Latest News

NXP tweet showing LPC5500 (ARMv8-M Cortex-M33) running FreeRTOS.

Meet Richard Barry and learn about running FreeRTOS on RISC-V at FOSDEM 2019

Version 10.1.1 of the FreeRTOS kernel is available for immediate download. MIT licensed.

View a recording of the "OTA Update Security and Reliability" webinar, presented by TI and AWS.

Careers

FreeRTOS and other embedded software careers at AWS.

FreeRTOS Partners

Home
MIT License
FreeRTOS Books and Manuals
⊞ FreeRTOS
⊞ FreeRTOS Interactive!

Quick Start Guide
Support Forum
⇓ Download Source ⇓

FreeRTOS+ Ecosystem

FreeRTOS+TCP:
Thread safe TCP/IP stack
SafeRTOS:
TUV certified RTOS
OpenRTOS:
Commercial Licensed RTOS
Fail Safe File System:
Ensures data integrity
FreeRTOS BSPs:
3rd party driver packages
Trace & Visualisation:
Tracealyzer for FreeRTOS
CLI:
Command line interface
WolfSSL SSL / TLS:
Networking security protocols
RTOS Training:
Delivered online or on-site
IO:
read(), write(), ioctl() interface

FreeRTOS+ Lab Projects

FreeRTOS+POSIX:
POSIX threading API
FreeRTOS+FAT:
Thread aware file system

Hint: Use the tree menu to navigate groups of related pages

Using FreeRTOS on RISC-V Microcontrollers

Preamble

The FreeRTOS RISC-V port has not been formally released yet, and as such, the port's details, and therefore the information on this page, are subject to change.

Introduction

The RISC-V instruction set architecture (ISA) is easily extensible and does not specify everything about physical RISC-V microcontroller or system on chip (SoC) implementations. Accordingly, the FreeRTOS RISC-V port is also extensible - it provides a base port that handles the registers common to all RISC-V implementations, and a set of macros that must be implemented to handle hardware implementation specific features and extensions, such as additional registers.

Quick start

The main body of this page provide detailed information on building FreeRTOS for RISC-V cores, but the simplest way to get started is to use one of the pre-configured example projects. The example RISC-V projects are located in the sub-directories of `FreeRTOS/Demo` that start "RISC-V" in the main FreeRTOS zip file download. These projects can be used directly, or simply as a worked example and reference for the source files, configuration options, and compiler settings detailed below.

In summary, to build FreeRTOS for a RISC-V core you need to:

1. Include the core FreeRTOS source files and the FreeRTOS RISC-V port layer source files in your project.
2. Ensure the assembler's include path includes the path to the header file that describes any chip specific implementation details.
3. Define either a constant in FreeRTOSConfig.h or a linker variable to specify the memory to use as the interrupt stack.
4. Define configCLINT_BASE_ADDRESS in FreeRTOSConfig.h.
5. For the assembler, #define portasmHANDLE_INTERRUPT to the name of the function provided by your chip or tools vendor for handling external interrupts.
6. Install the FreeRTOS trap handler.

Other links that may be helpful include:

- FreeRTOS kernel quick start guide
- Adapting a FreeRTOS demo to different hardware
- Creating a new FreeRTOS project

Detailed information

On this page:

Features of the FreeRTOS RISC-V port

The FreeRTOS RISC-V port:

- Supports machine mode integer execution on 32-bit RISC-V cores only, but is under active development, and future FreeRTOS releases will add features and functionality as required by our users.
- Implements a separate interrupt stack, and in so doing, greatly reduces RAM usage on small microcontrollers by removing the need for every task to have a stack large enough for both interrupt and non-interrupt stack frames.
- Provides a base port that can be easily extended to accommodate RISC-V implementation specific architecture extensions.

Source files

The FreeRTOS kernel source code organization page contains information on adding the FreeRTOS kernel to your project. In addition to the information on that page, the FreeRTOS RISC-V port requires one additional header file. The additional header file describes chip specific details, and is required because RISC-V chips often include chip specific architecture extensions.

The additional header file is called `freertos_risc_v_chip_specific_extensions.h`. There is one implementation of this header file for each supported architecture extension, with all implementations located in subdirectories of the `/FreeRTOS/Source/Portable/[compiler]/RISC-V-RV32/chip_specific_extensions` directory.

To include the correct `freertos_risc_v_chip_specific_extensions.h` header file for your chip simply add the path to that header file to the assembler's include path (note this is the **assembler's** include path, not the compiler's include path). For example:

- If your chip implements the base RV32I architecture without extensions, and includes a Core Local Interrupter (CLINT), then add `/FreeRTOS/Source/Portable/[compiler]/RISC-V-RV32/chip_specific_extensions/`**`RV32I_CLINT_no_extensions`** to the assembler's include path.
- If your chip uses a PULP RI5KY core as implemented on the RV32M1RM Vega board, which includes six additional registers and does not include a a Core Local Interrupter (CLINT), then add `/FreeRTOS/Source/Portable/[compiler]/RISC-V-RV32/chip_specific_extensions/`**`Pulpino_Vega_RV32M1RM`** to the assembler's include path.

Also see the compiler and assembler command line options section below for information on setting assembler command line options, and the porting FreeRTOS to new RISC-V implementations section for information on creating your own `freertos_risc_v_chip_specific_extensions.h` header files.

FreeRTOSConfig.h settings

`configCLINT_BASE_ADDRESS` must be defined in `FreeRTOSConfig.h`. If the target RISC-V chip includes a Core Local Interrupter (CLINT) then set `configCLINT_BASE_ADDRESS` to the CLINT's base address. Otherwise set `configCLINT_BASE_ADDRESS` to 0.

For example, if the CLINT's base address is 0x20040000, then add the following line to `FreeRTOSConfig.h`:

```
#define configCLINT_BASE_ADDRESS ( 0x20040000 )
```

If there is no CLINT, then add the following line to `FreeRTOSConfig.h`:

```
#define configCLINT_BASE_ADDRESS ( 0 )
```

Interrupt (system) stack setup

The FreeRTOS RISC-V port switches to a dedicated interrupt (or system) stack before any C functions are called from an interrupt service routine (ISR).

The memory to use as the interrupt stack can either be defined in the linker script or declared within the FreeRTOS port layer as a statically allocated array. The linker script method is preferred on memory constrained MCUs as it allows the stack that was used by main() prior to the scheduler being started (which is no longer used for that purpose after the scheduler has been started) to be re-purposed as the interrupt stack.

- To use a statically allocated array as the interrupt stack:

  Define configISR_STACK_SIZE_WORDS in `FreeRTOSConfig.h` to the size of the interrupt stack to be allocated. Note the size is defined in words, not bytes.

  For example, to use a 500 word (2000 bytes on an RV32, where each word is 4 bytes) statically allocated interrupt stack add the following to `FreeRTOSConfig.h`:

  ```
  #define configISR_STACK_SIZE_WORDS ( 500 )
  ```

- To defined the interrupt stack in the linker script:

  1. Declare a linker variable called `__freertos_irq_stack_top` that holds the highest address of the interrupt stack, and
  2. Ensure configISR_STACK_SIZE_WORDS is **not** defined.

  Using this method requires editing the linker script. If you are not familiar with linker scripts then it is important to know, when using GCC at least, that '**.**' is what is known as the location counter, and holds the value of the memory address at that point in the linker script. There is no need to understand the fine details of linker scripts though, just copy the example below.

  The stack that was used by main() before the scheduler is started is no longer required after the scheduler is started, so ideally reuse that stack by setting `__freertos_irq_stack_top` to equal the value of the highest address of the stack allocated for use by main(). For example, if your linker script contains something like the below (the actual linker scripts in use will vary):

  ```
  .stack : ALIGN(0x10)
  {
    __stack_bottom = .;
    . += STACK_SIZE;
    __stack_top = .;
  } > ram
  ```

  Then `__stack_top` (example name only) is a linker variable, the value of which equals the highest address of the stack used by main() (recall '.' holds the value of the memory address at any given place in the linker script). In this case, to give `__freertos_irq_stack_top` the same value as `__stack_top`, just define `__freertos_irq_stack_top` immediately after `__stack_top`. See the example below:

  ```
  .stack : ALIGN(0x10)
  {
    __stack_bottom = .;
    . += STACK_SIZE;
    __stack_top = .;
    __freertos_irq_stack_top= .; /* ADDED THIS LINE. */
  } > ram
  ```

Note: at the time of writing, unlike with task stacks, the kernel does not check for overflows in the interrupt stack.

Required compiler and assembler command line options

Different RISC-V implementations provide different handlers for external interrupts, so it is necessary to tell the FreeRTOS kernel which external interrupt handler to call. To set the name of the external interrupt handler:

1. Locate the name of the external interrupt handler provided by your RISC-V run-time software distribution - this is normally the software provided by the chip vendor. The interrupt handler **must** have one parameter, which is the value of the RISC-V *cause* register at the time the interrupt was entered. For example, the prototype of the interrupt handler is expected to be (sample name only, use the correct name for your software):

   ```
   void external_interrupt_handler( uint32_t cause );
   ```

2. Define an assembler macro (note this is an **assembler** macro, not a compiler macro) called `portasmHANDLE_INTERRUPT` to equal the name of the interrupt handler.

   If using GCC then this can be achieved by adding the following to the assembler's command line, assuming the interrupt handler is called `external_interrupt_handler`:

   ```
   -DportasmHANDLE_INTERRUPT=external_interrupt_handler
   ```

It is also necessary to add the path to the correct `freertos_risc_v_chip_specific_extensions.h` header file for the RISC-V chip in use to the assembler's include path (note this is the **assembler's** include path, not the compiler's include path). See the source files section above.

Installing the FreeRTOS trap handler

The FreeRTOS trap handler is called `freertos_risc_v_trap_handler()` and is the central entry point for all interrupts and exceptions. The FreeRTOS trap handler calls the external interrupt handler when the source of a trap is an external interrupt.

To install the trap handler:

1. If the RISC-V core in use includes a Core Local Interrupter (CLINT) then freertos_risc_v_trap_handler() is installed automatically, and no specific actions are required.

2. If the RISC-V core in use does not include a CLINT then it is necessary to install freertos_risc_v_trap_handler() manually. That can be done by editing the startup code provided by your chip provider.

Note: If the RISC-V chip uses a vectored interrupt controller then install `freertos_risc_v_trap_handler()` as the handler for each vector.

Porting to new 32-bit RISC-V implementations

Read the FreeRTOS RISC-V source files section above before reading this section.

The `freertos_risc_v_chip_specific_extensions.h` file contains the following macros that must be defined:

- `portasmHAS_CLINT`

  If the target RISC-V chip includes a Core Local Interrupter (CLINT) then set #define `portasmHAS_CLINT` to 1, otherwise #define `portasmHAS_CLINT` to 0.

- `portasmADDITIONAL_CONTEXT_SIZE`

  The RISC-V Instruction Set Architecture (ISA) is extensible, so RISC-V chips may include additional registers over and above those required by the base architecture specification.

#define `portasmADDITIONAL_CONTEXT_SIZE` to the number of additional registers that exist on the target chip - which might be zero. For example, the RI5CY core on the Vega board includes six additional registers, so the `freertos_risc_v_chip_specific_extensions.h` provided for use with that chip includes the following line:

```
#define portasmADDITIONAL_CONTEXT_SIZE 6
```

- `portasmSAVE_ADDITIONAL_REGISTERS`

  `portasmSAVE_ADDITIONAL_REGISTERS` is an **assembly** macro (not a #define) that must be implemented to save any chip specific additional registers.

  If there are no chip specific extension registers (`portasmADDITIONAL_CONTEXT_SIZE` is set to zero) then portasmSAVE_ADDITIONAL_REGISTERS must be an empty assembly macro as follows:

  ```
  .macro portasmSAVE_ADDITIONAL_REGISTERS
      /* No additional registers to save, so this macro does nothing. */
      .endm
  ```

  If there are chip specific extension registers (`portasmADDITIONAL_CONTEXT_SIZE` is greater than zero) then portasmSAVE_ADDITIONAL_REGISTERS must:

  1. Decrement the stack pointer to create enough stack space for the additional registers, then...
  2. Save the additional registers into the created stack space.

  For example, if the chip has three additional registers then `portasmSAVE_ADDITIONAL_REGISTERS` must be implemented as follows (where the names of the registers will be dependent on the chip, and not as shown here):

  ```
  .macro portasmSAVE_ADDITIONAL_REGISTERS
      /* Use the portasmADDITIONAL_CONTEXT_SIZE and portWORD_SIZE
      macros to calculate how much additional stack space is needed,
      and subtract that from the stack pointer.  This line can just
      be copied from here provided portasmADDITIONAL_CONTEXT_SIZE
      is set correctly.  Note the minus sign ('-').  portWORD_SIZE
      is already defined elsewhere. */
      addi sp, sp, -(portasmADDITIONAL_CONTEXT_SIZE * portWORD_SIZE)

      /* Next save the additional registers, which here are assumed
      to be called xx0 to xx2, but will be called something different
      on your chip, to the stack.  Assumes portasmADDITIONAL_CONTEXT_SIZE
      is 3. */
      sw xx0, 1 * portWORD_SIZE( sp )
      sw xx1, 2 * portWORD_SIZE( sp )
      sw xx2, 3 * portWORD_SIZE( sp )
      .endm
  ```

- `portasmRESTORE_ADDITIONAL_REGISTERS`

  `portasmRESTORE_ADDITIONAL_REGISTERS` is the reverse of `portasmSAVE_ADDITIONAL_REGISTERS`.

  If there are no chip specific extension registers (`portasmADDITIONAL_CONTEXT_SIZE` is set to zero) then `portasmRESTORE_ADDITIONAL_REGISTERS` must be an empty assembly macro as follows:

  ```
  .macro portasmRESTORE_ADDITIONAL_REGISTERS
      /* No additional registers to restore, so this macro does nothing. */
      .endm
  ```

  If there are chip specific extension registers (`portasmADDITIONAL_CONTEXT_SIZE` is greater than zero) then `portasmRESTORE_ADDITIONAL_REGISTERS` must:

  1. Read the additional registers from the stack locations used by `portasmSAVE_ADDITIONAL_REGISTERS`, then...
  2. Remove the stack space used to hold the additional registers by incrementing the stack pointer by the correct amount.

  For example, if the chip has three additional registers then `portasmRESTORE_ADDITIONAL_REGISTERS` must be implemented as follows (where the names of the registers will be dependent on the chip, not as shown here):

  ```
  .macro portasmRESTORE_ADDITIONAL_REGISTERS
      /* Restore the additional registers, which here are assumed
  ```

```
to be called xx0 to xx2, but will be called something different
on your chip from the stack.  Assumes
portasmADDITIONAL_CONTEXT_SIZE is 3. */
lw xx0, 1 * portWORD_SIZE( sp )
lw xx1, 2 * portWORD_SIZE( sp )
lw xx2, 3 * portWORD_SIZE( sp )

/* Use the portasmADDITIONAL_CONTEXT_SIZE and portWORD_SIZE
macros to calculate how much space to remove from the stack.
This line can just be copied from here provided
portasmADDITIONAL_CONTEXT_SIZE is set correctly.  portWORD_SIZE
is already defined elsewhere. */
addi sp, sp, (portasmADDITIONAL_CONTEXT_SIZE * portWORD_SIZE)
.endm
```