Exploration of FreeRTOS on a RISC-V Architecture

Vincent Abraham[1[0000-0003-3336-9595]], Divyesh Ranpariya[1[0000-0002-5419-2728]], Parin Parikh[1[0000-0002-2263-8086]], Sachin Gajjar[1 [0000-0002-7053-5970]], Dhaval Shah[1(✉)[0000-0002-2531-9590]]

[1] Department of Electronics and Communication Engineering
Institute of Technology, Nirma University, Ahmedabad, India
vincent.ahm@gmail.com, divyeshranpariya23@gmail.com,
parin.parikh10@gmail.com, sachin.gajjar@nirmauni.ac.in
dhaval.shah@nirmauni.ac.in,

Abstract. The RISC-V ISA has a lot of potential as a future adaptable ISA for general-purpose computing. However, RISC-V's adoption is hampered by a lack of software support, especially among major operating systems. RISC-V support has only been added to a few operating systems, notably Linux and FreeBSD. Many other Embedded Operating Systems have started supporting the RISC-V architecture. RTOS such as Zephyr OS and FreeRTOS have board support for RISC-V, but these ports often face the issue of improper documentation and maintenance. This work is a step towards exploring the FreeRTOS port for SPIKE simulator and analyzing its performance on a 64-bit RISC-V core. As a part of this effort, a dive is performed into the steps involved in installing the SPIKE simulator by setting up the RISC-V environment as well as installing the FreeRTOS port to develop real-time applications. The FreeRTOS port is explored by first writing two programs which use some methodologies of a RTOS. Next, with the help of the system call APIs and library support of the compiler, performance measurement of FreeRTOS, on top of the simulated core is carried out. The performance parameters were task creation and deletion time, context switching time, mutex locking and unlocking time, and the boot time of an OS.

Keywords: RISC-V, Real-Time Operating System, FreeRTOS, SPIKE Simulator, Inter-Process Communication, Mutex, Performance Analysis

1 Introduction

In the current day, embedded systems have found their application in various domains, ranging from home and industrial automation, businesses, to fields such as communications, aeronautics, defense and entertainment [1]. Some of these applications are dependent not only on the logical correctness but also on the time-predictive behavior and the memory efficiency of the systems. Simpler software for soft or firm real-time system requirements can be achieved by writing bare-metal software that controls the underlying hardware. But it's difficult to maintain the software with higher complexity for such systems. Hence, a real-time operating system is used in time critical embedded systems to fulfil these requirements. The use of an RTOS often becomes necessary

when dealing with deadline, resource and priority constraints, as well as a higher number of tasks. Through features such as scheduling, resource management, task synchronization, input-output management etc., a RTOS helps the memory constrained system achieve an optimal real-time performance [2].

RISC-V is a RISC based open-source instruction set architecture which has been slowly creeping up in the mainstream computing market [2]. As it is open-source, industries could modify it and design their own extensions suitable to their needs. However, one limitation of the RISC-V ISA is the fact that it's fairly recent. It has only been adopted for a handful of commercial applications and has limited software support [3]. Furthermore, even though the majority of the RTOS now have a port for this architecture, their documentation is often times hard to find or suffers from many errors due to lack of proper updates and maintenance.

Discussing the papers reviewed on existing implementations of porting different available RTOS on top of RISC-V cores, in the work presented by [4], Singhal S.P. et al. discusses a port of eChronos on RISC-V architecture and results are observed by compiling and debugging a generic "Hello World" program using SPIKE. The underlying toolchain, as well as any essential dependencies, such as the QEMU emulator and the RISC-V proxy kernel, are discussed, together with complete installation and execution instructions. Furthermore, architectural relationships and subsequent changes in eChronos are investigated. In [5] Yahia Mazzi et. al. evaluated the performance of two RTOS, namely Keil RTX5 and FreeRTOS v10.2.0. Task switching time, pre-emption time, semaphore shuffling time, and inter-task messaging delay are some of the timing measures used for comparison purposes. FreeRTOS outperformed Keil RTX5 in all parameters except inter-task messaging delay. In [6] A. K. Vishwakarma et al. has implemented a port of the MicroC/OS-II RTOS to a PowerPC 7410-based Avionics platform. It also includes a detailed performance comparison of the MicroC/OS-II port with RT-Linux on the same hardware, and the results show that the MicroC/OS-II port outperforms RT-Linux in terms of all the performance parameters.

The available works have not discussed about porting of FreeRTOS on the RISCV architecture as per our best knowledge. This paper intends to explore FreeRTOS, a real-time operating system which is widely used in various embedded system applications. The processor on which FreeRTOS would be ported is a generic 64-bit RISC-V processor. This paper presents the exact approach for porting FreeRTOS on the SPIKE Simulator, in a Linux environment. Next, application layer programs are written to explore certain real-time embedded systems concepts such as inter-process communication (IPC) and mutexes. Subsequently, to compare the time taken for various operations such as ask creation/deletion, context switching, mutex lock/unlock and booting the kernel, a program is written. The results of these performance parameters would help developers to take an informed decision when selecting an operating system for their embedded application.

The organization of the rest of the paper is as follows: section 2 discusses the system diagram for the proposed implementation of porting FreeRTOS on a RISC-V core. Section 3 discusses the SPIKE Simulator, FreeRTOS and the steps involved for porting

FreeRTOS. Section 4 defines the performance analysis parameters and finally, section 5 discusses the performance analysis, based on these results. The paper would be concluded in section 6.

2 System Diagram

Fig. 1 shows the proposed diagram of the system. FreeRTOS would be ported on the 64-bit RISC-V general-purpose computing core RV64GC, simulated by SPIKE. The user would use operation specific system calls provided as APIs through the system call interface, which would be interpreted to the host with the help of the FreeRTOS kernel. The kernel would also act as an abstraction layer for the RTOS by handling various low-level tasks.

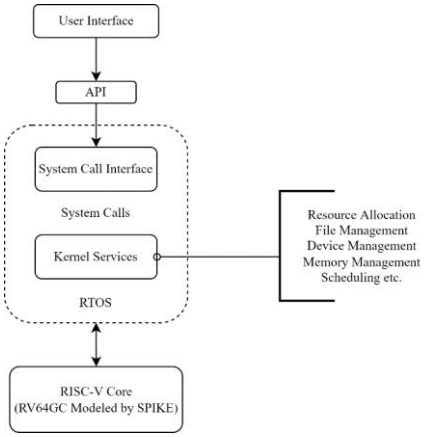

Fig. 1. Design for the proposed implementation

3 Overview of FreeRTOS and the SPIKE Simulator

This section discusses the software flow used for porting FreeRTOS on the SPIKE Simulator. The RISC-V GNU toolchain lies at the bottom of the hierarchy and is basically a C/C++ based utility, supporting the RV64GC [7] by default. RV32GC can also be accessed by setting up the console's corresponding environment variable. The RISC-V proxy kernel is an execution environment which helps to mimic the input-output related system calls on the host computer. The SPIKE simulator is used for the functional simulation of high-level as well as assembly language codes on the top of the RISC-V core [9]. These three software tools are completely open-source.

3.1 RISC-V GNU Toolchain

The RISC-V GNU compiler toolchain consists of a C and C++ cross-compiler which helps build the written source program to an executable and linkable format (ELF) which can run on the machine. The currently supported build modes include the Linux

cross-compiler and the Newlib cross-compiler [7]. The installation steps for RISC-V GNU Toolchain in a Linux environment [7] are shown in Table 1.

Table 1. RISC-V GNU toolchain installation steps

Step No.	Description	Command
1	Make a target installation directory. Here, the name of the installation directory is 'riscv'.	`mkdir riscv`
2	Clone the GitHub repository for the RISC-V GNU toolchain into a seperate directory (here, named 'GCC').	`git clone --recursive https://github.com/riscv/riscv-gnu-toolchain GCC`
3	Install the required dependencies	`sudo apt-get install autoconf automake autotools-dev curl python3 libmpc-dev libmpfr-dev libgmp-dev gawk build-essential bison flex texinfo gperf libtool patchutils bc zlib1g-dev libexpat-dev`
4	Move to the created directory ('GCC').	`cd GCC`
5	Make a new directory named 'build'	`mkdir build`
6	Move to the 'build' directory	`cd build`
7	Add the installation path to the environment variable. For example, if the the target directory has the path '/opt/riscv', the installation path would be '/opt/riscv/bin'	`export PATH="/opt/riscv/bin:$PATH"`
8	Build the cross-compiler using the make file	`../configure --prefix=/opt/riscv` `make`

3.2 RISC-V Proxy Kernel

The RISC-V proxy kernel provides a lightweight environment for the compilation and execution of RISC-V applications which require only limited input-output capabilities. With the help of the host target interface, the proxy kernel proxies the system calls to the host [8]. This package also includes the Berkley Boot Loader, which hosts the Linux port for RISC-V systems [9]. The steps for installing the proxy kernel in a Linux environment [8] are shown in Table 2.

Table 2. RISC-V proxy kernel installation steps

Step No.	Description	Command
1	Add the installation path to the environment variable as done in Step 7 of section 3.1.	
2	Clone the GitHub repository for the RISC-V proxy kernel and bootloader into a separate directory (here, named 'PK').	`git clone https://github.com/riscv-software-src/riscv-pk.git PK`
3	Move to the created directory ('PK').	`cd PK`
4	Make a new directory named 'build' and move to that directory.	`mkdir build` `cd build`
5	Build and install the proxy kernel	`../configure --prefix=/opt/riscv --host=riscv64-unknown-elf` `sudo make --> sudo make install`

3.3 SPIKE Simulator

SPIKE is a RISC-V ISA simulator that implements a functional model of RISC-V cores [10]. It supports RV32I and RV64I base ISAs along with various other features and

extensions. The steps for installing the SPIKE simulator in a Linux environment [10] are shown in Table 3.

Table 3. SPIKE simulator installation steps

Step No.	Description	Command
1	Add the installation path to the environment variable as done in Step 7 of section 3.1.	
2	Clone the GitHub repository containing the SPIKE simulator into a separate directory (here, named 'SPIKE').	`git clone https://github.com/riscv-software-src/riscv-isa-sim.git SPIKE`
3	Move to the created directory ('SPIKE')	`cd SPIKE`
4	Install the device-tree compiler	`apt-get install device-tree-compiler`
5	Make a new directory – 'build'	`mkdir build`
6	Move to the directory named 'build'.	`cd build`

7	Configure the 64-bit RV64GC toolchain path and compile the project using the make utility.	`../configure --prefix=/opt/riscv`
		`make`
		`[sudo] make install`

3.4 FreeRTOS

FreeRTOS is one of the most popular open-source real-time kernels, supporting approximately twenty different compilers and thirty-five different processor architectures [12]. It offers a relatively small code size, with its base core accumulating roughly 4000 to 9000 bytes [12]. This renders it to be suitable for many of the resource constrained microcontroller-based applications. Furthermore, as a majority of the source files are written in C, they are also easy to understand for the user. Each FreeRTOS port is contained within two directories, 'Source' and 'Demo'. The 'Source' directory contains the source and header files, whereas 'Demo' contains demo projects for the user to run. The 'Source' directory holds files which consist of function definitions for the various services, as well as two directories named 'include' and 'portable'. The 'include' directory contains header files for the respective source files. Whereas, 'portable' consists of directories for architecture specific files, as well as a directory named 'MemMang', which consists of examples of five heap allocation schemes. A brief description of the source files in a FreeRTOS port is given in Table 4.

Table 4. FreeRTOS source files

File Name	Description
queue.c/queue.h, semphr.h	Function definitions for queue, mutex and semaphore services.
timers.c/timers.h	Function definitions for software timers.
event_groups.c/event_groups.h	Function definitions for event group functionalities.
heap_x.c	Examples of five heap allocation schemes contained within heap_1.c heap_2.c, heap_3.c, heap_4.c, and heap_5.c.
list.c/list.h	List function definitions used by the task scheduler.
port.c/portmacro.h	Definitions of low-level functions involving hardware interrupts, context switching, tick timers, etc.
tasks.c/task.h	Function definitions relating to tasks and task utilities.
FreeRTOS.h	Configuration file which combines all the resources.
FreeRTOSConfig.h	Configuration file of the FreeRTOS port, system clock, and irq parameters.

The GitHub repository at [14] provides a port for the SPIKE simulator of FreeRTOS version 8.2.3. The 'Demo' directory contains a pre-configured demo program named main.c. To write a custom program, the necessary changes can be made to the the existing code or the new program file can be named as main.c. The steps to port the RTOS kernel onto the SPIKE simulator [14] are shown in Table 5.

Table 5. Porting FreeRTOS on SPIKE simulator

Step No	Description	Command
1	Add the install path to the environment. i.e. if the installation directory is /opt/riscv, execute the following command:	`export PATH="/opt/riscv/bin:$PATH"`
2	Clone the GitHub repository as a directory named 'FreeRTOS'.	`git clone https://github.com/-illustris/FreeRTOS-RISCV.git FreeRTOS`
3	Add the install path (opt/riscv) in line 61 of Makefile and line 5 of Makefile.inc. These files would be located at FreeRTOS/Demo/riscv-spike.	
4	Move to the directory containing the makefile in 'FreeRTOS'	`cd Demo/riscv-spike`
5	Compile the makefile to build the program	`make`
6	Run the program	`spike riscv-spike.elf`

4 Results and Discussion

This section would first discuss the methodology used for calculating the performance analysis parameters and obtained results observed for performance measurement.

4.1 Procedure for Measuring the Performance Parameters

In the case of FreeRTOS, the SPIKE simulator doesn't simulate a clock. However, an assembly function, named rdcycle is provided which returns the number of cycles executed, assuming a CPI of 1. Hence, a C function is created, which uses this instruction to calculate the performance parameters. Then, the time is calculated by taking a clock frequency of 1GHz, as most of the 64-bit RISC-V processors work at this frequency. Shown below is the function written for returning the number of cycles,

```
unsigned long read_cycles(void)
{
    unsigned long cycles;
 asm volatile ("rdcycle %0" : "=r" (cycles));
    return cycles;
}
```

4.2 Inter-Process Communication in FreeRTOS

IPC is a method that allows processes to interact and coordinate their operations. Here,

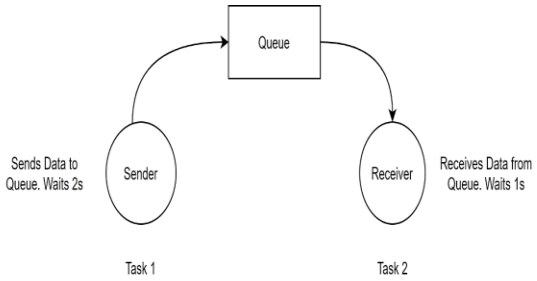

Fig. 2. Flow of inter-process communication performed

```
abraham@abraham-Inspiron-3558:~/Vincent/
e.elf

Sending 0 to receiver..

  Received 0.
Failed to receive data from queue.

Sending 1 to receiver..

  Received 1.
Failed to receive data from queue.

Sending 2 to receiver..

  Received 2.
Failed to receive data from queue.
```

Fig. 3. Results of inter-process communication

Two tasks, named as 'sender' and 'receiver', communicate with the help of a message queue. The queue is capable of holding 3 integer values and the sender task, after sending an integer, it waits for 2 seconds. The receiver task, after receiving a value from the queue waits for only 1 second. Hence, as the receiver receives the data at a faster rate, an asynchronization is found in the results. Fig. 3 gives a visual description of the program. The observed results are shown in Fig. 4.

4.3 Mutexes in FreeRTOS

Mutexes: Mutual exclusion objects which are used to allow task synchronization when accessing the same resources. It allows one thread in and blocks the other thread from accessing a critical resource. The program explores the use of a mutex when two tasks are writing some data into a single, shared array. An array called 'myResource' is created and both the tasks, 'vTask1' and 'vTask2'. Shown in Fig. 5 are the results observed when a mutex isn't used. It can be observed that the entire message isn't written properly due to task scheduling. When a mutex object (xMutex) is used each task locks the mutex, and the context switching doesn't take place until the mutex is released. Shown in Fig. 6 are the results observed when a mutex is used.

```
abraham@abraham-Inspiron-3558:~/Vince
e.elf
Task 2: Piolee
Task 1: Purple
Task 2: Purple
Task 2: Piolee
Task 1: Purple
Task 2: Purple
Task 2: Piolee
Task 1: Purple
Task 2: Purple
Task 2: Piolee
```

Fig. 4. Results when a mutex isn't used in implementation

```
abraham@abraham-Inspiron-3558:
e.elf
Task 1: Violet
Task 2: Purple
Task 1: Violet
Task 2: Purple
Task 1: Violet
Task 2: Purple
Task 1: Violet
Task 2: Purple
Task 1: Violet
Task 2: Purple
```

Fig. 5. Mutex implementation results

4.4 Performance Measurement of FreeRTOS

The performance measurement results are shown in Table 6. It is clear from the results that besides high context switching time, FreeRTOS fares well for the rest of the measurements. Hence, it can be stated that applications involving access to shared resources with mutexes would take a lesser time for locking and releasing of mutexes if using FreeRTOS. FreeRTOS also has a lower boot time because of its small code size.

Table 6. Simulation Results of the Performance Parameters

Parameters	FreeRTOS		Parameters	FreeRTOS
Task Creation	11.4 µs		Mutex Lock	0.162 µs
Task Deletion	0.134 µs		Mutex Unlock	0.188 µs
Context Switching	172 µs		Boot Time	108 µs

5 Conclusion

It can be concluded that RISC-V ISA support can provide better opportunities for an open-source development environment. Aided by proper documentation and understanding of the OS along with toolchain setup, it's easier to build an application on the top of RISC-V that contains any real-time OS running and managing the resources of the underlying hardware. This paper provides thorough documentation on implementing an embedded ecosystem using the FreeRTOS port on a RISC-V architecture. The SPIKE simulator provides an easy-to-use interface and an interactive debug mode to simulate a RISC-V core. This paper provides documentation on installing the SPIKE simulator and using the FreeRTOS port to run application layer programs. These steps are later used to run programs that involve an application of inter-process communication and mutexes. Using some fundamental metrics, the performance measurement of FreeRTOS on a RISC-V core was performed, which helps in understanding the timing requirements of FreeRTOS on SPIKE. Wherein it took a relatively higher time for context switching than the rest of the parameters. For future work, the emulation work can be carried out on a specific target board with the help of the required source files and board support packages.

References

1. Hambarde P, Varma R, Jha S (2014) The Survey of Real Time Operating System: RTOS. 2014 International Conference on Electronic Systems, Signal Processing and Computing Technologies. doi: 10.1109/icesc.2014.15
2. Stankovic J, Rajkumar R (2004) Real-Time Operating Systems. Real-Time Systems 28:237-253. doi: 10.1023/b:time.0000045319.20260.73
3. (2022) OpenBSD Journal: A resource for the OpenBSD community. In: Undeadly.org. http://undeadly.org/cgi?action%3Darticle%26sid%3D20070915195203%26mode%3Dexpanded. Accessed 5 Apr 2022.
4. Singhal S, Sridevi M, Sathya Narayanan N, Shankar Raman M (2020) Porting of eChronos RTOS on RISC-V Architecture. Lecture Notes in Electrical Engineering 1269-1279. doi: 10.1007/978-981-15-5341-7_96
5. Mazzi Y, Gaga A, Errahimi F (2021) Benchmarking and Comparison of Two Open-source RTOSs for Embedded Systems Based on ARM Cortex-M4 MCU. Indian Journal of Science and Technology 14:1261-1273. doi: 10.17485/ijst/v14i16.387
6. Vishwakarma A, Suresh K, Singh U (2014) Porting and systematic testing of an embedded RTOS. International Conference on Computing and Communication Technologies. doi: 10.1109/iccct2.2014.7066752
7. (2022) GitHub - riscv-collab/riscv-gnu-toolchain: GNU toolchain for RISC-V, including GCC. In https://github.com/riscv-collab/riscv-gnu-toolchain. Accessed 5 Apr 2022.
8. (2022) Running simulations · lowRISC: Collaborative open silicon engineering. In: Lowrisc.org.https://lowrisc.org/docs/tagged-memory-v0.1/simulations/#:~:text=Newlib%2FProxy%20Kenel%20(PK)&text=The%20proxy%20kernel%20simply%20proxies,with%20the%20PK%20and%20newlib. Accessed 5 Apr 2022.

9. (2022) GitHub - riscv-software-src/riscv-pk: RISC-V Proxy Kernel. In: GitHub. https://github.com/riscv-software-src/riscv-pk. Accessed 5 Apr 2022.
10. (2022) GitHub - riscv-software-src/riscv-isa-sim: Spike, a RISC-V ISA Simulator. In: GitHub. https://github.com/riscv-software-src/riscv-isa-sim. Accessed 5 Apr 2022.
11. (2022) Why RTOS and What is RTOS?. In: FreeRTOS. https://www.freertos.org/about-RTOS.html. Accessed 5 Apr 2022.
12. (2022) Mastering the FreeRTOS Real Time Kernel. In: Freertos.org. https://www.freertos.org/fr-content-src/uploads/2018/07/161204_Mastering_the_FreeRTOS_Real_Time_Kernel-A_Hands-On_Tutorial_Guide.pdf. Accessed 5 Apr 2022.
13. (2022) FreeRTOS - Free RTOS Source Code Directory Structure. In: FreeRTOS. https://www.freertos.org/a00017.html. Accessed 5 Apr 2022.
14. (2022) GitHub - illustris/FreeRTOS-RISCV: A port of FreeRTOS for the RISC-V ISA. In: GitHub. https://github.com/illustris/FreeRTOS-RISCV. Accessed 5 Apr 2022.