FreeRTOS Concepts Used

Priority

Priority is a relative characteristic of a task represented by a number. At the context of an individual task it may appear meaningless, because it is relative to other priorities from other tasks. A higher priority number means higher priority. A task with higher priority over another task means the scheduler prefers to run and will run that higher priority task over the lower priority one whenever they are both ready i.e. in ready state. Or, if the lower priority task is running and the scheduler sees the higher priority task is ready, it switches out in favor of the latter.

Tasks

Tasks are the basic unit of execution in RTOS'es like FreeRTOS. It has its state, own context, own stack for keeping local variables, and a function/routine that runs infinitely. Whenever the scheduler context switches, that function does not exit nor the task ended; it's context via MCU registers, stack pointers, are saved into memory and restored later when the tasks is run again. With that, the task resumes from the point when it was context switched and is never aware that it was placed in suspended animation. So all local variables' lifetime is indefinite and practically never destroyed as long as the task is running except perhaps in some rare cases where the task has to exit.

Tasks are where users place application and processing code that are subset of an overall system to accomplish outputs or goals.

Queue

Queue is one of FreeRTOS mechanisms for inter-task communication. The queue is FIFO, and the length and size per element can be set using an API. One or more tasks can be producers and at the other end one or more tasks can consume data. One good thing about queues is that tasks can pass data without having the need to rely on global variables – which is a messy way to handle data. By not using global variables, tasks can have a data that is visible only to them.

FreeRTOS Ticks

Ticks are periodic events where the scheduler code runs instead of the current task. The implementation may differ across architectures. In ARMv7 MCUs, the ticks are triggered by Systick Interrupt, meaning the ticks operate in interrupt context. Within that interrupt, scheduler code runs for some checking. If there is nothing urgent, the interrupt will return and the current task will resume.

The tick also provides the RTOS some basic form of time tracking especially useful for timers and delays.

Context Switching

Context switching is when the running task gets replaced by another task in running the CPU. Either the FreeRTOS kernel/scheduler pre-empted the task, or the task voluntarily gives up the CPU.

Different architectures implement context switching variably. In ARMv7, pre-emptive context switching happens during ticks. Based on the explanation above, the scheduler code runs during ticks courtesy of systick interrupt. Scheduler then checks if there are other tasks of higher priority over the current one. If there is, it triggers a PendSV interrupt (a SW interrupt) and within that context it switches the registers for the running task tbe saved to memory and loads the registers of the to-be-run task. Then that new task resumes running. If there is no reason to context switch, systick interrupt scheduler code will exit without calling the PendSV handler.

Blocking/Non-blocking wait
In some cases a code portion is unable to proceed since it is waiting for a requirement first, like a data to be read from somewhere like a hardware device. Non-blocking wait means the code will not return, will poll and busy-wait indefinitely until the requirement it waits for arrives. While it is simple and it works, the opportunity cost of busy wait if long is high when you can do other things instead while waiting.

The blocking wait is an alternative waiting method which OS provide as service. In contrast to non-blocking wait, this kind of waiting means the routine can pause and resume later when its requirement has arrived, effectively freezing the routine and allowing us to use the time for other tasks. We can say the waiting task is "blocked" from running and is placed in blocked state. The arrival of the requirement is marked by an event usually an interrupt, and so within that something must be done to track the task waiting for it and alert it that it can resume. The downside is that it is more complex as it requires intervention to forces outside of the routine/task like the kernel.

FreeRTOS has services that have blocking features. Like in queues. If xQueueSend() with non-zero waiting is called from the sending end and there is no available space, the task will be blocked at this point i.e. it will sleep. At the receiving end, if xQueueReceive() is called with non-zero wait time and there is nothing in the queue, the receiving task will be blocked at this point, and will resume from this point once data is available to consume.

These blocking features should only be used while in process context. There are codes that run in interrupt context, and so blocking APIs should not be used there, and usually there are separate APIs called from ISR context.

Timer
Timer is a service that provides a capability to run some routine after a certain time (in number of FreeRTOS ticks). Timer is still considered a task with its own priority. Like in tasks, when a timer service is created the user also provides a function. That function will eventually return so any local variable there gets destroyed, unlike in tasks. It is also not recommended to block wait/sleep within timers.

Idle Task
Idle task is a low priority task that is not created by the user. It is automatically created when vTaskStartScheduler() is called. It runs whenever no user task is running or all are blocked.

How the Tasks Work

Sensor Filtering

This task samples the temperature every 50ms. Temperature is not a fast-changing signal so the assigned sampling rate is not too high. It performs moving average filtering to temperature samples.

The temperature worked on here by all tasks is assumed to be in degrees celsius. The target range is 16.0 to 30.0 deg C. The temperature granularity is in the tenths of a degree Celsius (+/- 0.1 degC). To avoid having to use floating point numbers which can complicate things, the temperature number is represented as a 16-bit integer. So 16.0 degC is worked on by the system as 160, 25.5 degC as 255, and so on.

Once the task gets the average temp, it sends it to a queue, then go to sleep to be woken 50ms later. That is, it produces temp data every 50ms.

Button Input Handling

The task for this periodically polls the GPIO for the buttons and implements debouncing. Another implementation can be by interrupts or in the case of STM32, EXTI interrupts. However, button glitches are significant problems and these needs to be filtered so the debouncing feature is included.

The task runs, detects, and sleeps to be woken in 25ms. It therefore polls the 2 buttons every 25ms. The debouncing works such that if the button is activated (pressed) for 4 task runs, then it's a valid press. Lower than that and it will get filtered. When a press is detected, the temperature setpoint (represented same as in Sensor Filter task) is updated. The default setpoint from bootup is 20.0 degC. Each press of the UP button increments the setpoint by 0.5 degC, and DOWN button presses decreases setpoint by 0.5 degC. It has a local variable to store the tracked temperature setpoint. While it runs every 25ms, it sends the most recent setpoint to a queue every 50ms. Notice that when using xQueueSend(), the waiting time is 0 meaning it will not block even if there are no available slot in the queue. This is also unlikely to happen as the 3 tasks except logger all produce and consume data at the same rate – every 50ms.

Temperature Control

In this task, it consumes updated measured and setpoint temperature from the two queues mentioned above every run. Since it is put to sleep and awaken every 50ms, it runs, consumes data, and processes those every 50ms.

It stores a data structure containing the measured/actual and setpoint temperature, along the heater status. If actual temp is lower than setpoint, the task accesses a GPIO corresponding to a heater relay, and turns it on, otherwise the heater is kept off. In the NUCLEO board, to make it easy for visual inspection, the GPIO is connected to a BLUE LED. If heater is on this LED is lit up and vice versa. This data structure is being fed to a queue that is to be received by a timer service.

All queue waiting times are 0 so this task won't be put to sleep if no new temp data is available, it will just use the existing one.

Timer Service

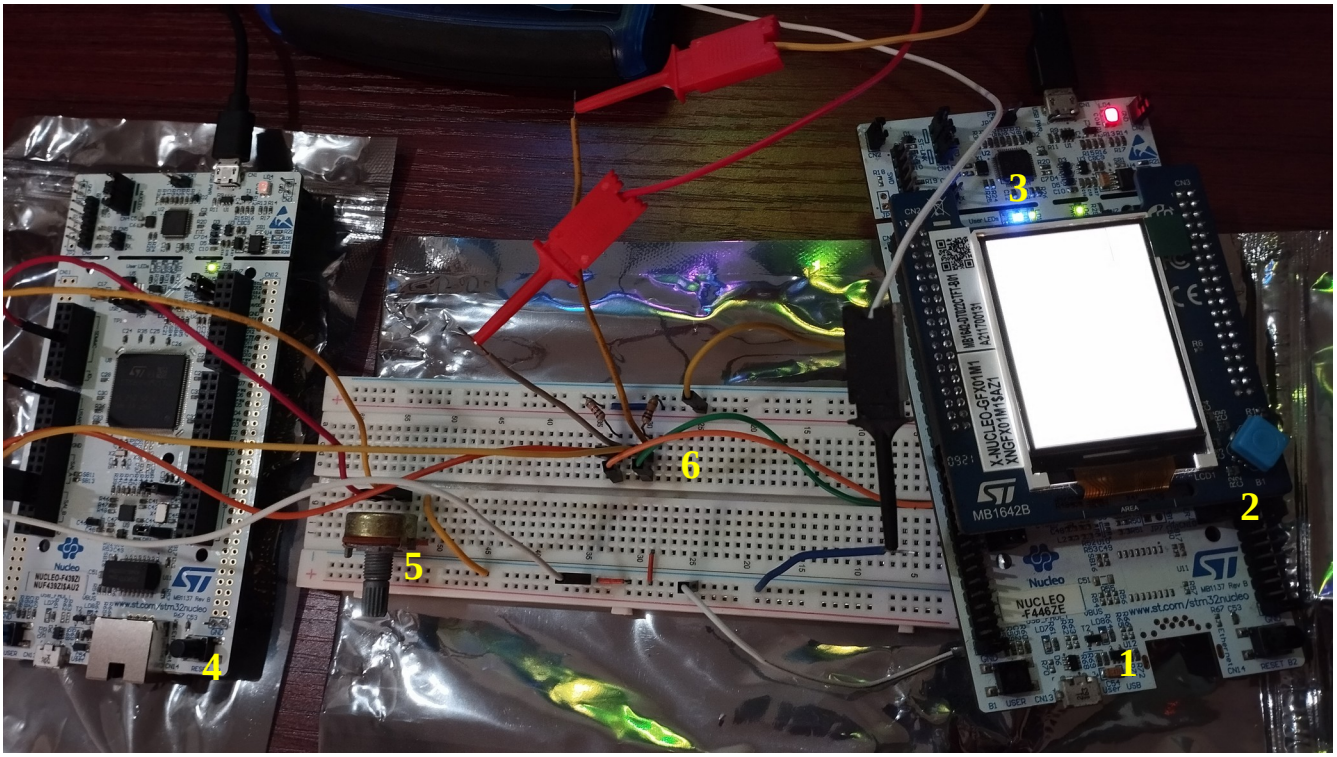
This service runs every 50ms. From the heater status it consumes from a queue that temp control task fed, it can determine the LED state. The waiting time for the queue receive was set to 0 since we don;t

want to sleep inside a timer. If heater is off, it blinks the green status LED by toggling it every 10 timer runs or 500ms.

Logger

The logger prints the actual, target temp and heater status. For this task to know it, it receives it from the last queue that is situated between the timer task and the logger task. The rate it posts to the UART is a lot different from the rate of other tasks. To do that, we exploit the capability of queues to block receiving task if there is no new data. To do that, timer service only sends data to the queue once every minute, and the waiting time for the logger task is at the maximum.

Hardware Prototype



1 – Main MCU
2 – Buttons (Up & Down)
3 – Status LED (Green) and Heater GPIO (Blue LED)
4 – Slave MCU that simulates the I2C Temp sensor
5 – potentiometer for controlling the temp output of the slave
6 – I2C link between the two devices

Challenges Encountered

I2C Issues

No I2C temp sensor was available, so I decided the I2C slave device would be implemented by another MCU with its temp output controlled by a potentiometer connected to that MCU's ADC. With that, I should be able to simulate various scenarios like increasing or decreasing the "actual" temperature going to the main MCU rather than an actual temp sensor whose temp readings I cannot control.

The problem is that initially this setup didn't work at first, as the master always get 0 temp data, so some debugging must be done. One concern is that I didn't know which of the two MCUs is at fault or whether both of them are at fault. So some isolation debugging must be done. I reviewed I2C specification to know what to expect. I also used a known working I2C device as a reference to be connected to the I2C master MCU (the one with FreeRTOS) and from there proceeded to fix code issues one by one. I also had to use my USB logic analyzer to verify the master indeed sends I2C packets and if correct ones are being sent. With that fixed, I proceeded to connect it to the slave MCU knowing the master is functioning correctly for the case we are testing.

FreeRTOS Build Issues

Building the project code which is quite complex and large due to FreeRTOS for the first times resulted in some build errors that can be traced to FreeRTOS configuration, not on my code (at least indirectly). Some build errors I encountered are that the heap is not enough, some FreeRTOS functions can't be linked, etc.

To fix this, I have to read the error messages and understand its contexts, along with researching from the web for similar problems. For the heap problem, I simply increased the heap size in the config file, as well as a define switch in the config file to fix the link errors.

Stack Overflow

Once the firmware is run in the MCU, chances are it can crash and apparently there are no useful clues what happened, which part of the code it happened, and so on. Based on my previous experience with FreeRTOS, when such wicked errors appear, consider stack overflow as a possible suspect. And so when I increased stack size of all tasks to 512 words, the firmware now run smoothly.

Refactoring and Modularization

This is perhaps the longest aspect of the development, and I came here after making the initial code working. Separating the tasks into different files and abstracting the devices from its device drivers to make it easily portable to a different hardware require so many code changes that it broke the initial working code.

What I did is to take things one-by-one and not all at once. I started abstracting one device like buttons, tested if it works, then if it did I commit it immediately so that I have a working fallback should next refactoring fails. I did this for the next items – other abstracting of devices, breaking up a huge file with several tasks into several modules – until refactoring is completed and the resulting project is still working.

For Improvements

I don’t have much to say, I think one thing I can suggest is to add diagnostic code that will enable production to test the system before deployment. It can also be useful for test engineers or even developers during debugging. We can start with FreeRTOS-CLI to use an existing command line interface infrastructure for our application, then figure out what parameters can be tested.