Notes for the FreeRTOS presentation

Bedřich Said: 409874@mail.muni.cz

May 17, 2020

Contents

1 Introduction

1.1 Slide 1 – 2

- Video time: 0:00 – 1:41
- The presentation contains these attachments:
  - These notes
  - The LaTeX beamer slides
  - Set of examples in C++ language
  - Recorded presentstaion as video
- Recommended reading procedure:
  - Look at the presentation slides and at these notes.
  - If you find something unclear, please see the corresponding slides on the video presentation.
  - You can find the video timing in the beginning of every slide in these notes.
  - You can check the source code of the corresponding example. There are written the numbers of corresponding slides for every example.
  - If you still have some questions or if there is still something unclear, please contact me in the discussion forum inside IS or via e-mail.

2 FreeRTOS and used hardware

2.1 Slide 3 – 5

- Video time: 1:41 – 2:42
- FreeRTOS can be compiled for several platforms.
- Some manufacturers already included support of FreeRTOS in their chips.
- For other hardware you have to port it and build by yourself.
- Here we selected ESP32 microcontroller.
- ESP32 is all-in-one chip controller with dual core 160 MHz 240 MHz CPU, 520 KiB SRAM, 4 MB flash memory, onboard WiFi or Bluetooth, FreeRTOS support and implemented Arduino API.

2.2 Slide 6

- Video time: 2:42 – 3:02
- Arduino Learning Kit Starter (ALKS) is an electronic board for beginners and contains several basic peripherals (LEDs, buttons, potenciometers, speaker, sensors, connectors).
- We will use the connected LEDs for visualization of the FreeRTOS tasks.

2.3 Slide 7 – 9

- Video time: 3:02 – 6:53
- Fast LED blinking cannot be seen by the human eye.
- We used oscilloscope for visualization of the LED behavior on the screen.
- The screen displays a plot, where X axis is time and Y axis is voltage.
- See the traffic lights simulation and its visualization on the oscilloscope (shown in the video).

3 Minimal working example

3.1 Slide 10 – 11

- Video time: 6:53 – 8:47
- The minimal full working program
- The red LED (L_R) is blinking after uploading the program to the ESP32.
- setup() – this functions is called once after boot
- loop() – this function is called periodically in a loop
- pinMode() – set the processor's pin as input (for buttons) or output (for LEDs)

- digitalWrite() – set the processor's pin as HIGH (positive voltage) or LOW (zero voltage)
- delay() – delay the time in milliseconds
- Documentation for all these functions can be found at www.arduino.cc
- vTaskDelay() – FreeRTOS specific delay function, this functions aloows to run another thread during waiting, documentation for this function can be found at the FreeRTOS webpage

3.2 Slide 12

- Video time: 8:47 – 9:14
- Print-screen of the Visual Studio Code editor with installed PlatformIO plugin.
- The minimal working example is displayed on the screen.

3.3 Slide 13 – 14

- Video time: 9:14 – 10:30
- Differences between different delay functions.
- Remember the difference between the number of RTOS scheduler ticks and the time in milliseconds.
- See the FreeRTOS documentation for details.

3.4 Slide 15

- Video time: 10:30 – 10:51
- Just a note: See the naming convention of the FreeRTOS functions, the most interesting are prefixes used for determination of the return type.

4 Tasks and jobs (New task creation)

4.1 Slide 16 – 17

- Video time: 10:51 – 11:28
- The tasks and jobs have the same naming convention as in any RTOS.
- Here we use the priority based scheduler.
- The task with the highest priority is executed.
- Remember that we have 2 cores on this CPU.

4.2 Slide 18

- Video time: 11:28 – 11:46
- When the task is running, it is switching the LED on and off all the time.
- We can see this changes (blinking) on the oscilloscope.
- This blinking reliably indicates that this specific task is running right now.

4.3 Slide 19

- Video time: 11:46 – 12:09
- Example of the complete program with creation and execution of the yellow task.
- The yellow task is blinking with the yellow LED.

4.4 Slide 20

- Video time: 12:09 – 12:22
- We can see the result on the oscilloscope.
- The yellow LED is blinking all the time with 220 ms long gaps.

4.5 Slide 21

- Video time: 12:22 – 12:29
- The screen shows the zoomed area with the gap.
- Inside this time gap the task is not running and the LED is kept on.
- The yellow area on the screen shows that the LED is changing its state very fast.

4.6 Slide 22

- Video time: 12:29 – 12:36
- The gap is present periodically every 5.24 s

4.7 Slide 23

- Video time: 12:36 – 12:45
- The program is running exactly 5 s between these two gaps.
- This is quite suspicious...

4.8 Slide 24 – 25

- Video time: 12:45 – 14:15
- In reality, the processor resets itself periodically after this 5 s
- The slide shows the boot procedure displayed using the serial terminal connected to the UART peripheral (serial port).
- **If the task is running for long time without interruption (when the jobs are infinite), the task will not trigger the watch dog timer on time.**
- **If the watch dog timer is not triggered, it is considered as failure and the processor performs reset.**
- The watch dog timer can be disabled, but I strongly don't recommend it.

4.9 Slide 26

- Video time: 14:15 – 14:38
- When the job is finished, the task is waiting until the next execution.
- The job is very simple here. The job only toggles the LED state (on/off).

4.10 Slide 27

- Video time: 14:38 – 14:48
- Imagine the red task with jobs and delays among them.
- Let's see whats happens...

4.11 Slide 28 – 30

- Video time: 14:48 – 16:04
- Now, the task is running with no gaps.
- The LED is on for 5 ms and off for the next 5 ms.
- The context switching and the time when the scheduler is running is hidden in this 5 ms delay.

5 Tasks and jobs (Task properties)

5.1 Slide 31 – 36

- Video time: 16:04 – 17:31
- Let's create a new task with no delay which is pinned to one specific CPU core. Let's disable the watch dog timer and let's see what happens.
- The task is running all time time and we can see that the processor is able to switch the LED on and off 226 ns. It is about 4425000 blinks during one second (4.425 MHz). The CPU frequency in this example is 160 MHz.

- It means, that the for loop and the two digitalWrite() functions need about 36 CPU instructions.
- But ...

5.2 Slide 37 – 41

- Video time: 17:31 – 19:01
- We found a gap long 8.6 µs.
- There is no CPU reset like in the previous examples, because the CPU reset takes much more than 36 instructions or 8.6 µs.
- This gap occurs exactly every 1 ms.
- The scheduler tick rate in this example is 1 kHz. So, the scheduler is executed every 1 ms.
- The gap shows exactly when the task is suspended and when the scheduler is running.

6 Tasks and jobs (Multiple tasks)

6.1 Slide 42 – 45

- Video time: 19:01 – 19:58
- Imagine multiple tasks (more tasks than CPU cores) with no delay. How the RTOS scheduler will handle this situation?
- Here we created red and yellow task, and the green task is running in the main thread.
- All tasks have priority one here.
- The red and green tasks are sharing the same CPU core, but the yellow task occupies the whole second CPU core.

6.2 Slide 46 – 47

- Video time: 19:58 – 20:29
- Let's change the priorities of these tasks.
- Now, the red task with priority 2 and the yellow task with priority 3 occupy both CPU cores all the time.
- The green task with priority one has never been executed.

6.3 Slide 48 – 49

- Video time: 20:29 – 21:03
- Now let's change the priorities again.
- The main therad (green task) still has default priority one.
- The red and yellow tasks now have priority 0.
- The result: The green task has the highest priority, so it is always executed.
- The second CPU core is occupied by the first executed red task with lower priority zero. So, the yellow task has never been executed by the scheduler.

7 Tasks and jobs (Jobs)

7.1 Slide 50 – 53

- Video time: 21:03 – 22:15
- Imagine a blinkLedForTime() functions, which emulates a periodic task with one repeating job. The job is running for `run_us` microseconds and waiting for `wait_ms` scheduler ticks (here equeals to milliseconds).
- Let's call this task with 300 µs running and 1 ms sleeping.
- You can see that the job is executed every millisecond. Why? Because the vTaskDelay() waits for one scheduler tick. In other words, the scheduler waits until the first scheduler tick after the job finished. The first shceduler tick occurs after 600 µs.
- This periodic task executes their jobs every 1 ms.
- But this is not guaranteed. For example, the situation is quite different immediately after boot as you can see on the oscilloscope.

7.2 Slide 54 – 55

- Video time: 22:15 – 22:56
- Let's change the execution time of the job to 900 µs.
- Now the first scheduler tick after 100 µs is missed.
- The next job is executed with the first unmissed tick after the next 1 ms.

7.3 Slide 56

- Video time: 22:56 – 23:15
- You can see other examples with different settings.
- Here 1500 µs execution time and 2 ms waiting time.

7.4 Slide 57 – 60

- Video time: 23:15 – 24:20
- Let's try to find when the first scheduler tick is still not missed and when the remaining time is too short.
- With 1500 µs the first tick is missed.
- With 1488 µs the first tick is not missed.
- With 1489 µs the first tick is sometimes missed and sometimes not.
- With 1490 µs the first tick is always missed.
- The best practice for handling the jobs inside the task is by using timers. See slide 65 for more information.

7.5 Slide 61 – 62

- Video time: 24:20 – 25:06
- Imagine 3 periodic tasks running on two CPU cores with different priorities.
- Try to think about what happens here and how the scheduler handles this work. I think that this is a good exercise.
- The best practice for handling the jobs inside the task is by using timers. See slide 65 for more information.

7.6 Slide 63

- Video time: 25:06 – 25:24
- I can't present all the supported functions and the whole FreeRTOS API.
- If you are interested in other task control related FreeRTOS functions, please refer the documentation of the FreeRTOS API.

8 Tasks and jobs (Timers)

8.1 Slide 64 – 67

- Video time: 25:24 – 26:59
- Timers are the best practice to handle jobs inside a periodic task.
- In most hardware architectures we have a limited number of hardware timers, but more tasks can be dependent on the same timer.
- See the FreeRTOS documentation for more information about supported API.
- The example shows a job running 600 µs and called by the timer every 2 ms.

9 Sharing data among threads (Critical sections)

9.1 Slide 68

- Video time: 26:59 – 27:40
- This chapter about data sharing is more about parallel systems in general. The real time systems mostly work with more CPUs, so thetheory of parallel systems is important here, too.
- I have selected only critical sections and queues, because they are probably the first structures you will need in any of your implementations with FreeRTOS. But remember that there are much more parallel data structures you can learn in other subjects.

9.2 Slide 69 – 76

- Video time: 27:40 – 29:54
- Critical section is a set of instructions between entering and leaving commands.
- The code inside the critical section cannot be interrupted and no other task can activate the same critical section handler when the first one is still running.
- You can handle memory writing and other shared resources using this parallel structure.
- Imagine a function with the critical section and the same function without it.
- Then imagine two tasks using the same critical section (red and yellow task).
- What happens after execution?
- Slides 73 and 74 show the situation without critical sections. One slide shows the situation immediately after boot and the second one somewhere later in time.
- Slides 75 and 76 show the same situation with critical sections enabled.
- See the difference: There are more tasks running at the same time in the first example (slides 73 and 74), but only one task is running at the same time in the second example (slides 75 and 76).
- The difference is caused by using of the critical sections.
- Imagine that you need to share a resource (e.g. the memory), then you need the critical sections to prevent writing to the memory by two or more processes at the same time.

10 Sharing data among threads (Queues)

10.1 Slide 77 – 78

- Video time: 29:54 – 31:14
- The parallel queues work in the producer/consumer layout.
- There are tasks that write (push) something to the queue – producers.
- And there are tasks that read (get) something from the queue – consumers.

- You can define the maximum number of elements that can be stored in the queue – the queue size.
- Imagine one thread reading some data from a sensor and another thread processing and writing the measured data to the persistent memory.
- Then the thread which is reading from the sensor is the producer (it writes the captured data to the queue) and the thread which is reading the data from the queue and processes them is the consumer.

10.2 Slide 79 – 82

- Video time: 31:14 – 32:22
- The example shows 3 tasks. One produces, the green task and two consumers, red and yellow tasks.
- The image on the oscilloscope shows when the task is writing the data (green task) and when the other two tasks (red and yellow) are reading the stored data from the queue.

10.3 Slide 83

- Video time: 32:22 – 32:55
- Here you can see the log of all these tasks. The log shows when the green task published the data and when the red and yellow task read the data from the queue.
- You can see very similar examples in exercises dedicated to parallel and distributed systems in general.
- This example is a screen capture of the serial terminal connected to the ALKS with ESP32 processor running this example.

10.4 Slide 84

- Video time: 32:55 – 33:25
- Thank you for your attention. I hope you enjoyed this presentation.
- In case of any questions please refer to the dedicated forum in IS or write me directly to the e-mail.
- Have a nice day!