Managing Firmware with FreeRTOS

S. Jaritz

Embedded Austria No. 3: Real Time OS

History of FreeRTOS

- Developed ca. 2003 by Richard Barry
- 2007 SafeROTS
- 2013 start integrating libs
- 2016 Amazon becomes the steward of FreeRTOS, licence becomes MIT
- 2019 FreeRTOS + C-SDK
- 2021 AWS IoT Express Link

Overview

- Available for almost all arch (arm, x86, risc V, pic, etc.)
- Low overhead
- Tick and tickless
- Preemptive and cooperative
- Hw tracer support (J-Link + Systemview etc.)

Why RTOS?

- multiple “processes/tasks” run at the same time (management of tasks)
- separation of peripheral + protocol & action (data streaming)
- priority based event handling(detecting and handling of HW and SW events)
- low power modes (= idle task/sleep/deep sleep)

Why FreeRTOS?

- one of the most common RTOS(its old 2003 and it has always been there) – tons of solved issues on StackOverflow, etc.
- lots of official and unofficial ports & HALs for platforms
- you can do your own port easily
- super light weight (you can tune it to run on HW with not much RAM, low cost MCUs)
- sw timer, events, co-routines, ringbuffers, memory pools, easy MCU (deep) sleep

Trade-offs

- FreeRTOS = kernel = scheduler
- tick based → every 1ms isr → prob D/I cache flush → task switch check → overhead(pre-emptive) vs task yielding(cooperative)
- each task has a task control block (extra memory)
- each task has its own stack

→ RAM & MIPS for management

Yielding example(msp430)

```
portSAVE_CONTEXT macro

    IMPORT pxCurrentTCB
    IMPORT usCriticalNesting

    /* Save the remaining registers. */
    push    r4
    push    r5
    push    r6
    push    r7
    push    r8
    push    r9
    push    r10
    push    r11
    push    r12
    push    r13
    push    r14
    push    r15
    mov.w   &usCriticalNesting, r14
    push    r14
    mov.w   &pxCurrentTCB, r12
    mov.w   r1, 0(r12)
    endm
```

```
portRESTORE_CONTEXT macro
    mov.w   &pxCurrentTCB, r12
    mov.w   @r12, r1
    pop     r15
    mov.w   r15, &usCriticalNesting
    pop     r15
    pop     r14
    pop     r13
    pop     r12
    pop     r11
    pop     r10
    pop     r9
    pop     r8
    pop     r7
    pop     r6
    pop     r5
    pop     r4

    /* The last thing on the stack will be the status register.
     * Ensure the power down bits are clear ready for the next
     * time this power down register is popped from the stack. */
    bic.w   #0xf0, 0(SP)

    reti
    endm
```

A deeper dive into FreeRTOS

Key Components

- Startup code (aka hw init)
- Start scheduler
- FreeRTOSconfig.h
- preemptive or cooperative multitasking
- memory pools
- external stacks/libs like TCP/IP, mqtt, aws IoT, MCUboot

General issues

- c-based kernel/scheduler does not really take care of MCU features
- To make it run with well defined behaviour the tick ISR need to have highest prio (nested intr, Intr controler config)
- Critical sections = en/disable tick ISR and the other ones
- MCU features can cause problems (like stacked/shadow registers – aka loop ISAs etc.) → your compiler does not know that it compiles code for task 1 and the other for task 2

Special Issues

- Atomic operations – single producer consumer FIFO as ringbuffer is much more performant than the generic ring buffer
- Stack overflow detection
- Gdb debugging is platform dependent
- In general no APIs like gpio, serial, eth, bt etc – own driver development closely coupled to the HW

Tips

FreeRTOS repro and sources

Do your own:

- Cmake file with boiler template for your toolchain
- Remove all unused ports from the git

Separate:

- Startup code, peripheral init/deinit/sleep code into HAL
- Setup INTR controller and assign manually INTR prios in one place
- Use static code for TCBs, semaphores, etc. that works very well for debugging and in case of stack overflow you can check the memory map file generated by the linker

Imho best practises

Decouple HW and the FreeRTOS managed process!

- **Use a RTC or your TIMER0 as sys time**
- **Create your own Ctest framework – the RTC/TIMER0 can be replaced by your own timer counter → you can write unit tests for your tasks**
- **Manage all aspects of a task – creation, init, run, finished, failure(with/out exception)**

Some tips – data passing

- **create your own ringbuffer for your platform (performance)**
- **Use your ringbuffer as the interface between peripheral/driver and task**
- **Use a counting semaphore to activate the task from ISR/other tasks (semaphores are ISR save and give you the best performance)**
- **Next lvl:**
  - make your ringbuffer compatible with the DMA controler(claim - finish)
  - Use the DMA INTR to set the semaphores

Some tips – state machines

- Implement your state machine as semaphore + function pointer(state function)
- A change of state triggers a set of the semaphore (use counting not binary so you can transition through several states without context switch)
- Timing behaviour like timeouts – use a SW timer

  magic formula = sem + sw timer + set of states as function pointer = best performance + no SM polling/event based SM processing

Some tips – timing and INTR

- Do critical timing via own INTR (HW timer with higher prio than FreeRTOS tick timer) → for your own safety
- Move non critical timing to FreeRTOS → relaxed fw development
- INTR are your friends! Think more in that way – the silicon manufacturer gives you a HW OS(via INTR controler) and you embed a SW OS by using one INTR
- HW always wins over SW but you can manage (debug from your periphals backwards into your firmware)

Some tips – arch data

- **Follow the data! Are they event based(random timing) or are the constant(streaming)?**
- **Stream:**
  - data → DMA in → ISR → semaphore ← processing task wait for
  - task → trigger DMA out
- **Events(non critical):**
  - ISR → ringbuffer & semaphore
  - Task → read ringbuffer till semaphore blocks
- **Events(critical):**
  - ISR = handler

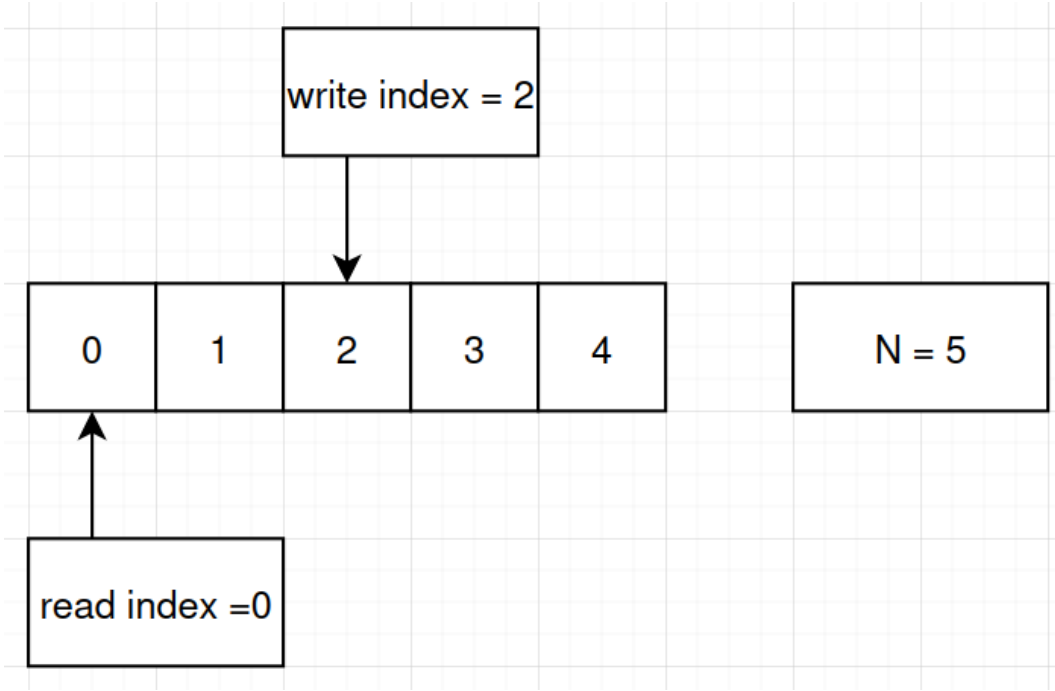
Some tips – arch processing

- Less is more! Avoid tasks
- You maybe need only one task that checks all ringbuffers of the UARTs – so UARTx ISR triggers the UART processing task via semaphore
- You may want to process all state machines in one task – think of a list of state machines(function pointer vector)
- You may want to design your data tasks as async worker queue – so you can put your system to deep sleep when there is nothing to do

Thanks – Let’s talk!

single producer – consumer ringbuffer – atomic ops 1/2

```
void util_rb_put(util_ringbuffer_t *rb, uint8_t *pSrc, const uint16_t Nbytes){
    uint8_t * pB;
    uint16_t am;
    const uint16_t wi = rb->wi; // local const copy of the write index
    const uint16_t wiUnWrapped = wi + Nbytes; // the end write index (not wrapped)
    pB = rb->pB + wi;
    // 1st write data till we need to wrap
    am = wiUnWrapped > rb->N ? rb->N - wi : Nbytes;
    for (uint16_t i = 0; i < am; i++) {
        *pB = *pSrc;
        pB++;
        pSrc++;
    }
    // 2nd wrap and write the rest of the data
    am = Nbytes - am;
    // still needs to write some stuff
    if (unlikely(0 != am)) {
        pB = rb->pB;
        for (uint16_t i = 0; i < am; i++) {
            *pB = *pSrc;
            pB++;
            pSrc++;
        }
    }
    // update write index
    rb->wi = (wiUnWrapped) % rb->N;
    // and amount of bytes stored at the buffer
    util_atomic_add(&(rb->am), Nbytes);
}
```



single producer – consumer ringbuffer – sync isr & task