Software Technology of Internet of Things

Budget Management Exercises: Getting the Most for Your Money

Aslak Johansen <asjo@mmmi.sdu.dk>

Mar 28, 2022

You have now reached level 4. Congratulations on the great accomplishment! It seems like only yesterday, you were a green level 1'er. It has been a joy to see your skillset grow and prosper. In this set of quests we take a step back from the radio work because it allows us to focus on how we can structure concurrent applications on our IoT devices. Surely, by now, you can appreciate how the followings quests would translate into a radio communication model?

Quest 1: Getting Started (5XP)

Let's start out slow, now that you are in the big league. Your first task is to revisit the answer to an old quest, and touch up one the code structure.

1. Make a copy of your solution for the first quest, *"Seeing Potential"*, of the *"Sampling Physical Phenomena"* quest line. You may give it a reasonable name like `sample-transmit`.

   Note: You can use the following application as a starting point should your own implementation have gotten lost:

   ```
   #include <stdio.h>
   #include "freertos/FreeRTOS.h"
   #include "freertos/task.h"
   #include "driver/adc.h"

   void app_main(void)
   {
       adc1_config_width(ADC_WIDTH_BIT_12);
       adc1_config_channel_atten(ADC_CHANNEL_6, ADC_ATTEN_DB_11);

       while (1) {
           int reading = adc1_get_raw((adc1_channel_t)ADC_CHANNEL_6);
           printf("%u\n", reading);
           vTaskDelay(pdMS_TO_TICKS(1000));
   ```

```
    }
}
```

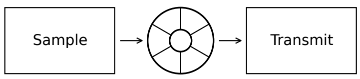
2. Update your codebase – should you need to – to do all your initialization in `app_main`, and have one function called `sample` that does the sampling and another `transmit` that sends out the value over the serial line.

3. Verify that codebase works as you expect it to.

Quest 2: Initial Pipeline (15XP)

Okay, that was perhaps a bit too easy. You are Software Engineers, I hear. That must mean you care about the structure that makes up the codebase and how it can be composed into patterns? Your code so far is essentially a two-stage pipeline, although it bears little resemblance to one structurally. Lets change that by converting each stage in the pipeline to a task and allow these to operate concurrently by placing a queue in between.

1. ***Starting Point*** Make a copy of your result from Quest 1 and call it `pipelined-sample-transmit`. The goal is to establish this setup:



2. ***Queue*** In this variant we will be working with a queue. To set it up you:

   (a) Include the `"freertos/queue.h"` header file. For some reason it needs to go *after* the `"freertos/FreeRTOS.h"` header inclusion.

   (b) Make a `TX_BUFFER_SIZE` precompiler definition with a value of `10`.

   (c) Declare a global variable `tx_queue` of type `QueueHandle_t`.
   Note: Bear in mind, as you make your way through this quest, that `QueueHandle_t` is typedef'ed to a pointer type.

   (d) In `app_main`, call `xQueueCreate` with `TX_BUFFER_SIZE` and `sizeof(int)` as parameters. Assign the return value to `tx_queueu`.

3. ***Sample Task*** Convert the `sample` function to a `TaskSample` task:

   (a) Replace the function prototype with the standard task prototype; that is, a `void` function that takes a `void*` parameter that is typically called `pvParameters`. Naturally, by doing this we remove the functions ability to transfer a return value. But that is okay: We are going to use the queue from step 2 instead.

   (b) In the `app_main` function we create the task by calling `xTaskCreate` with `TaskSample`, `"Sample"`, `4096`, `tx_queue`, `1` and `NULL` as parameters.

   (c) Back in the `TaskSample` function from step 3a we make a variable of type `QueueHandle_t` called `output_queue`. We then cast `pvParameters` to a `QueueHandle_t` and assign the resulting value to `output_queue`.

(d) Finally, we create a `while-true` loop with a body that does two things: Firstly, it contains your original code for performing an AD conversion. The value should end up in an `int` called `value`. Lastly you add the following line:

```
while (xQueueSendToBack(output_queue, &value, 10) != pdTRUE) ;
```

What does this `xQueueSendToBack` do, you may ask? It attempts to push `value` to the `output_queue` queue. If it fails to do so in `10` ticks time, it will return with some error code. `pdTRUE` indicates a successfull operation.

4. ***Transmit Task*** Convert the `transmit` function to a `TaskTransmit` task:

   (a) Replace the function prototype with the standard task prototype so that it looks like `TaskSample`.

   (b) In the `app_main` function we create the task by calling `xTaskCreate` with `TaskTransmit`, `"Transmit"`, `4096`, `tx_queue`, `1` and `NULL` as parameters.

   (c) Back in the `TaskTransmit` function we mirror the casting of `pvParameters` from `TaskSample`, only this time it should end up in a variable called `input_queue`.

   (d) Declare an `int` variable called `value`.

   (e) Finally, create a `while-true` loop with a body that does two things: Firstly receives a value from thw `input_queue` and assigns it to `value`. This is done by running:

   ```
   while (xQueueReceive(input_queue, &value, 10) != pdPASS) ;
   ```

   In this code, the call for `xQueueReceive` will block for 10 ticks or until a value can be received from the queue (whichever happens first). A return value of `pdPASS` indicates that a value was received, and no timeout occurred. Lastly, the loop body uses `printf` to output the contents of `value` to the serial line.

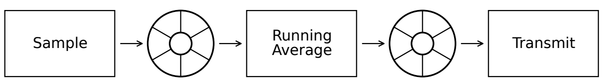
5. ***Cleanup*** Make sure that you have no lingering loop in `app_main` or left-overs from your original implementations of `sample` and `transmit`.

6. ***Verify*** Verify that the application does as you would expect.

Quest 3: Running Average (10^{+5}XP)

Still, the foundation is not quite there ... We need a generic stage. Something that consumes data from one queue and produces data to another. Lets explore how that fits into the setup by adding a stage that calculates a running average.

1. ***Starting Point*** Make a copy of your result from Quest 2 and call it `pipelined-sample-avg-transmit`. The goal is to establish this setup:



2. ***Second Queue*** Make a `avg_queue` that is defined and initialized next to `tx_queue`. Allow for this queue to have a size different from `tx_queue` by introducing a `AVG_BUFFER_SIZE` precompiler definition that maps it to a value of `10`.

3. ***Pipeline Stage*** A generic stage in a pipeline such as this is modeled as a task that has an input queue and an output queue. These queues represent the interface of the stage. Lets formalize that:

   (a) Use `typedef` to create a `struct` called `stage_interface_t` that consists of two `QueueHandle_t` fields called `input` and `output`.

   (b) Create a global variable called `avg_pair` of this type. It will play the role of host to the queues for a *running average* stage that we will introduce shortly.

   (c) In the `app_main` function, add the following line after the queue initialization lines:

   ```
   avg_pair = (stage_interface_t){avg_queue, tx_queue};
   ```

   This assigns the input and output queues to the newly defined variable. Make sure that `avg_queue` maps to the input and `tx_queue` maps to the output.

4. ***Running Average*** The next step is the write the code for the running average task:

   (a) Create a `TaskAvg` function with the same interface as any other task.

   (b) In the body of this function we first have to get our hands on the two queues:

      i. Declare a variable called `pair` of type `stage_interface_t*`, and initialize it to `pvParameters`. For this you must use a cast.

      ii. Declare a variable called `input_queue` of type `QueueHandle_t`, and initialize it to the `input` field of `pair`.

      iii. Similarly, declare and initialize `output_queue` to the `output` field of `pair`.

   (c) Still in the body we need to initialize the initial data structure:

      i. Near the top of the file, add a precompiler definition mapping `AVG_WINDOW_SIZE` to `7`.

      ii. Declare a variable called `buffer` of type `int[AVG_WINDOW_SIZE]`.

      iii. Next, this has to be cleared:

      ```
      memset(buffer, 0, AVG_WINDOW_SIZE*sizeof(buffer[0]));
      ```

      What this does is, starting at the `buffer` address in memory, for as many bytes as the buffer is large, assign the value `0` to that piece of memory.
      Note: `memset` is in the `string.h` header file.

      iv. Declare an `index` variable of type `uint8_t` and initialize it to `0`.

      v. Declare a `sum` variable of type `int` and initialize it to `0`.

   (d) Add this point you should add a comment stating `TODO: consume first AVG_WINDOW_SIZE-1 messages`. We will get back to this later.

(e) Finally, we need to implement the *service loop* of the task. This is a `while-true` loop with a body containing:

   i. ***Receive*** Receive a message over the input queue into an `int` variable called `new`. By now you should already have code in your hand that can be easily adjusted for this purpose.

   ii. ***Update*** The update step is a series of instructions:

      - Create a temporary `old` variable of type `int` to hold the value of `buffer[index]`.
      - Assign `new` to `buffer[index]`.
      - Decrement `sum` by `old`.
      - Increment `sum` by `new`.
      - Update `index` by adding one to it, and then do a modulo `AVG_WINDOW_SIZE` to make sure that it wraps around.

   iii. ***Output*** Declare an `int` variable called `value` and assign it the value of `sum/AVG_WINDOW_SIZE` and insert this into the output queue. You should also have code in your hand that can be easily adjusted for this purpose.

(f) In the `app_main` function, create a task by calling `xTaskCreate` using `TaskAvg`, `"Avg"`, `4096`, the address of `avg_pair`, `1` and `NULL` as parameters.

5. ***Verify*** Verify that the application does as you would expect.

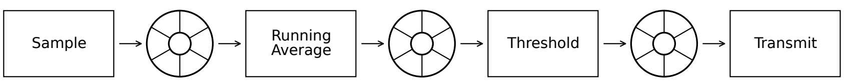
6. ***Perspective*** Is it always a good idea to place the calculation of a running average in a separate task?

7. ***Bonus*** An additional 5XP will be granted should you choose to revisit the comment from step 4d. Here, you should add a loop that consumes the `AVG_WINDOW_SIZE-1` first messages on the input queue and update `buffer`, `index` and `sum` accordingly. Why is this attractive?

Quest 4: Thresholding (10XP)

The running average keeps the noise of the signal down, but we are transmitting the same number of values; more or less. Let's introduce some adaptive sampling into the mix by only transmitting values that deviate more than some threshold from the last transmitted value.

1. ***Starting Point*** Make a copy of your result from Quest 3 and call it `pipelined-sample-avg-threshold-transmit`. The goal is to establish this setup:



2. ***Queue Preparation*** Because we are adding one stage to the pipeline we need another queue. Lets call it `thres_queue` and size it according to `THRES_BUFFER_SIZE`. Additionally, we will need a `stage_interface_t` called `thres_pair`, and to make sure that the stage interfaces for both the running average task and the thresholding task properly properly chain the queues.

3. ***Threshold Task*** Implement the task by following these instructions:

   (a) Make a precompiler definition of `THRESHOLD` to a value of `100`.

   (b) Add a `TaskThreshold` function and make sure that it is created in `app_main` and fed a pointer to the right stage interface.

   (c) In `TaskThreshold`, create two `QueueHandle_t` variables – let's call them `input_queue` and `output_queue` – and make them refer to the input and output of the received stage interface.

   (d) Declare an `int` variable called `last` and initialize it to a value that is at least `THRESHOLD` higher than the highest value the ADC can give you (e.g., `1«16`).

   (e) Then add a `while-true` loop with the following body:

      i. Receive an `int` into `current` from `input_queue`.

      ii. If the absolute difference between `current` and `last` is greater than `THRESHOLD`, then you add `current` to `output_queue` and assign `current` to `last`.

4. ***First Test*** At this point you should try out your application and – if your race conditions match mine – observe how it *doesn't* work.

5. ***Understand*** Read a description[1] of what is going wrong, and try to understand it.

6. ***Apply Fix*** Adjust the priority argument for every call to `xTaskCreate` from `1` to `0`. This should allow the idle task to get time and feed the watchdog for you.

7. ***Second Test*** Verify that the application does as you would expect.

[1] `https://github.com/espressif/arduino-esp32/issues/3871#issuecomment-913186206`