COMPSYS 723 Project Report
Designing and Implementing a Low-cost Frequency Relay

Chang Liu `cliu712`

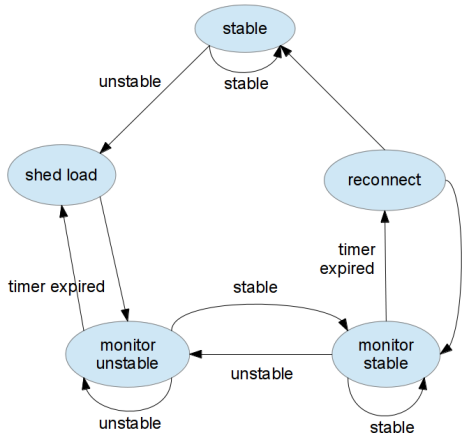
April 29, 2014

1 The State Machine

The most crucial and thus challenging part of our design is the functional model of the frequency relay. The frequency relay must have a functional behaviour that is dictated by the assignment specification. To summarize, the frequency relay must monitor the stability of the power network, the stability being defined as satisfying both criteria (the frequency and the rate of change in frequency). If the network becomes unstable at a certain point in time, the frequency relay shall start to switch off loads, from the lowest priority load to higher priority ones. It is assumed that the network would recover itself if enough loads were switched off. Once the network has recovered, the frequency relay has to reconnect the loads to maximize the utility of the power system.

It is apparent from the functional description above that our system must implement some sort of a state machine. It is, theoretically speaking, possible that the entirety of our system be implemented as a big giant state machine. Doing this, however, would nullify one of the primary purposes of this assignment as part of a course on real-time operating system. We therefore implement the state machine as a FreeRTOS task, the `state_machine` task, in file `state_machine.c`. The task would in turn interact with other parts of the system, the most notable interaction being sleeping on the frequency analyser interrupt. This is how we make use of the multitasking facilities of FreeRTOS in this design — by having one 'main task' doing the job and several utility tasks providing support to the main task, their communications and synchronisations coordinated through the use of queues and semaphores.

The state diagram of our state machine is as follows:



The system is in the '`stable`' state when it is first powered on. It will try to read the network state by calling `is_network_unstable()`, implemented in `freq_irq.c`. If stable, the system will stay in this state. Otherwise, loads have to be shed. Therefore the state machine jumps to the `shed load` state. Note that calling `is_network_unstable()` is blocking. This will imply that each time the state machine task calls this function, it sleeps until the frequency analyser interrupt is triggered and provides it with new frequency data to determine if the network is stable or not. As the frequency analyser interrupt is triggered periodically, it is in this sense that the state machine task is 'driven' by the frequency analyser interrupt — the frequency analyser interrupt provides the 'tick' that drives the state machine to move forward in time.[1]

Once the system is in the `shed load` state, it will try to disconnect load from the power system, by calling function `try_shed_load()`, implemented in the file `load_control.c`. This function will shed the low-

[1] The only exception to this is that the expiration of timers described below can trigger the jumping of state `monitor stable` and `monitor unstable` to `shed load` and `reconnect`. This is not driven by the frequency analyser 'tick' but is asynchronous.

est priority load if there are still loads that are connected. Otherwise it is equivalent to a no-op. Status for each load is represented by the LEDs. The LED states are managed by the load control task, which is implemented in the file `load_control.c`. The details will be discussed in a later section.

The `shed load` state will then reset the timer, and jumps to the `monitor unstable` state. The term '`monitor unstable`' simply means that in this state, the power network is assumed to be unstable and is being monitored for a possible change of stability. In this state, the system will keep monitoring the stability of the power network, again by calling `is_network_unstable()`. If the power network is still unstable, the state machine remains in this state until the 0.5 sec timer (initialised before entering this state (from `shed load`)) expires, in which case the system jumps back to `shed load` and will try to shed load for a second time. This continues until all the loads are shed (if the network continues to remain unstable). In other words, if the network remains unstable for a prolonged period of time, the state machine will jump between `monitor unstable` and `shed load`, the jumping being separated by 0.5 sec.

However, immediately after the frequency analyser has declared the power system stable (by returning `false` in `is_network_unstable()`), the system will reset the 0.5 sec timer, and jumps to the state `monitor stable`. Like in the state `monitor unstable`, being in this state indicates that the power system is assumed to be stable but is under constant monitoring for a possible change in its stability. If the power system becomes unstable again, our system immediately resets the timer and jumps back to the previous state, `monitor unstable`. This is to ensure that during this turbulent times of our beloved power system, the frequency relay can minimize instability and maximize utility.

If the power system has remained in the stable state long enough (again indicated by the expiration of the 0.5 sec timer), the system jumps to the `reconnect` state. In this state, the system will try reconnecting the loads. If there are loads to be reconnected, it reconnects the highest priority one, resets the timer, and jumps back to `monitor stable`. However if no loads are to be reconnected, it jumps to where the system starts — the `stable` state.

Note that the system has a maintenance state which essentially pauses the entire state machine (and tasks that sleep on it; we will discuss this in later sections). This is implemented in an ad-hoc way by adding an `if` clause in the jump table for the state machine. The implementation details will be discussed in the next section.

2 Task Decomposition

There are four FreeRTOS tasks in this project: 1) the main task `state_machine`; 2) the task controlling the states of red LEDs, `red_led_control`; 3) the `threshold_control` task, responsible for reading the keyboard input and updating the frequency and ROC threshold; 4) the `vga` task, for updating vga frame-buffers. Communication is done through static global variables, protected by semaphores and accessed via getter and setter functions. Synchronisation between interrupt service routines and tasks is done with queues. Synchronisation between tasks is implicit through the access to global variables. Access to global variables is blocking and serves as the synchronisation points between tasks.

The rationale for this scheme of decomposition is multifold. The main task, as is alluded above, implements the state machine of our system and has to sleep on the frequency analyser interrupt as latter is responsible for providing the state machine with stability information on the power system. It is given a higher priority than all other tasks, simply because our system has to satisfy a timing constraint — the load shedding has to happen within 200 ms from the onset of instability. As the state machine task is the only task with this priority, this makes sure that the system respond as quickly as possible.

The specification dictates that the states of the red LEDs are controlled by both the states of the loads and the slide switches. As there are no interrupts associated with these slide switches, a task must constantly monitor the states of the slide switches. This is why we decided to add a `red_led_control` task to our design. This state is implemented in file `load_control.c`, along with other utility functions to manage load states. As the updating of LED states happens in the `state_machine` state, the array of red led states must be made a global variable. As access to global variables in different tasks/threads can lead to race conditions and thus data inconsistency, they must be protected by semaphores. As these accesses are common to and frequently done in different functions, it is a better software engineering practice to make them into convenient getter and setter functions, and make the global variables themselves invisible to external functions. This is done by making

these global variables `static`.

The presence of the `threshold_control` task is entirely due to the limitation of the interrupt context. As the thresholds for the frequency and the rate of change are set by the keyboard (and thus in the PS/2 interrupt) and accessed in the main `state_machine` task, they must again be made into semaphore-protected static global variables. However, semaphores are inherently incompatible with interrupt contexts. The reason is simply that interrupt service routines cannot sleep. As we turn off interrupts globally before entering an interrupt service routine (this is ensured either by the hardware itself or BSP software code), sleeping in an ISR will result in an indefinitely block of the entire system — no tasks will be able to wake up the ISR, and neither can any ISR because interrupts are disabled! We thus reached a dilemma — we want to use semaphores to ensure mutual exclusion, but we can't sleep in an ISR!

The solution for this problem is to divide the ISR into two parts — to use the terminology from the Linux kernel [1], the top half and the bottom half. The top half is the actual ISR, that runs in the interrupt context — it must be fast and non-blocking, otherwise the system will suffer from serious lagging or worse, complete deadlock. This means that the top half will only do the absolute minimum of processing the interrupt request and must not call any function that might be blocking. The rest of the work is done by the bottom half. The bottom half typically runs in the task context and therefore has much relaxed timing and blocking requirements than the top half. It can sleep on semaphores and can do lengthy computations without adversely effecting the responsiveness of the system.

It is in this light that we created the third task in our design, the `threshold_control` task, implemented in the file `threshold_control.c` along with the PS/2 ISR and the getter function of `current_threshold`. The duty of this task is simple — it simply sleeps on the PS/2 ISR and wait for it to send the user command, interpret the command, and publish the updated threshold to the static global variable `current_threshold`, which is accessed in the main `state_machine` task via the getter function `get_current_threshold()`.

The final task in our design is the `vga` task. The reason for it is obvious — we need a task to update the vga framebuffer. However, its interaction with other tasks is more subtle and warrants discussion. To draw the information on the screen, the `vga` task needs to know the current and past history of the frequency and the rate of change of frequency. It also needs the frequency and ROC thresholds in order to display them in the screen (as lines in the plot and in the textual form). The former problem is solved by having the `is_network_unstable()` function doing an additional job — publishing the frequency and ROC information to the vga task, by calling `post_freq_to_vga_buffer()`. The latter is solved by calling `get_current_threshold()` directly. We see here our design decision to have a getter function to access `current_threshold` (instead of accessing it directly) substantially simplifies our code — we no longer have to worry about using semaphores to ensure mutual exclusion; we just call the getter function.

3 Implementation Notes

3.1 Synchronisation and Communication

As is alluded above, the synchronisation and communication between ISRs and tasks are implemented with FreeRTOS's queue API. `freq_queue`, defined statically and globally in `freq_irq.c`, has a twofold function: to synchronise the `state_machine` task and the frequency analyser irq, and to pass the frequency data read from the frequency analyser to the state machine task. The frequency analyser ISR, `do_freq_irq()`, publishes data read from the hardware to the queue, by calling `xQueueSendFromISR()`. The state machine task, by calling the function `is_network_unstable()`, reads this data from the queue. Reading is done by calling `xQueueReceive()`. Consequently, reading an empty queue will block the task calling `is_network_unstable()`. In other words, the task will sleep on the queue until next frequency analyser interrupt arrives, essentially synchronising the two events.

Likewise, for the keyboard irq, we have `threshold_queue` to synchronise the PS/2 interrupt and the `threshold_control` task. The `threshold_control` task will wake up on the PS/2 interrupt and update the current threshold for the frequency and the rate of change. As is discussed in detail above, the current threshold is protected by a semaphore and accessed by external tasks through utility functions `is_off_threshold()` and `get_current_threshold()`.

Synchronisations between tasks are implicitly

done through communications. Communications are done through semaphore-protected global variables. For example, `red_led`, a `unsigned long` defined in `load_control.c`, is used between the `red_led_control` task and `update_led()` (called from the main state machine task) to exchange the states of the red led. The functions `try_shed_load()` and `try_reconnect()` internally use the boolean array `load_states` to store states of each load, and generate the appropriate masks for the green leds and the red leds. The green leds are solely determined by the load states, and therefore can be updated directly in the main state machine task (which calls `try_*()` functions). However, the red led states are co-determined by the load states and the slide switches, and therefore cannot be directly updated. We solve this by having the function `update_led()` publishes updated red led states (before doing an `&` with the slide switches mask) to the global variable `red_led`. It is then read in the `red_led_control` task to make the actual update to the red led states. Mutual exclusion is ensured through the use of a semaphore, and this synchronises the two tasks (ie. they cannot execute the same portion of the program at the same time).

Global variables accessed by two different tasks need semaphores to ensure mutually exclusive access. However, there are cases when the global variables accessed by different tasks need not be protected by semaphores. An example would be the boolean variable `is_maintenance`, defined in file `state_machine.c` and is used to indicate if the system is in the maintenance state or not. What makes it special is that the only read-write cycle is in the button interrupt service routine. It is not written but only read in the state machine task. As we turn off global interrupts when entering an ISR, we essentially make an ISR a big giant synchronisation block — no tasks will be able to interrupt the data access sequence as they will not be scheduled! In fact the scheduler won't be able to run until we exit the ISR. Therefore if a global variable is only written in an ISR, no mutual exclusion mechanism needs to be applied to it as there is no possibility for data inconsistency.

The other example is `is_timer_expired`, defined in `timer.c`. It is both read and written in a task context. However, as the variable is only a boolean variable and all accesses to it are either a simple read or a simple write (no read-process-write cycle), they are atomic in the sense that no inconsistent intermediate state is possible in these simple cases. Additionally, as our timer is one-shot, our state machine algorithm ensures that the timer is only activated in the state `monitor stable` and `monitor unstable`. Therefore, `reset_timer()` and the timer callback will not be called at the same time. There is simply no chance for possible data races.

3.2 Priority Inversion

We do not have the problem of priority inversion in our design. This is thanks to a simple design principle: do not use semaphores to protect *code*; use semaphores to protect *data access*. This is enforced through the use of getter and setter functions: as they are the only functions directly taking and releasing the semaphores, they are made as small as possible to minimise the time spend during the critical regions. Therefore even if a higher priority task is blocked by a lower priority task due to a semaphore take, it will not be blocked for a prolonged period of time, simply because the critical regions are small. Thus lower priority tasks will never take an excessive amount of cpu time that is supposed to execute the higher priority tasks.

3.3 Flickering and VGA Back Buffer

We do have a problem in our final frequency relay — the screen is flickering. However, this is not a design failure but is simply a hardware bug, as we properly implement double-buffering in the vga task. To implement double-buffering, we do not draw to the front-buffer. Instead, we do all our drawing to the back-buffer and swap the front- and the back-buffer after all the drawing is done. The hardware is supposed to properly do the actual swapping during the refreshing of the display. However, there seems to be a hardware bug that prevents this from happening. We are confident that once the hardware bug is fixed, our product can deliver a flicker-free user experience.

The drawing to back-buffer is implemented in the function `draw_to_back_buffer()`, in file `vga.c`. The drawing function iterates over `vga_freq_array[]`, and calls `draw_freq_roc_line()` for each pair of points. The latter function is a modified version of the frequency relay example supplied in the example code zip file. The vga task then calls `alt_up_pixel_buffer_dma_swap_buffers` to swap the front- and the back-buffer.

Note that the hardware does not implement double buffering for the character buffers. Therefore drawing

to the character buffer is not double-buffered. It is done after the drawing of the pixel buffer, by calling function `draw_characters()`, which implements the drawing of characters.

References

[1] Linux Kernel Development (3rd Edition), Robert Love, 2010