.6Hz

Solution Description

Calculating Frequency and ROC task ‘calc_fre_roc’

Priority 11 (priority compared to other tasks 4) - Non periodic

This task is used to calculate the frequency value and the frequency rate of change value (ROC). This is a non-periodic task and has the highest priority since it is critical for the task to be completed on time for all the subsequent systems to operate.

The task reads the `queue q_ADC_sample_values` to retrieve the raw ADC sample of the incoming frequency. Once the semaphore `freq_ADC_received_semaphore` is called, the old frequency value is stored using the previous new frequency value and same with the old frequency ADC samples. The new frequency ADC samples are retrieved from the front of the queue. Afterwards, the new frequency value is calculated by:

$$new\_freq\_value = \frac{SAMPLING\_FREQ}{new\_freq\_ADC\_samples}$$

Where $SAMPLING\_FREQ = 16000$. And this new frequency value is updated to a queue `q_freq_values` to be read by the `service_VGA` task. The frequency ROC is calculated as shown below,

$$freq\_ROC\_value = \frac{(new\_freq\_value - old\_freq\_value) \cdot 2 \cdot new\_freq\_value \cdot old\_freq\_value}{new\_freq\_value + old\_freq\_value}$$

And also this new `freq_ROC_value` is updated to the queue `q_freq_ROC_values` to be read by the `service_VGA` task. Finally, within this task the `threshold_monitor` function is called to compare with the threshold values and further executions.

Handle Load task ‘handle_load’

Priority 10 (priority compared to other tasks 3) - Periodic, 30ms

The `handle_load` task performs the core functionality for the system. This is time critical as the response timing should be guaranteed as the load shedding is critical to the *correctness* and timeliness of the LCFR. The load status in the external environment directly influences sampling and subsequent calculation.

The task starts by retrieving the load’s actual status, load’s status as demanded by switches and load’s status as logged by automatic shedding information. And once the system status is known the task processes to the free load control. This manages calling to the LED function once the current switch status is known. Afterwards, the automatic load management section is processed. This section of the code controls the automatic load shedding and automatic load connecting if the frequency or ROC is above or below the threshold values and depending on the system status. More in depth, `shed_load` function is called if unstable and `connect_load` is called if the system is stable.

The verification timer is called after every time entered to the automatic load management section, and once the 500ms is elapsed the flag `verification_elapsed` flag is set and read back by the free load controller for its executions. The response timing function is called every time the shed load function is called and ends once come back to the automatic load management section. This response timing function calculates the time it takes for the load to shed after a new frequency or ROC is compared with the thresholds.

Handle Keyboard task *'handle_keyboard'*

Priority 9 (priority compared to other tasks 2) - Non periodic

This is a non periodic task as it is executed upon the keypress on the PS/2 keyboard. It is critical to know the current frequency threshold and the frequency ROC threshold values to compare as immediately the user enters these values. It is important to have ready access to an updated threshold. Otherwise, any subsequent calculations are meaningless if operated on outdated value.

Once keys are pressed on the keyboard, the PS/2 ISR is activated and the data is written into the `PS/2 Buffer`, and also a counting semaphore, `semaphore_keyboard` is given. The system knows if the user is about to type a new frequency or ROC threshold value when the user first types the commands “FREQ” or “RATE”. The data is also written into a queue `string_queue`. The `semaphore_keyboard` semaphore is taken from the `handle_keyboard` task and then `string_queue` is read and discarded afterwards. And finally these values are directly updated to `threshold_freq` and ‘threshold_ROC`.

Service VGA task

Priority 8 (priority compared to other tasks 1) - Periodic, 10ms

This task chooses the lowest priority as this is the task which displays the information to a human-readable format in a VGA display. The main use of this task is to read and send data to the VGA display in a periodic form.

The task reads both frequency and ROC values and plots them in separate graphs in green colour. Above the graphs the VGA displays current frequency and ROC threshold values, 5 most recent frequency and ROC values, minimum/maximum/average reaction times, and system uptime.

Final Design Diagram

System Boot

Tasks
- calc_freq & calc_ROC
- handle_load
- handle_keyboard
- service_VGA

Memory Buffers Queues
- Q_freq_calc_values
- Q_roc_calc_values
- Q_ADC_sample_values
- Q_response_time
- ps2_keyboard_inbox
- response_timing

Semaphores
Counting
- ps2_keyboard_sem
- freq_ADC_received_semaphore

Mutex
- roc_queue_sem
- freq_queue_sem
- threshold_freq_sem
- threshold_roc_sem
- response_time_sem
- global_system_status_sem

Binary
- response_timer_end_gate_sem
- response_timer_start_gate_sem

Intialisation Process in main()
timer
- verification_timer_init();
- response_timer_init();

board state & pherpherials:
- led_init();
- buttons_init();
- ps2_keyboard_init();
- init_freq_analyser();

RTOS (task + inter-task resources)
- init_OS_data_structs();
- init_create_tasks();
- vTaskStartScheduler();

Pheripherials
- LED
- VGA (intialised in task)

ISR
- Pushbuttons
- PS/2 Interface
- Freq-relay

Timers
- Software Verification Timer (500ms)
- Hardware Response Timer (200ms)
- Software Boot Timer

KEY:
Function (or a group of functions)
Tasks
Peripherals
Memory Data
(Border) Shared Resources
(Border) ISRs
Semaphores/Flag

Direction of arrow shows diraction of control.
Priority diagram provided elsewhere.

Peripheral Inputs

Digital Hardware: Simulated Powernetwork
Peak IRQ
ADC count
Reads
IRQ

Measurement
frequency_relay_ISR()
- Read FREQUENCY_ANALYSER_BASE values

Gives
freq_ADC_received_semaphore (Counting)
Writes
q_ ADC_sample_values (Counting Semaphore Guarded)

semaphore_response_timer_start_gate (binary)
Gives
semaphore_response_timer_end_gate (binary)
Takes
reponse_timing Functions
- response_timer_start

Calls
See *handle_load* for next step in response timing control/data flow

Button Input
IRQ
Display/Human interface
Buttons ISR
- Toggles maintainence

Toggles Maintainence (storing current state)

Threshold Input
data stream
Display/Human interface
IRQ
Writes to back
string_queue
Read & discarded by
PS/2 Buffer
Scans
PS/2 ISR
semaphore_keyboard (Counting)
handle_keyboard
- Event (semaphore_keyboard)

Slide Switch Input
Display/Human interface
Note: No IRQ for switches.
check_switch
Calls to read value

Takes

Starts

calc_freq & calc_ROC
SAMPLING_FREQ = 16000
Began
old_freq_value = **new_freq_value**
new_freq_ADC_samples = retrieve the front value from q_ ADC_sample_values
new_freq_value = SAMPLING_FREQ/new_freq_ADC_samples
freq_ROC_value = (new_freq_value - old_freq_value) * 2.0 * new_freq_value * old_freq_value / (new_freq_value + old_freq_value)
threshold_monitor(new_freq_value, freq_ROC_value)
Writes to back
Writes to back
Calls
Q_roc_calc_values queue (Mutex Guarded)
Q_freq_calc_values queue (Mutex Guarded)
Empties
Empties

Global Variables
Each:
- Mutex Guarded
- Wrapped in getters /setters

global_sys_status (Mutex)
threshold_freq_sem threshold_roc_sem (Mutex)

Untable Loop

global_sys_status
Reads & Updates
instant freq >= instant threshold || ROC >= ROC threshold
Reads
threshold_freq threshold_ROC
Read/Compare
threshold_monitor Function

Stable Loop

Updates
verification_elapsed
Reads
500ms Elapsed (IRQ from verification timer)
Reads
Starts
handle_load
- Shred next priority load (set in macro)
- Event (on is_verification_elapsed and sys_statuc)

verification_timer Callback Functions
Free load control - limited load determination
shed_load Function
Independently autonomous (self-contained logic to unstable and normal load shedding)
semaphore_response_time_end_gate (binary)
Takes
semaphore_response_time_start_gate (binary)
Gives
If Unstable: Calls
reponse_timing Functions
- response_timer_end
- calc_response_time

Calls
See *service_VGA* for next step in response timing control/data flow
Automatic load management
If Stable: Calls
connect_load Function
Independently autonomous (self-contained logic to unstable and normal load connecting)
Call: Began timing

Calls
Calls
Calls
I*with appropriate logic

Peripheral Outputs

freq_queue_sem (Mutex)
roc_queue_sem (Mutex)

Display/Human interface
service_VGA
Starts
Data Reading
- All data dumped into local buffer for processing

service_VGA
- Sends data
- Draws Plot
- Periodic (10ms)

Outputs
VGA Display
Get max, min and avg
Reads Uptime
calc_response_values Function
- calculate maximum/minimum, average

boot_timing Functions
- xTaskGetTickCount()
- Returns formatted time

Empties
response_timing_data Queue (Mutex Guarded by response_time_sem)
response_time_sem (Mutex)
When measurement_concludes: Writes data
response_timing Functions
- calc_response_time

Display/Human interface
LED Functions
- read/write

LED (RED)
- 5

LED (GREEN)
- 5

Outputs
Outputs

Changes from Conceptual Design

Separated load reconnection from `shed_load` function as independent `connect_load` function
This choice gives us more flexibility in implementing our design and clearly separates the responsibilities of load shedding and reconnection.

`handle_switches` removed, `is_threshold_exceeded` removed
We found out there is absolutely no need for the `is_threshold_exceeded` flag. Load managing logic solely controlled by system status and verification timer's elapsed indication. The function of `handle_switches` (free load management via the switches) is reallocated to `handle_load` for better sole function delegation. Therefore, rendering `handle_switches` redundant.

`is_maintainence` flag expanded to three-state `global_sys_status` variable
A binary `is_maintainence` was unable to capture the system status completely. Since we have opted for controlling automatic load management via system status instead (removing `is_threshold_exceeded`), a robust indication of system status was necessary.

Other Changes:

- Response timing data flow updated with more details
- Verification timer usage updated to be a flag
- `service_VGA` updated with more details
- 'Calc_freq_roc` updated with more details
- Relocated usage of `boot_timing` function
- Added all semaphores as a diagram block next to mutex guarded resources

Design Decision Justification

Splitting of Functionality

The `handle_keyboard` task was made particularly to interpret the received keys and subsequently set the threshold values. This functionality was separated from the functionality to read the incoming key in the `ps2_isr` to reduce ISR execution time (where the reading functionality was unable to be executed outside the ISR). The task constructs word commands from received keys and maintains a small FSM logic to receive and interpret commands. It was more appropriate to instantiate this function as its own task to retain an exclusive memory over the human-computer input.

Similarly, it was more appropriate to dedicate a `service_VGA` task to service all VGA functionality. The task is a consumer of frequency & roc data, response timing data and reads `global_sys_status`; all to be appropriately stored, formatted and outputted through the VGA pixel and char buffer. In which case even the initialization process for the VGA was performed inside the task. This allows the task to be encapsulated. Like the `handle_keyboard`, this task is also required to retain memory over its buffered output values.

`handle_load` works with the `threshold_monitor` located in the `calc_freq_roc` task to execute the automatic load handling logic. The monitor was placed in the `calc_freq_roc` task to check the newly calculated values against the threshold as soon as possible. The results are communicated to the `handle_load` through the `global_sys_status` where the task will exclusively be responsible for shedding and reconnecting the load in accordance with the system state. However, to allocate all responsibility over the load, the task also handles manual load control. This made it easier to disable certain manual load control functions when necessary instead of communicating the change over another task.

The response timing functionalities are split as an independent set of functions. This allows the functions to be reused in other designs, but the primary reason was to be able to easily perform timing from any location, be it a task or ISR.

Moreso, many functionality was wrapped as helper functions to improve code readability and allow development of separated functionalities across our developers.

Automatic Load Management Design (Sentimentality) Assumptions

Reconnecting Orphaned Green LED Indication
While in load managing state, if a load was disconnected by the relay but was later also disconnected manually BEFORE the relay reconnected the load automatically. The green LED indication would still be on (indicating it had been automatically shedded) while the load became manually shedded as well. Hence, when reconnecting automatically (as the system is in stability and is reconnecting loads), we shall turn off these forgone green LED indications on the next automatic reconnection (and when the system returns to normal monitoring operation). Thus, the green LED will appear as if multiple loads are reconnected within the same cycle of reconnection.

Reconnection of Newly Available-to-reconnect Loads
Another assumption is that while the switches cannot turn the load on manually while in load managing state, if the switch was OFF when load is shedded (hence the off load is skipped) and is then in the ON position while the load management is reconnecting loads, the load will be available to be reconnected by the automatic reconnection logic DESPITE the fact the load was never disconnected by the automatic shedding logic.

Maintenance
An assumption over the maintenance state was that when returning from the maintenance state, the system will enter the state it was in before entering the maintenance state. This assumption was made in awareness of the maintenance state's purpose. We believe the maintenance state will most likely be entered to test out adjustments made by a technician, but also during a prolonged unstable state where a technician might wish to resolve through the maintenance state and then perform checks. Therefore, we made this assumption where the maintenance state effectively pauses and resumes the normal load managing logic (while leaving all other subsystems to operate as normal).

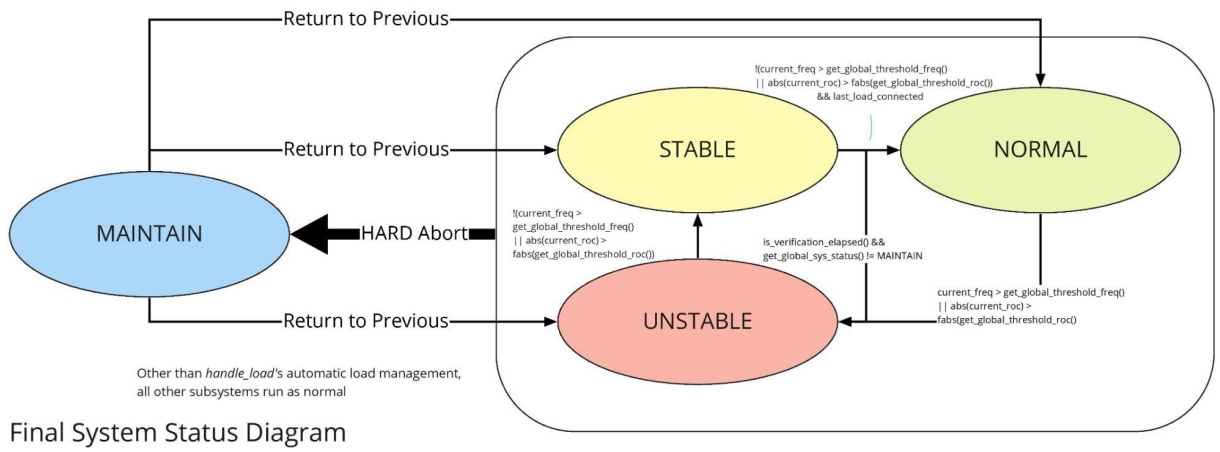

Fig. 2. System status state machine

Stable state is the default operation. Unstable state activates the automatic load shedding behaviour, but is otherwise impactful to other tasks of the system. Meanwhile, maintenance mode effectively blocks automatic load shedding, but has an ambiguous impact on the rest of the system's behaviour.

We have decided to treat the stable and unstable modes as a singular program flow (for automatic load management). With the unstable mode activating the `handle_load` task to execute the automatic load shedding and the stable state reconnecting load until it is safe to return to normal state of operation.

Use of Mutex Semaphore

We have chosen to use a mutex semaphore to protect shared variables. Disabling interrupts and schedulers is unnecessary and will disrupt the operation of the RTOS. A mutex semaphore must be taken and given within the same task. The mutex semaphore was chosen over a binary semaphore as it inherits the priority of the called task, thus preventing the occurrence of priority inversion in our program's operation.

Only the `global_sys_status` semaphore was encapsulated within a getter/setter function, as `global_sys_status` is frequently used as a control signal in the tasks `calc_freq_roc` and `handle_load`, while all other mutexes are explicitly taken and given in the task.

The only case where a binary semaphore was used, is to protect the resources (response_timing_data) of the response timing function flow. Two binary semaphores (semaphore_response_timer_start_gate,semaphore_response_timer_end_gate) are needed to communicate with the ISR to ensure that start and stopping of measurements always operates in pairs before repeating.

Communication mechanism

The main communication mechanisms used in this project were global variables, queues and semaphores.
Global variables do not stay limited to any specific function or a task, which means any given function or a task can read global variables and also modify them. For instance, `global_sys_status` variable is declared as a global variable as the system status (normal, stable, unstable and maintained) is read/modified by multiple tasks/functions.

Queues were used in a situation where the data do not have to be processed immediately, however, have to be processed in a First In First Out (FIFO) order. For example `Q_roc_calc_values` and `Q_freq_calc_values` are FIFO buffers to store frequency and ROC values to be read by the `service_VGA task`. FreeRTOS queues provide a thread-safe buffer and are very simple to communicate between tasks. FreeRTOS also provides queues that are able to communicate between ISRs and tasks safely. These queues can queue data from incoming ISRs data and store them in a FIFO manner and later use them from tasks. As an example, `string_queue` queue is used to store PS/2 ISR data from the keyboard and later read them from the `handle_keyboard` task.

FreeRTOS semaphores are another communication mechanism used in this project. Semaphores are used to control access to shared resources between tasks in a concurrent environment. And these process synchronisation. For instance:

> `freq_ADC_recived_semaphore` is a counting semaphore that is given by the `frequency_relay_ISR()` and takes by the task ‘calc_freq & cal_ROC` to start.

Task priorities

Priorities for each task are assigned based on the importance to the system functionality and to meet the given requirements. `calc_freq_ROC` task is given the highest priority with 11 as this is a prerequisite task to be completed on time for all the subsequent systems to operate.

This is also given the highest priority met to a shorter response time. The next lowest priority 10, is given to the handle load task. This is the next most important task as this is time critical for the response timing to be guaranteed for the load shedding. And also as this task also manages the status of the system.

The next lowest priority with 9 is given to the handle keyboard task. Second priority is given to this task as it is critical to know the current frequency and frequency ROC threshold values to compare as immediately the user enters these values. It is important to have ready access to the updated threshold. Otherwise, any subsequent calculations are void if operated on outdated values.

The lowest or priority 8 is given to the `service_VGA task`. This is chosen as it is not time critical to show information on the VGA display. The values displayed are historical nonetheless. Therefore as long as this service is executed often enough to display all recorded frequency (without backlog), the priority is sufficient.

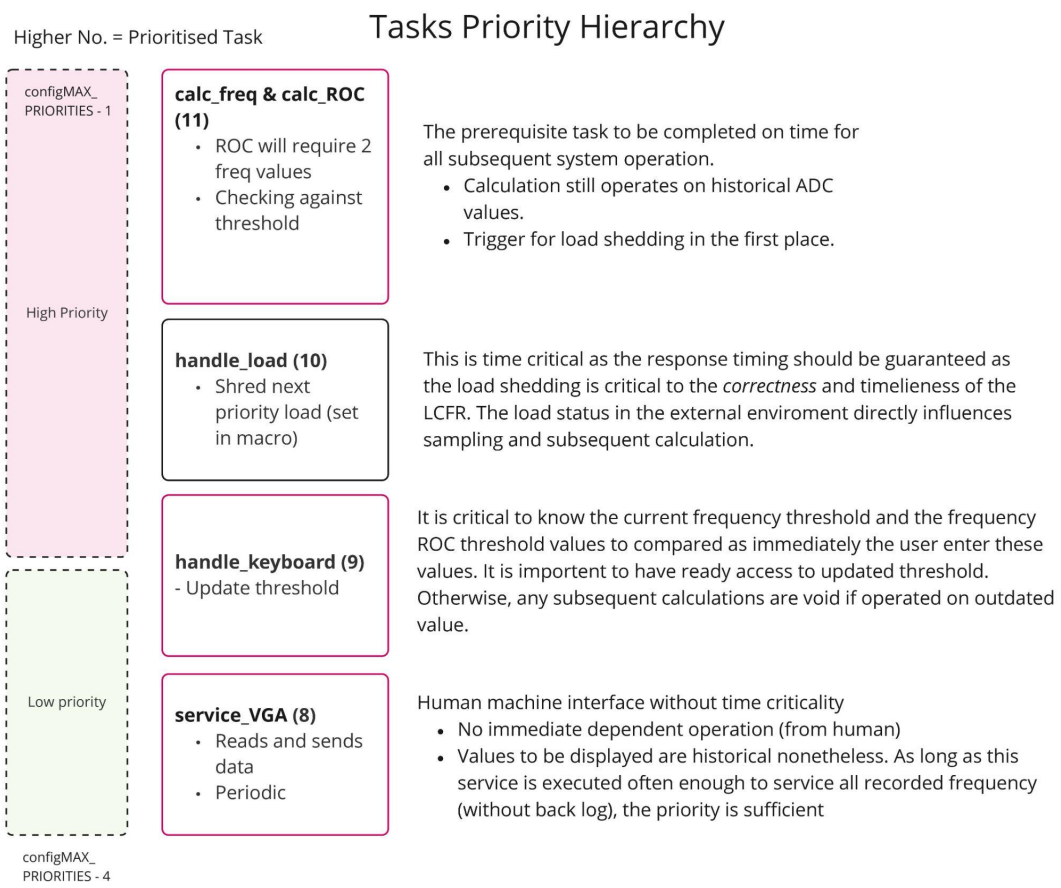

Fig. 3. Task Priority Hierarchy Chart

Issues with Application Design

Potential Failure of Response Timing Logic

Sometimes the response time measurements come as short as 0ms, indicating that the end of measurement must've been waiting for the start of measurement instead of the other way around. This indicates that the gating method was not successful in ensuring that starting and stopping measurements are always synchronised in pairs (see Task Interaction Section for `freq_relay` ISR and `handle_load`).

Service_VGA's Display Design

We have chosen to also display the five most recent values of the frequency and ROC as digits. However, the frequency (2 s.f.) and the entirety of roc is barely readable when displayed in real-update time (approx. 20ms, as per the sampling rate). Hence values are skipped to hold characters to maintain readability, but the most recent values of the current display cycle are still being displayed.

A consideration is given to instead display the values in sets of fives — a "snapshot" of the buffer of each value which is updated every 1-2s to the latest values. However, while this improves the readability of the digits, the display will by technicality: not be the five most recent values (apart from the plot).

FreeRTOS Engagement

Evidence that you're engaging with FreeRTOS

Used FreeRTOS features

In this project, many FreeRtos features are used. Among them, queues, semaphores and software timers are the main additional features used.

Queues are used as the primary form of intertask communications. These are used to send messages between tasks, and between interrupts and tasks. These are the first-in, first-out (FIFO) systems where items are removed from the queue once read. And the used queues are:

- `*Q_freq_cal_values*`; where calculated frequency values are stored,
- *`Q_roc_calc_values':* where calculated frequency rate of change values are stored,
- *'Q_ADC_sample_values'*; where the raw values of ADC sample values are stored,
- *'Q_response_time'*,`*response_timing*`; where the most recent response times in ms are stored and
- `*ps2_keyboard_inbox*`; where the inputs of ASCII values from the keyboard are stored.

In this project, FreeRTOS semaphores are used. These are used as a signalling mechanism in which a task in a waiting state is signalled by another task. In the project counting semaphores such as `*ps2_keyboard_sem'* and `*freq_ADC_recived_semaphore*` are used as an event listener to 'give' and 'take' semaphore each time it processes an event. Also, this project has also used Mutexes. A Mutex provides a flag or lock used to allow only one thread to access a section of the code at a time. It blocks or locks out all the other threads from accessing the resource. The used mutexes are `*roc_queue_sem*`, `*fre_queue_sem*`, `*response_time_sem*` and `*global_system_status_sem*`. Finally, binary semaphores are also used to control access to a signal resource. These are `response_timer_end_gate_sem` and `response_timer_start_gate_sem`.

Among FreeRTOS features, software timers are used to allow a function to be executed at a set time interval or simply to know the system uptime. The verification_timer function is used in the project as a callback function. Once the function is called and 500ms has expired a `*verification_elapsed*` flag is set to indicate to the handle_load task to shred the next priority load. The other software timer used in the project is in the `boot_timing()` function to get the system uptime. This function utilises the `xTaskGetTickCount()` to get the system uptime in milliseconds and calculates seconds, minutes, hours and days afterwards.

Task Interaction

`calc_fre_roc` and `service_VGA`

Tasks such as `calc_fre_roc` and `service_VGA` readily interact through a data queue along with their mutex semaphores. `calc_fre_roc` produces data to be sent through the respective queue dedicated for sharing values with `service_VGA`, who empties the queue. Loading and emptying data in respective queues are mutex guarded to ensure mutually exclusive access so that the data is ordered as they are captured by the ADC.

`calc_fre_roc` and 'handle_load'

In performing the performing the load management functionality, `calc_fre_roc` and 'handle_load` interacts through the global 'global_sys_status` variable and the `is_varification_elapsed` flag.

`calc_fre_roc` is responsible for detecting violation of the threshold and respectively setting the system state as UNSTABLE (if the current status was NORMAL) or STABLE (if the current status was NORMAL) when threshold is no longer violated.

'handle_load' is the only task responsible for resetting the system to the NORMAL operation state after confirming the reconnection of all loads (that are not manually switched off). All three of the `threshold_monitor`, `shed_load` and `connect_load` parts of their respective tasks are delayed by the verification time.

'handle_keyboard`

Does not interact with other tasks, save from updating the mutex guarded threshold value, which is used by the 'threshold_monitor` part of `calc_fre_roc`.

`freq_relay` ISR and `handle_load`

It is noteworthy that the `freq_relay` ISR and `handle_load` must communicate with two binary semaphores to coordinate response timing measurements. The "start gate" semaphore gates the start of an instance of measurement and the "end gate" semaphore gates the end of the same instance of measurements.

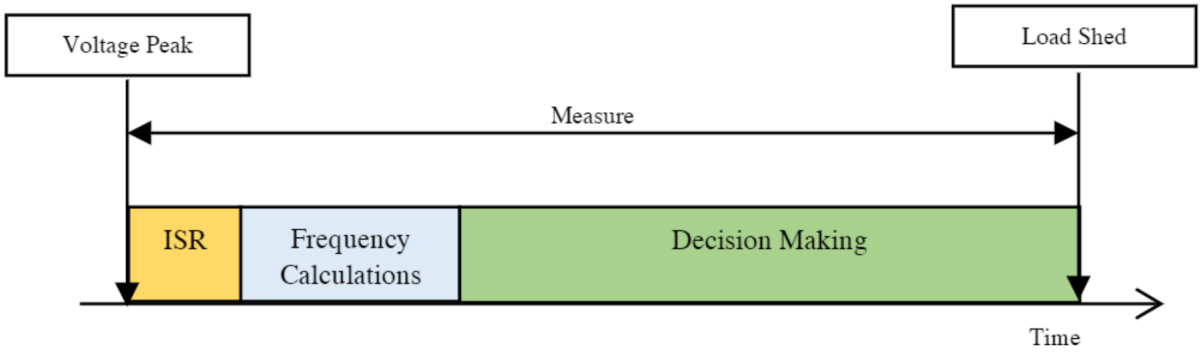

Fig. 4. A graphical representation of response timing measurement.

Limitations and Issues of Design

Multitasking Issues

Potential multitasking problems include:

- Race-conditions over commonly limited data.
    - There should not be any race condition issues as all data value queues are produced and consumed in pairs. Meanwhile control signals such as `global_sys_ststus` are only written by `handle_keyboard`.
- Critical sections over shared resources (writing only)
    - Any potential critical section access is prevented with mutex semaphores (see Use of Mutex Semaphore section above).
- Busy waiting (handle_load)
    - There are many situations where a task would be busy waiting over semaphores. As a majority of our tasks are event based, this cannot be resolved. But the wait time is reduced as much as possible with appropriate priority setting, priority inheritance from mutex and as short as possible critical sections by design.

Limitations

`ps2_isr` & Reading key Values

Currently, reading and interpreting an incoming key value are both done in the keyboard ISR. We have attempted to send the key values fetched over a queue to be interpreted in the `handle_keyboadr` task, which have resulted in the null operation of the ps2 keyboard as it fails to perform its initial self-check. Unable to reconcile with the self-check logic which are unknown to us (at least we know that it sends a self-check MAKE_CODE, but we are unable to decipher or handle it), we have relocated the interpretation logic inside the ISR, which seemed to operate as intended. If possible, we still wish to relocate the interpretation logic outside of the ISR to reduce ISR execution time.

Infinite Loop Stalls

Some tasks planned to be event-driven are implemented in such a way that they do not wait on a semaphore and instead run in a loop that stalls the system. To resolve this, the tasks were introduced with a short delay through `vTaskDelay()`. However, this is not the intended behaviour the task should introduce and will require reformatting that task to be event driven (if development time allows).

Work Partition

Organisation

All work on tasks are created and developed independently and aggregated together by Hao Lin (who interfaces with the de2-115 Board).

Work Delegation

Hao Lin
Have access to the de2-115 board.

Setup Pheripherials

- Pheripherials Setup (2 weeks)
  - 3 Push Buttons (KEY1 to 3)
  - 7x green LEDs (LED0 to 6)
  - 5x red LED (LED0 to 4)
  - Ps/2 Keyboard
  - VGA

- Timer Setup (2 hours)
  - Response Timer (200ms) (1u TIMER 0)

Task Creation & Multitasking

- Peripheral Serving Tasks (1 week)
  - `handle_keyboard`
  - `service_VGA`

- Load Implementation & servicing Tasks (2 days)
  - `handle_load` task
  - `shed load` function
  - `connect_load` function

- Threshold monitoring function (3 hours)

Shared Documentation & Report Writing

Chamith Nanayakkara

- Frequency calculations (2 days)
  - Frequency calculation
  - Frequency ROC calculation

- Timing Calculations (3 days)
  - Verification Timer (500ms) (FreeRTOS Software Timer + Callback)
  - Boot Timer (FreeRTOS Software Timer + Callback)

Shared Documentation & Report Writing

Assignment 1 COMPSYSY 723

Read-only view, generated on 19 Apr 2022