FreeRTOS port on Renesas RISC-V MCU

Developed using R9A02G021 board

Riccardo Polelli*

November 2024

1 Introduction

In this document I will present the problems, and the relative resolutions, met during the port of FreeRTOS on an unsupported RISC-V MCU implementation by Renesas. I will also give some information on how to setup a development environment and what to keep in mind while developing an application with this MCU.

1.1 FreeRTOS port basics

Usually a port consists of implementing the files `port.c` and `portmacro.h`. They are specific for a certain compiler and architecture. In the case of a RISC-V port, there is already an official support for GCC and IAR.

Since the RISC-V specification is *wide*, the two cited files above are not enough. There is also the need of `portASM.S`, `portContext.h` and one of the `freertos_risc_v_chip_specific_extensions.h` to address all the peculiarities of a specific implementation of this architecture, such as register extensions, presence of Machine TIMEr and/or Core Local INTerruptor.

The port I've been working on uses the LLVM toolchain. However my starting point has been the existing GCC port.

Trying to be as much faithful as possible to the original documentation, the project directory structure will be the following:

```
Demo
    blink
        main.c
    Common
        common.c
        common.h
    FreeRTOSConfig.h
(smc_gen)
    ...
Source
    include
        ...
    Portable
        LLVM
            freertos_risc_v_chip_specific_extensions.h
            chip_extensions.cmake
            port.c
            portASM.S
            portContext.h
            portmacro.h
        MemMang
            heap_*.c
    list.c
    queue.c
```

*Contact: riccardo.polelli@mail.polimi.it

```
    tasks.c
    ...
linker_script.ld
```

1.2 Development Environment

1.2.1 With E2 Studio IDE

From version 2024-07 Linux host is supported for working with RISC-V MCUs. Renesas states that only Ubuntu version 22.04 is officially supported. But in my tests, I could not manage to flash the board from Ubuntu. My suggestion is to keep an eye on Renesas updates since their are releasing them very frequently and in the next ones they might address this problem.

The fastest way to setup the development environment is to install the IDE on Windows.

1.2.2 Without E2 Studio IDE

The IDE under the hood uses a LLVM toolchain. You can download it from this link: `https://llvm-gcc-renesas.com/riscv/riscv-download-toolchains/`. If you do not want to use the IDE, I suggest to look at the commands that the IDE uses to compile and create your own makefile.

Renesas provides a program, called *Smart Configurator* that allows to configure various aspects of the board (like pins, ADC, interrupts and so on). This can be easily accessed from the IDE or installed as a separate program.

1.2.3 Misc

There are many useful tools that can be used to analyze the `.elf`. Like `llvm-objdump -h file.elf` or `llvm-readobj --sections file.elf`. They are installed with the toolchain and they can be very useful while developing an application.

2 Challenges

Let me sum up the main challenges and problems that I've encountered:

- Non standard implementation of the `mcause` register
- Incorrect setup of the exception handler
- Setting up the interrupt vector table
- Memory misalignment in linker script
- Dealing with 0x3000 bytes of memory
- Dedicated interrupt stack or not

3 Problem-Solving Process

In this section I will present you in more detail the process of solving the problems cited in the previous section.

3.1 `mcause`

While debugging one thing that was not right was a comparison during the exception handling. I am referring to this part of the `portASM.s` file:

```
freertos_risc_v_trap_handler:
    portcontextSAVE_CONTEXT_INTERNAL

    csrr a0, mcause
    /* a0 now contains mcause. */
    /* a0 = 11 ==> environment call. */
    csrr a1, mepc
```

```
    bge a0, x0, synchronous_exception

...

synchronous_exception:
     /* Synchronous so update exception return
     address to the instruction after the instruction
     that generated the exeption. */
    addi a1, a1, 4
    /* Save updated exception return address. */
    store_x a1, 0( sp )
    /* Switch to ISR stack. */
    load_x sp, xISRStackTop
    j handle_exception

...

handle_exception:
    /* a0 now contains mcause. */
    /* a0 = 11 ==> environment call. */
    li t0, 11
    /* vvv HERE vvv */
    bne a0, t0, application_exception_handler
    /* ^^^ HERE ^^^ */
    call vTaskSwitchContext
    j processed_source
```

This exception handler is called whenever there is an environment call (as written in the comment). It is referring to the `ecall` instruction. When this instruction is executed (again as stated in the comment) we can recognize such situation during the handling of the exception thanks to the value in the `a0` register. This value should be 11 (0xb) for an `ecall`. While debugging I noticed that the register did not contain this value. The 16 less significant bits contained the value 0xb, but most significant bits contained more information. Because of this, the comparison was not right causing a not correct execution.

Since we do not care about those additional information, the handler now masks the value of a0 to only the last significant bits. This results in a correct boolean computation of the value for the comparison.

More practically the solution was achieved by adding the two lines:

```
handle_exception:
/* Lower 16 bits of a0 contains exception code. */
li t0, 0xffff       /* !!! HERE !!! */
and a0, a0, t0      /* !!! HERE !!! */
li t0, 11
bne a0, t0, application_exception_handler
call vTaskSwitchContext
j processed_source
```

3.2 Exception Handler Should Be Naked

The next problem regards this section of the auto generated code from Renesas for the exception handling:

```
#define EXVECT_SECT                     \
    __attribute__ ((section (".nvect")))
const void * gp_ExceptVectors[] EXVECT_SECT = {
    nvect_function
};

void nvect_function(void)
{
```

```
    /* Check the contents of mcause and
    implement a process to branch to the
    appropriate process
    after determining what happened. */
};
```

Everything seems right. Only the jump to the routine to handle the exception should be coded, but unfortunately there is more.

If you need to install a function for interrupt or exception handling, then the function must be declared as naked so that it does not include any epilogue or prologue.

To achieve this the interrupt vector table (that includes also as first entry the one for the exceptions) is often written in assembly with many jumps so that the epilogue an prologue are not created. Something like this:

```
interrupt_vector_table:
    j exception_handler
    j interrupt_handler
    j interrupt handler
    ...
```

Once this is done, with a specific instruction to write the control and status register (`csrw`) you can install this interrupt vector table.

During my tests writing in the control and status register with `csrw` had some wanted effects and some unwanted ones. So it was much better and easier to work with what Renesas provides and make it work as I wanted.

Renesas achieves the same functionality by jumping to memory areas called `nvect` (for exceptions) and `vect` (for interrupts) that contain the handlers. The problem is that those handlers are coded as C functions, that by nature have an epilogue and prologue. By doing so we need to manually specify that the function must be naked.

The fix in the end required to add the already cited `naked` compiler attribute and add the jump to the correct handler.

Compiling and running this has not the wanted effect, since the compiler optimizes it by completely removing this part of code (since it is only implicitly called after the `ecall`). To ask the compiler to not delete it, also the `used` attribute is needed.

```
#define EXVECT_SECT                         \
    __attribute__ ((section (".nvect"))) \
    __attribute__((naked))                \
    __attribute__((used))
void nvect_function(void) EXVECT_SECT;
void nvect_function(void)
{
    asm( "j freertos_renesas_risc_v_trap_handler" );
};
```

3.3 Setting up Interrupt Vector Table

As I already said, my aim is to use as much as possible the code from Renesas. Their original interrupt vector table contained function pointers that the user needed to code. This time the implementation was much easier since the only thing left to was changing the function pointers to the labels of the routines in FreeRTOS to handle the interrupts.

As you can see this is where the developer can configure the external interrupts from buttons, pins and so on.

```
#define VECT_SECT          __attribute__ ((section (".vects")))
const void * gp_Vectors[] VECT_SECT = {
/*
 * Reserved (0x00)
 */
  freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x04)
 */
```

```
  freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x08)
 */
  freertos_risc_v_interrupt_handler,

/*
 * INT_ACLINT_MSIP (0x0C)
 */
  freertos_risc_v_mtimer_interrupt_handler,

/*
 * Reserved (0x10)
 */
  freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x14)
 */
  freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x18)
 */
  freertos_risc_v_interrupt_handler,

/*
 * INT_ACLINT_MTIP (0x1C)
 */
 freertos_risc_v_mtimer_interrupt_handler,

/*
 * Reserved (0x20)
 */
 freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x24)
 */
 freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x28)
 */
 freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x2C)
 */
 freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x30)
 */
 freertos_risc_v_interrupt_handler,
```

```
/*
 * Reserved (0x34)
 */
 freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x38)
 */
 freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x3C)
 */
 freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x40)
 */
 freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x44)
 */
 freertos_risc_v_interrupt_handler,

/*
 * Reserved (0x48)
 */
 freertos_risc_v_interrupt_handler,

/*
 * INT_IELSR0 (0x4C)
 */
 freertos_risc_v_interrupt_handler,

/*
 * INT_IELSR1 (0x50)
 */
 freertos_risc_v_interrupt_handler,

/*
 * INT_IELSR2 (0x54)
 */
 freertos_risc_v_interrupt_handler,

/*
 * INT_IELSR3 (0x58)
 */
 freertos_risc_v_interrupt_handler,

/*
 * INT_IELSR4 (0x5C)
 */
 //freertos_risc_v_interrupt_handler,
 r_Config_ICU_irq4_interrupt,

/*
```

```
 * INT_IELSR5 (0x60)
 */
 freertos_risc_v_interrupt_handler,
...

/*
 * INT_IELSR29 (0xC0)
 */
 freertos_risc_v_interrupt_handler,

/*
 * INT_IELSR30 (0xC4)
 */
 freertos_risc_v_interrupt_handler,

/*
 * INT_IELSR31 (0xC8)
 */
 freertos_risc_v_interrupt_handler,
};
```

I took the freedom to heavily change the approach to interrupt handling since this file already needed to be changed because of the problem described in the previous section (note this is the **only** Renesas auto generated modified file). If you don't like to bypass the Renesas interrupt handling, it is also possible to integrate it more into Renesas code and avoid such heavy modification, since they provide some place where to write user code for interrupt handling. But this solution is much simpler, has the same effect, is cleaner and much easier to understand.

3.4 Misalignment in Linker Script

While linking I noticed how the linker was warning me about a misalignment in the `text` section. Once identified the error, the fix was easy. I modified the auto generated linker script by Renesas this way:

```
/*.text (ALIGN(. + __romdatacopysize +
    __romdataeccramcopysize, 4)): /* old */
.text (ALIGN(. + __romdatacopysize +
    __romdataeccramcopysize, 256)): /* new */
{
    *(.text)
    *(.text.*)
    /*INPUT_SECTION_FLAGS(SHF_EXECINSTR) *(*_n)*/
} >ROM AT>ROM
```

3.5 0x3000 Bytes of RAM

The RAM is of size 0x3000. This is not a lot. I managed to get up to 7 processes running but using for each of them a really small stack. As a consequence the heap must be configured to be pretty small too. This is something to keep in mind while writing the application.

3.6 Non Dedicated Interrupt Stack

The official port on RISC-V allows to use a dedicated interrupt stack. My tests have been conducted without using this feature (officially it is stated to be supported only with GCC). I am sure that adding this feature is not a difficult task. It would imply to declare the top of the dedicated interrupt stack in the linker script. Because of the small RAM available, using a part of memory just for the interrupts seemed not reasonable to me. However I tested this feature and it works, but I would not suggest to use it in a real world application.

4 Misc

4.1 Hardware breakpoints

Since hardware breakpoints are like programmable comparators that read from the program address bus, there can be only a limited number of them. In this case there is **maximum of 4** while debugging a project. Pay attention to the fact that sometimes the IDE automatically adds breakpoints.

Adding breakpoints at runtime works.

4.2 Renesas Updates

The updates from Renesas in this month have been many. Renesas published some update of the IDE and other components orbiting around the IDE. With those also some *tips* and updates on how to make things work on the board.

I strongly suggest to keep an eye on those updates and read all of them since you may find useful information even in documents you did not expect to.

4.3 Variable Clock Frequency

This board supports different clock speeds. Now the 48000000 (Hz) value is hard coded inside the FreeRTOS configuration file (you can find it in the Demo folder). If you change the clock value of the board, remember to also change the value there. As far as I know FreeRTOS does not support changing clock frequency at runtime (at least not officially), but this can be easily implemented at will. If so, you will need to call `get_iclk_freq_hz` to gather the running clock frequency.

Note that after a reset the clock frequency might change.

However I suggest to be as close as possible to the official version of FreeRTOS and not change the clock frequency at runtime.

5 Conclusion

During this project I have learned a lot. I discovered for the first time FreeRTOS and digged deeper into the internals of a RISC-V implementations. It took a lot of time because the bugs and problems that I described are hard to find and solve. They do also require a lot of study of the components involved and a lot of unsuccessful tries. In the end, I feel really satisfied with the outcome of the project and I hope it can be helpful for someone who wants to use this MCU for a real world application.

For more updates, keep an eye on the *readme* of the repository:
`https://github.com/Pole11/FreeRTOS-port-renesas-R9A02G021`.