Embedded electronics exam: running FreeRTOS on RISC-V

J.-M Friedt, June 27, 2025

The RISC-V architecture, an opensource/openhardware ISA (instruction set architecture), has become popular thanks to its efficient implementation in FPGAs as softcores, lack of commercial usage restrictions and flexibility, even becoming available as hardware mostly compatible with the STM32 ARM Cortex core with manufacturers including Microchip (PIC64GX), Renesas, Gigadevice (GD32VF103) or SiFive. It has for example replaced the LM32 softcore processor initially used in the White Rabbit time-transfer gateware, being in charge of the slow feedback loop controls and Ethernet message routing in the White Rabbit switches and nodes [1].

1. What are the software tools needed to port FreeRTOS to a new processor architecture? Provide a detailed list and how to generate these tools.

   Compiler, linker and software for communicating with the bootloader, possibly an emulator. Cross-compiling these tools is achieved with gcc+binutils+newlib or using a dedicated framework such as crosstool-ng

2. What are the software tools needed to run GNU/Linux to a new processor architecture? Provide a detailed list and how to generate these tools.

   Compiler, linker and software for communicating with the bootloader, possibly an emulator, as well as the operating system kernel, libraries and applications. Cross-compiling these tools is achieved with frameworks such as OpenEmbedded/Yocto or Buildroot to make sure a consistent framework is generated.

3. A new `Makefile` targeting the RISC-V architecture has been added in the `1basic` example of the `https://github.com/jmfriedt/tp_freertos` repository. The RISC-V compilers targeting 32 and 64 bit architectures are found in `/home/jmfriedt/x-tools`. Demonstrate how you can try and compile the `1basic` example for the RISC-V architecture

   `export PATH=/home/jmfriedt/x-tools/riscv32-unknown-elf/bin:$PATH` allows accessing riscv32-unknown-elf-gcc for compiling FreeRTOS targeting the RISC-V 32-bit architecture as stated in the Makefile with `CROSS_COMPILE =riscv32-unknown-elf-`

4. How can you identify and solve the compilation error?

   The error `undefined reference to '__freertos_irq_stack_top'` is identified with `grep -r "__freertos_irq_stack_top" ../../FreeRTOS-Kernel/portable/` which states in `../../FreeRTOS-Kernel/portable/GCC/RISC-V/port.` that "The stack used by interrupt service routines. Set configISR_STACK_SIZE_WORDS to use a statically allocated array as the interrupt stack. Alternative leave configISR_STACK_SIZE_WORDS undefined and update the linker script so that a linker variable names __freertos_irq_stack_top has the same value as the top of the stack used by main..." so we add in src/FreeRTOSConfig.h the statement
   `#define configISR_STACK_SIZE_WORDS (500)`

5. Two constant definitions had to be added in `src/FreeRTOSConfig.h` specifically for meeting the RISC-V FreeRTOS source code requirements [2]: how can you modify this header file to only include the constants if compiling for a RISC-V target and not when targeting other architectures?

   define with `#ifdef` conditions, either using a custom `-D` option in the Makefile or using a standard constant defined by gcc targeting the RISC-V

6. The proposed example is too simple to execute in `qemu`, but a complete, functional example is found at `https://github.com/FreeRTOS/FreeRTOS/tree/main/FreeRTOS/Demo/RISC-V-Qemu-virt_GCC`. Compile this example and execute in `qemu` targeting the RISC-V architecture [3] as found in `/home/jmfriedt/qemu/build`. What is the result of executing this code?

[1] `https://gitlab.com/ohwr/project/wrpc-sw/-/tree/wrpc-v5-lm32/arch`
[2] `https://www.freertos.org/Using-FreeRTOS-on-RISC-V`
[3] `https://www.freertos.org/Documentation/02-Kernel/03-Supported-devices/04-Demos/0thers/RTOS-RISC-V-FreedomStudio-QMEU`

In Readme.md we are told to use the command "qemu-system-riscv32 -nographic -machine virt -net none -chardev stdio,id=con,mux=on -serial chardev:con -mon chardev=con,mode=readline -bios none -smp 4 -s –kernel output/RTOSDemo.elf" to run the simulation which displays: "FreeRTOS Demo Start"

7. What is the memory map used in this FreeRTOS demonstration? How much RAM and ROM are allocated to this emulated microcontroller?

   The compilation ends with -T fake_rom.ld which states that rom (rxa) : ORIGIN = 0x80000000, LENGTH = 512K and ram (wxa) : ORIGIN = 0x80080000, LENGTH = 1024K

8. How much memory has this Demo application allocated for the stacks of the FreeRTOS tasks? Two possible answers (or both) can be accepted. How does it compare with the hardware configuration of the target microcontroller?

   The task stacks are defined on the FreeRTOS heap with FreeRTOSConfig.h defining
   `#define configTOTAL_HEAP_SIZE ((size_t)(220*1024))` or
   `#define configTOTAL_HEAP_SIZE ((size_t)(80*1024))` so the memory allocation is much less than the amount provided in the linker script.

9. Add one more task that increments a variable from 0 to 50000 every millisecond. How can you control remotely the simulation to observe the state of this variable as the program is running in the emulator? Provide the tools, server and client, and their usage to achieve this result.

   running qemu with the -s option launches a gdb server and -S stops the execution unless a gdb client is connected and launches the execution with the run or continue commands. The GDB client is available in the same directory than the compiler

10. Print the value of the stack pointer as the program is being executed on the emulator using the same tools as identified in the previous question. How would you achieve this result on real hardware?

    GDB allows for print $sp. I am not aware of any technique for displaying the stack pointer on real hardware, but FreeRTOS provides the means to display the remaining amount of stack or stack corruption if the watermark is overwritten by an erroneous memory access.