



StarFive  
赛昉科技

# Analysis of Running Real-Time Linux on VisionFive 2

Version: 1.12

Date: 2024/03/27



# Legal Statements

Important legal notice before reading this documentation.

## PROPRIETARY NOTICE

Copyright © Shanghai StarFive Technology Co., Ltd., 2024. All rights reserved.

Information in this document is provided "as is," with all faults. Contents may be periodically updated or revised due to product development. Shanghai StarFive Technology Co., Ltd. (hereinafter "StarFive") reserves the right to make changes without further notice to any products herein.

StarFive expressly disclaims all warranties, representations, and conditions of any kind, whether express or implied, including, but not limited to, the implied warranties or conditions of merchantability, fitness for a particular purpose, and non-infringement.

StarFive does not assume any liability arising out of the application or use of any product or circuit, and specifically disclaims any and all liability, including without limitation indirect, incidental, special, exemplary, or consequential damages.

All material appearing in this document is protected by copyright and is the property of StarFive. You may not reproduce the information contained herein, in whole or in part, without the written permission of StarFive.

## Contact Us

Address: Room 502, Building 2, No. 61 Shengxia Rd., China (Shanghai) Pilot Free Trade Zone, Shanghai, 201203, China

Website: <http://www.starfivetech.com>

Email:

- Sales: [sales@starfivetech.com](mailto:sales@starfivetech.com)
- Support: [support@starfivetech.com](mailto:support@starfivetech.com)



---

# Contents

List of Tables.....	4
List of Figures.....	5
Legal Statements.....	ii
Preface.....	vi
<b>1. Introduction.....</b>	<b>8</b>
<b>2. Linux in Real-Time Systems.....</b>	<b>9</b>
2.1. Real-time systems.....	9
2.2. Real-Time Extensions.....	10
2.3. Task Scheduling.....	12
2.4. Kernel Preemption.....	13
2.5. Scheduling Latency.....	14
<b>3. Measurement Setup.....</b>	<b>15</b>
3.1. Kernel Configuration.....	15
3.2. Load Generation.....	15
3.3. Latency Measurement.....	16
<b>4. PREEMPT_RT For RISC-V.....</b>	<b>18</b>
4.1. Patch Application.....	18
4.2. LAZY_PREEMPT.....	19
<b>5. Cyclist Test Result.....</b>	<b>21</b>
5.1. General Functionality.....	21
5.2. Latency Measurements.....	21
<b>6. Analysis.....</b>	<b>24</b>
6.1. Latency Analysis.....	24
6.2. Suitability for Real-Time Applications.....	25
6.3. Future Work.....	26
<b>7. Conclusion.....</b>	<b>28</b>
<b>8. Appendix A: Real-Time Linux Defconfig.....</b>	<b>30</b>
<b>9. APPENDIX B: Standard Deviation.....</b>	<b>31</b>



# List of Tables

Table 0-1 Revision History..... vi

Table 5-1 Latency Measurement Results..... 22





# List of Figures

Figure 2-1 Real-Time System Service Utility.....	9
Figure 2-2 Different Architectures of Real-Time Linux Systems.....	11
Figure 2-3 Real-Time Scheduling Principle [6].....	12
Figure 2-4 Scheduling Latency Components.....	14
Figure 3-1 Cyclicttest Latency Measurement.....	17
Figure 9-1 Formula of Standard Deviation.....	31



# Preface

About this guide and technical support information.

## About this document

This thesis was written by StarFive, as a part of the JH-7110 SoC project. This thesis introduces running RT-Linux on VisionFive 2 board, showing the performance of the RT-Linux on the VisionFive 2 board. Also, it introduces the development of the RISC-V RT-Linux. One latest and important news is that RT-Linux is officially support RISC-V architecture starting form kernel 6.6 (LTS). Several years later we can see RISC-V SoC will be running on industrial projects.




## Revision History

Table 0-1 Revision History



Version	Released	Revision
1.12	2024/03/27	Updated the data in <a href="#">Latency Measurements (on page 21)</a> .
1.11	2024/01/25	Added the description about RT-Linux officially support RISC-V architecture from kernel 6.6.
1.1	2023/10/31	Updated <a href="#">Appendix A: Real-Time Linux Defconfig (on page 30)</a> .
1.0	2023/06/13	The first official release.

## Notes and notices

The following notes and notices might appear in this guide:

-  **Tip:**  
Suggests how to apply the information in a topic or step.
-  **Note:**  
Explains a special case or expands on an important point.
-  **Important:**  
Points out critical information concerning a topic or step.



-  **CAUTION:**  
Indicates that an action or step can cause loss of data, security problems, or performance issues.
-  **Warning:**  
Indicates that an action or step can result in physical harm or cause damage to hardware.





---

# 1. Introduction

This thesis was written by StarFive, as a part of the JH-7110 SoC project. This thesis introduces running RT-Linux on VisionFive 2 board, showing the performance of the RT-Linux on the VisionFive 2 board. Also, it introduces the development of the RISC-V RT-Linux. One latest and important news is that RT-Linux is officially support RISC-V architecture starting form kernel 6.6 (LTS). Several years later we can see RISC-V SoC will be running on industrial projects.





---

## 2. Linux in Real-Time Systems

Real-time systems are commonly used when a given task has to be completed within a certain time limit. Such systems need to be very accurate, predictable, and deterministic, so these timing requirements can be guaranteed already at the development stage. Therefore, real-time capable operating systems need to be specially crafted for this purpose. This leads to different design decisions and internal implementation in comparison to what would be used, for example, in GPOSSs.

This chapter describes different aspects of real-time capable Linux systems and especially focuses on the PREEMPT\_RT patch. First, in [Real-time systems \(on page 9\)](#), the general concept of a real-time system is explained. Afterward, in [Real-Time Extensions \(on page 10\)](#), multiple real-time extensions for the Linux kernel are presented. Then, the basics of task scheduling are introduced in [Task Scheduling \(on page 12\)](#). Next, different preemption settings available in Linux are described in [Kernel Preemption \(on page 13\)](#). After that, various aspects of scheduling latencies are presented in [Scheduling Latency \(on page 14\)](#).

### 2.1. Real-time systems

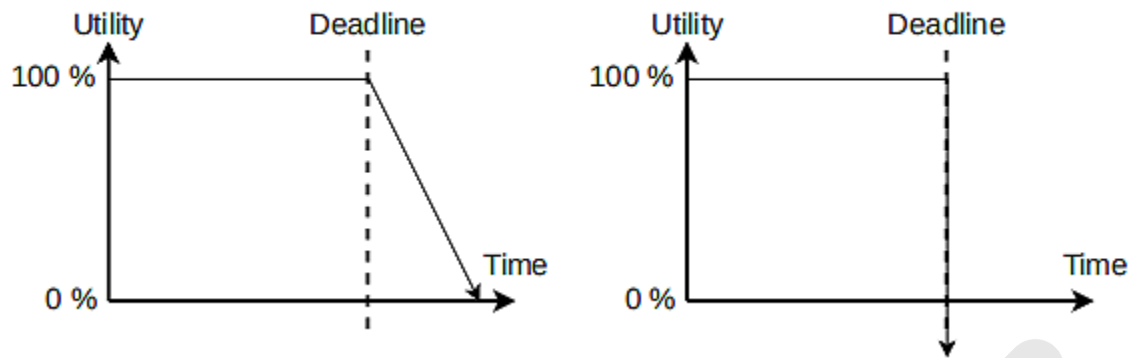
Real-time systems are used when the given application needs to meet some specific timing requirements. The criticality of these requirements usually varies quite a lot, so it is common to divide these real-time systems having different requirements into two distinct categories.

- **Soft real-time systems** will only have degraded results if the timing requirements
- **Hard real-time systems** will experience a catastrophic failure if the timing requirements are not met.

The utility differences between these two systems are illustrated in [Figure 2-1 : Real-Time System Service Utility \(on page 9\)](#). In a soft real-time system, the utility of the result is 100 % until the deadline is reached. After that, the result starts to gradually degrade according to some application-specific curve, eventually reaching zero at some point. This level of real-time is often relatively easy to achieve, as occasional missing of the deadline can be accepted. Alternatively, the hard real-time systems behave the same as soft real-time systems until the deadline is reached, but right at this time instance, the utility of the result immediately becomes negative. This represents the effect of catastrophic failure that the system will experience at that point. Designing hard real-time systems is very challenging, as it must be guaranteed that the system will always meet the timing requirements.

**Figure 2-1 Real-Time System Service Utility**





(a) Soft real-time system (b) Hard real-time system

Soft real-time systems can be used in many typical non-critical embedded applications that do not have too strict timing requirements. Missing a deadline in a soft real-time system should only lead to reduced user experience or inaccurate results at worst. For example, the timing requirements should not be critical regarding the safety of the product under any circumstance. Hard real-time systems, on the other hand, are required in more critical applications that cannot ever miss the deadline in any operating conditions. Missing the deadline with these systems could even cause significant damage to property or severe injuries.

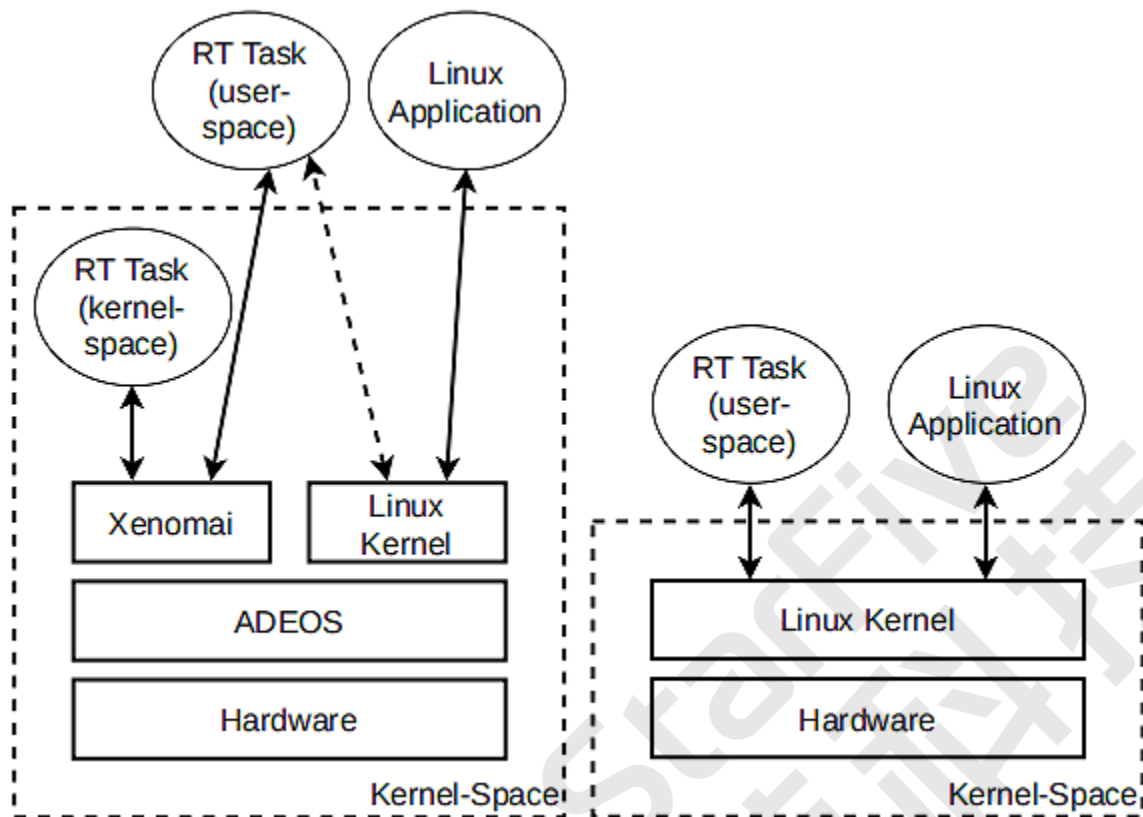
## 2.2. Real-Time Extensions

Like any GPOS, the Linux kernel is mainly optimized for throughput, as that is the most important design goal for such systems. This means that by default, unmodified Linux systems will regularly experience significant unbounded latencies that cannot be accepted in a real-time system. However, it is possible to add the missing real-time capabilities to the Linux kernel. This way, the operating system can provide the best of both worlds by offering a wide range of system services and acceptable real-time performance. Generally, there are two different ways of implementing the real-time capabilities, either by implementing a co-kernel-based system or by patching the mainline kernel, as presented in [Figure 2-2 : Different Architectures of Real-Time Linux Systems \(on page 11\)](#). In the first option, the Linux kernel is run as a normal process under a separate real-time scheduler. When this same scheduler also handles real-time tasks and system events, the real-time portion can be quite easily separated from the rest of the system. The second option is to leave the underlying kernel architecture untouched and instead introduce a set of changes to the kernel itself. In this configuration, the Linux kernel is responsible for simultaneously scheduling both normal and real-time tasks executing within the system.

In co-kernel systems, the Linux kernel is isolated from real-time tasks in some way.



Figure 2-2 Different Architectures of Real-Time Linux Systems



(a) Co-kernel (Xenomai) (b) Patch (PREEMPT\_RT)

One quite popular example of such a system is the Xenomai project, utilizing an intermediate layer between the hardware and Linux kernel, which effectively acts as an interrupt dispatcher and scheduler. Basically, it handles the system events so that processing is always prioritized for the highest priority real-time tasks to guarantee the specified timing requirements. Another viable alternative is a PREEMPT\_RT patched Linux, i.e., Real-Time Linux kernel. In this approach, underlying architecture remains exactly the same before. Previously, the PREEMPT\_RT introduced a very significant number of changes to the kernel, but today many of these features have already been merged into the mainline kernel. This work is still ongoing, but eventually the whole PREEMPT\_RT patch should be included in the mainline kernel.

Choosing between these two architectures is very much application-dependent. Typically, approaches utilizing co-kernel can achieve slightly lower latencies. But on the other hand, the system complexity is increased, and real-time tasks need to be specially crafted for the used real-time kernel. That is the opposite of the PREEMPT\_RT patched Linux kernel as the architecture is simpler, and the real-time tasks can be written almost like any other regular application. Also, the overall performance of patched Linux is on average better than what the co-kernel counter parts can achieve.

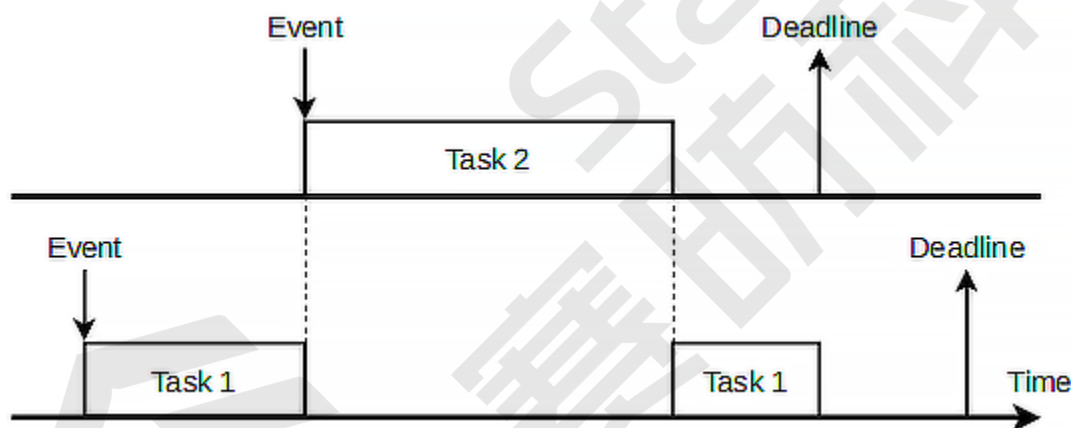


## 2.3. Task Scheduling

A scheduler, in the context of operating systems, simply decides what task is executed at a given instance of time. Often it is one of the most critical components of any operating system. Usually, GPOSs and RTOSs have different preferences for scheduling decisions, so multiple scheduler implementations have been developed for both cases. Having this distinction between the categories is important, as the design goals are so different.

When designing real-time systems, the most appropriate scheduling algorithm always depends on the situation. Perhaps the simplest, yet still useful, available real-time scheduling algorithm can be thought of as being the fixed-priority preemptive scheduling algorithm, as illustrated in [Figure 2-3 : Real-Time Scheduling Principle \[6\] \(on page 12\)](#). In this scheduling scheme, every task gets a static priority assigned to them at the design phase. During the system operation, the scheduler simply guarantees that the current task always has the highest possible priority. As the scheduling is preemptive, it means that even a task that is in the middle of executing something can be briefly aborted to let another higher priority task execute instead.

Figure 2-3 Real-Time Scheduling Principle [6]



Compared to the previously mentioned fixed-priority preemptive scheduling algorithm the Linux scheduler is a bit more complicated, but fundamentally it handles real-time tasks quite similarly. The Linux scheduler implements several scheduling policies which can be divided into non-real-time and real-time scheduling groups. Normal applications typically execute with one of the non-real-time policies, and they will always have lower priority than tasks running with any of the real-time policies. The Linux scheduler supports three distinct real-time policies that can be assigned to any task requiring a real-time priority.

- **SCHED\_FIFO**: a task running with this policy gets to execute until it finishes and voluntarily preempts, or a higher priority task preempts it.
- **SCHED\_RR**: tasks are only allowed to run at maximum with a specified time slice before being preempted if not preempted by higher-priority task.
- **SCHED\_DEADLINE**: is a policy implementing a task execution deadline-based scheduling algorithm.



All tasks executing with any of the real-time policies have a static real-time priority as signed to them. With **SCHED\_FIFO** or **SCHED\_RR** policies, this priority can be between 1 (low) and 99 (high). A task with **SCHED\_DEADLINE** is always executed with an effective priority of 100 having the highest priority possible. Any task executed with higher priority will always have precedence over another task with lower priority. With these policies, it is possible to carefully design the execution of a real-time system to meet even demanding timing requirements.

## 2.4. Kernel Preemption

The mainline Linux kernel currently has three preemption settings available: server, desktop, and low-latency desktop. With the introduction of the **PREEMPT\_RT** patch, a fourth real-time option becomes available. By using these options, it is possible to trade throughput for latency determinism. These options are described in the following list ordered from worst to best in terms of real-time performance.

- **Server** is the traditional preemption model. With this selection, the kernel code is executed with preemption disabled for the maximum throughput.
- **Desktop** preemption model adds explicit preemption points to the kernel code. This option provides better responsiveness at the cost of slightly lower throughput.
- **Low-Latency Desktop** reduces the latencies by making all normal kernel code preemptible. This setting allows better reaction times to interactive events.
- **Real-Time** option practically makes the whole kernel preemptible, including the most critical sections. This option is available only when the **PREEMPT\_RT** patch is applied.

As the preemption option names suggest, each one of these settings has an appropriate use case. The server preemption can be used in server installations where throughput is the single most important figure. On the other hand, real-time preemption should be used in embedded systems where the absolute throughput is not critical but rather the maximum experienced latency is. Therefore, different preemption levels in Linux allow for great flexibility to be utilized in varying environments, i.e., the same operating system can be used in servers and embedded systems. Overall, this synergy is very beneficial for the whole ecosystem as improvements or fixes implemented for server systems can be also automatically applied for small embedded devices.

Even if some of the preemption settings claim to have reduced latency, in practice, the real-time setting is the only viable option for any real-time system. In this configuration, the majority of spinlocks are converted to normal sleeping locks, interrupt handlers are threaded, and high-resolution timers are used for precise timing. Additionally, there are some other more insignificant changes introduced. With these improvements, practically the whole kernel is completely preemptible. Only things like very low-level event handling are executed in a non-preemptible context. Altogether, the real-time preemption model significantly improves the system responsiveness but decreases the overall performance as every introduced change causes some additional overhead.

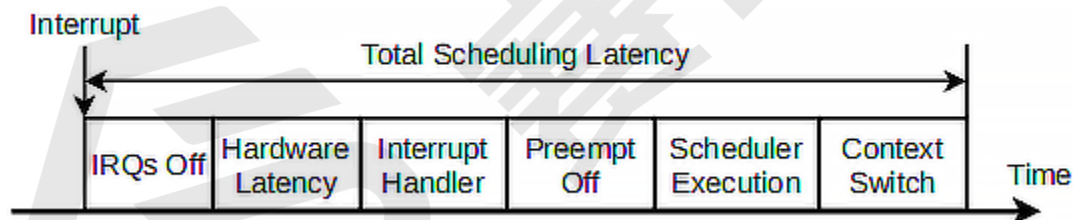


## 2.5. Scheduling Latency

Scheduling latency measures the elapsed time from an external event or an interrupt to the start of the execution of the related task. Usually, this would mean, for example, a delay between a timer interrupt and the beginning of executing some periodic task. The scheduling latency is often the single most important value to evaluate when observing a system's real-time performance. Basically, it combines all sources of latencies into a single variable that is easy to analyze, measure, and compare.

However, for a more detailed analysis, it is often required to individually study the different parts of the scheduling latency. These separate elements are listed in [Figure 2-4 : Scheduling Latency Components \(on page 14\)](#). The first cause of latency always happens right after the interrupt at the hardware level. Typically, the actual latency caused by hardware is very minimal, just a few clock cycles, but at this stage, additional delay can be experienced if the system interrupts are completely disabled. This can sometimes extend longer periods and is dependent on the currently executing software. The next cause of latency is the immediate interrupt handling routine. Often this part can contain some additional operating system logic, but in general, it is kept as short as possible in real-time systems. After the interrupt is handled, a decision about the next executing task can be done in the scheduler. The scheduler itself is most of the time quite fast, but again at this point, previously running software can cause additional latencies. If the preemption is disabled, the previously executed program is at this point resumed and executed until the preemption is again enabled. Finally, after these steps, the scheduler can schedule a task corresponding to the interrupt by performing a context switch.

**Figure 2-4 Scheduling Latency Components**



With these different parts of latencies, it can be seen that other parts but disabled interrupts and preemption latencies are always fairly constant and predictable. Therefore, these two causes of latencies are significant as they depend on the currently executing program, which means that the delays can extend for a lengthy period. Optimizing and minimizing these two scheduling latency components in the Linux kernel is the most important goal of the **PREEMPT\_RT** patch.



---

## 3. Measurement Setup

Various RISC-V development platforms and software utilities were used in the process of measuring and analyzing the real-time performance of the Real-Time Linux system. The selected development platforms express typical RISC-V systems currently available. In addition, the utilized measurement tools are common in the industry and widely used in other similar studies, thus they provide easily comparable results.

### 3.1. Kernel Configuration

The RISC-V architecture has been supported by the Linux kernel since version 4.15, dating back to 2017. StarFive VisionFive 2 BSP now uses the 5.15.0 version of kernel. So the BSP kernel 5.15.0 is chosen to apply RT-Linux patches. Due to RT-Linux not supporting the RISC-V architecture on kernel 5.15, kernel has multiple custom changes that allow PREEMPT\_RT to function on RISC-V. These changes are described thoroughly in [PREEMPT\\_RT For RISC-V \(on page 18\)](#). This customized kernel codebase was used for all measurements and latency analysis.

The final measurements featured two differently configured kernels. The only difference between the kernels was the preemption setting, but otherwise, the used configurations were the same. In the real-time kernel configuration, the preemption setting was set to the real-time mode. Whereas in the mainline kernel configuration, this same setting was instead set to low-latency desktop.

### 3.2. Load Generation

When a given computer system is idling, i.e., doing nothing, it can typically react to external events very quickly. But, on many systems, this behavior can drastically change when a load is applied. For this reason, measuring the latencies of a real-time platform without any load usually does not give representative results. Having the actual application software running would ideally allow for measuring the real latencies a particular system experiences. However, this software might not always be available, or running it would not be practical for some reason. Also, if the application is doing some specific operation very rarely, catching the longest possible latencies could take a significant amount of time. For these situations, it is usually best to run some artificial loads during the latency measurements.

There are many tools available for this purpose, but *stress-ng* (version 0.13.0) was selected for this test, as it is one of the most flexible utilities and is already widely used in the industry. It has been specially designed to test various operating system interfaces and to exercise physical subsystems of computer platforms. The original purpose of *stress-ng* was to find hardware issues such as thermal overruns. Today, the tool has a very wide range of stressors capable of, for example, discovering kernel bugs and executing benchmarking. But most importantly, it can be also used to reveal unexpected latencies from real-time systems.



In total, the *stress-ng* implements over 220 different stressors which makes it possible to design a comprehensive set of tests. For this study, every category was selected so that they loaded distinct parts of the system but also mimicked realistic use cases best as possible. To get broad information about the system behavior, a total of five different stress categories were selected for further inspection. Once each of the categories was executed, always a total of four processes were started, one for each core. This made sure that all cores had an equal amount of load applied to them.

- **idle**: system without any background workers performing computation. The system will be able to respond as quickly as possible (*stress-ng* not invoked).
- **cpu**: create workers that perform various floating-point arithmetic operations. This will add CPU and cache stress (*stress-ng --matrix 4*).
- **os**: create workers that exercise various *set\*()* system calls. This will stress internal kernel operations and data structures (*stress-ng --set 4*).
- **memory**: create workers that continuously call *mmap()* operations and write to the allocated memory. This will stress memory handling (*stress-ng --vm 4*).
- **storage**: create workers that repeatedly create and remove directories. This will stress the filesystem and Input/Output (I/O) infrastructure (*stress-ng --dir 4*).

A simple process was followed to select the most appropriate stressors for each category. First, all available stressors were briefly tested, and the measured latencies were recorded. Then, the results were sorted according to the maximum observed latencies, and from this list, it was possible to pick the most appropriate stressors. The final selection was done so that each one of the categories would represent a typical workload in an embedded real-time system, but in a way that would reveal some interesting results. Therefore, it would have been possible to find stressors that generate even longer latencies with some specially crafted arguments, but as they would not represent realistic scenarios, these options were disregarded.

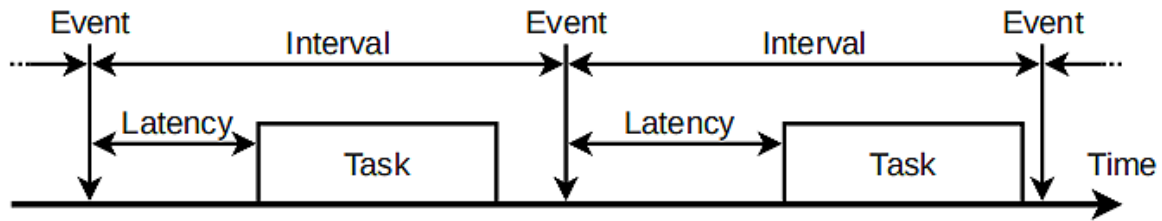
### 3.3. Latency Measurement

There are multiple ways of defining the latency of a given system and equally many ways of measuring it. For this study, a simple and widely used method of evaluating the real-time capabilities of a given system based on scheduling latency was selected. This mimics the behavior of many typical control systems that execute a periodic task that must be precisely timed. Also, this same situation arises when the system is reacting to some external asynchronous events.

The tool in question *cyclictest* (version 2.20) was used as the main measuring utility in the latency measurements. It was originally built by the authors of PREEMPT\_RT to help with the development effort by precisely measuring the system latencies, as presented in [Figure 3-1 : Cyclictest Latency Measurement \(on page 17\)](#). The *cyclictest* works by starting a regular master thread, which will start several real-time measuring threads. Each one of these real-time threads is set to periodically wake up after a defined interval. Every time they are woken up, the difference between intended and actual wake-up time is then recorded and passed along to the master process. The master process stores all these results, and once the test ends, it outputs the aggregated results.



Figure 3-1 Cyclicttest Latency Measurement



For SMP systems, such as the VisionFive 2, the *cyclicttest* provides statistics individually for each core. However, having these separated results is not generally important for overall latency evaluation. For this reason, the calculations and graphs presented in this thesis are simplified by simply adding the results from every core together to get a single number of latencies for the whole system. This makes the provided measurement results easier to visualize and understand.

The *cyclicttest* should be carefully configured for each system and measurement scenario.

Most importantly, the priority and interval need to be set according to the measurement situation. To measure system latencies, the priority should be set higher than the load running on the system. The most optimal value for the interval would be slightly bigger than the maximum observed latency. Overall, multiple options were set to specific values to get the most appropriate latency results.

- **--priority=99** selects the measuring thread real-time priority equal to 99.
- **--interval=200** sets the intended measuring thread wake-up period to 200  $\mu$ s.
- **--duration=10m** sets the total duration of the measurement to 10 minutes.
- **--histogram=200** enables histogram generation from the measurement up to latencies of 200  $\mu$ s.
- **--smp** causes a single pinned thread to be launched for each available core.
- **--mlockall** makes process memory lock paging once everything is allocated.

The *cyclicttest* tool also automatically sets some noteworthy parameters, which could also be manually set, to default values based on these selections. Most notably the distance parameter of measurement thread intervals is automatically set to zero because the histogram option is specified. In addition, the scheduling policy is by default set to *SCHED\_FIFO*. Altogether, these parameters are quite typical and well-suited for measuring latencies that a real-time application would experience.



## 4. PREEMPT\_RT For RISC-V

This chapter mainly describes the application of `PREEMPT_RT` in StarFive JH-7110 SDK kernel (Linux kernel v5.15) for test only. If it comes to the implementation of industrial projects, using the kernel v6.6 LTS is more suitable, because RT-Linux has already supports the RISC-V architecture in the kernel 6.6 LTS, and most of the drivers for StarFive JH-7110 can be found in kernel v6.6. Only needs to apply the RT-Linux patches on kernel v6.6 and compile the kernel containing the VisionFive 2 driver to use and test `PREEMPT_RT`.

The `PREEMPT_RT` patch already supports multiple architectures, and most of the important real-time functionality is implemented in architecture-independent kernel code. Therefore, it is safe to assume that these portions of the kernel are working correctly, and all changes required for RISC-V operation would have to be done only in architecture specific code. Within the Linux kernel source tree, this would include all files under the `arch/riscv/` directory and every RISC-V specific driver in the `drivers/` directory. The implemented changes to the Linux kernel are listed in [Appendix A: Real-Time Linux Defconfig \(on page 30\)](#) as a complete patch file.

This chapter discusses the different steps and modifications required when porting the basic functionality of the `PREEMPT_RT` patch to RISC-V architecture. First, in [Patch Application \(on page 18\)](#), the process of applying the `PREEMPT_RT` patch to the mainline kernel source tree is presented. Finally, in [LAZY\\_PREEMPT \(on page 19\)](#), RT-Linux configuration `LAZY_PREEMPT` is discussed.

### 4.1. Patch Application

The first step of getting the `PREEMPT_RT` working with RISC-V architecture was to apply the official patch file to the mainline Linux kernel source tree. Usually, with correctly selected versions, this would be a very simple step of just using the `patch` command to apply a single patch file. However, in this case, there was a very specific version of the Linux kernel required to correctly support the StarFive JH-7110 DevKit. Now the StarFive JH-7110 DevKit Linux version is 5.15.0.

Download 5.15.0-rt patch from: [This Link](#).

Download 6.6 patch from: [This Link](#).



#### Note:

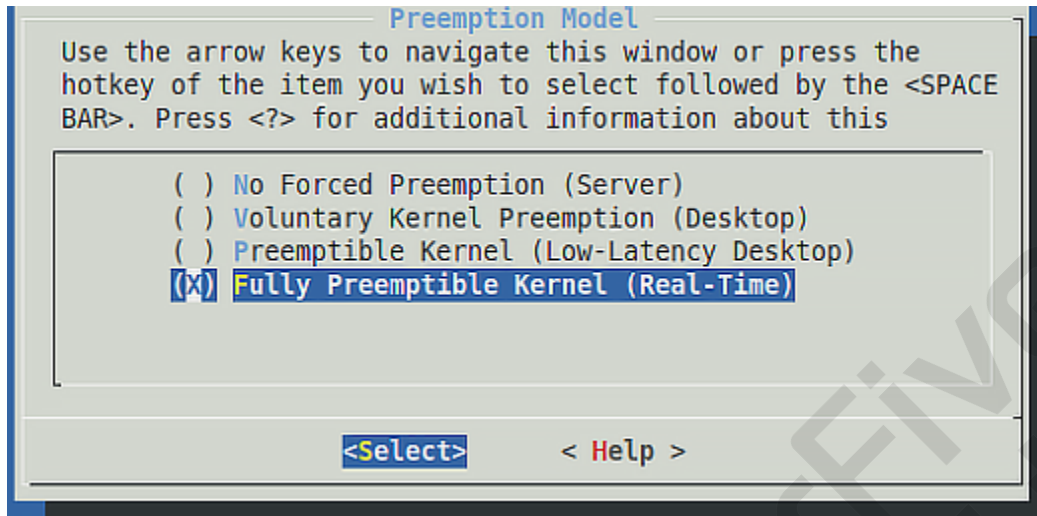
To apply this patch, please execute following command:

```
patch -p1 < (patch)
```

After applying the 5.15.0 RT patch, apply the patches in [Appendix A \(on page 30\)](#). These patches contain the `LAZY_PREEMPT` configuration for the RISC-V architecture and enable the `PREEMPT_RT` configuration for JH-7110 DevKit. Additionally, a modification has been made to the `kernel/sched/cpupri.c` file regarding task prioritization by patch: `0004-cpupri-a-work-`



around-for-non-rt-test-panic. This modification addresses an issue in the official RT patch where non-RT tasks would trigger a 'kernel panic' during a stress test. However, please note that this is a workaround.



## 4.2. LAZY\_PREEMPT

In the RT patches set. A LAZY\_PREEMPT configuration is not supported by RISC-V. But it is an important feature in RT-Linux. This patch enhances the non RT workload performance. It is necessary to support it. Though RT-linux not support RISC-V architecture. But we can get other architecture patch(ARM/x86) to implement it.

An implementation of RISC-V LAZY\_PREEMPT is list in Appendix A. The patch has been verified by OpenPLC test. Both latency and jitter are improved from test result.

For more detail about LAZY\_PREEMPT, below is the commit messages of LAZY\_PREEMPT.

*It has become an obsession to mitigate the determinism vs. throughput loss of RT. Looking at the mainline semantics of preemption points gives a hint why RT sucks throughput wise for ordinary SCHED\_OTHER tasks. One major issue is the wakeup of tasks which are right away preempting the waking task while the waking task holds a lock on which the woken task will block right after having preempted the wakee. In mainline this is prevented due to the implicit preemption disable of spin/rw\_lock held regions. On RT this is not possible due to the fully preemptible nature of sleeping spinlocks.*

*Though for a SCHED\_OTHER task preempting another SCHED\_OTHER task this is really not a correctness issue. RT folks are concerned about SCHED\_FIFO/RR tasks preemption and not about the purely fairness driven SCHED\_OTHER preemption latencies.*

*So I introduced a lazy preemption mechanism which only applies to SCHED\_OTHER tasks preempting another SCHED\_OTHER task. Aside of the existing preempt\_count each tasks sports now a preempt\_lazy\_count which is manipulated on lock acquiry and release. This is slightly incorrect as for lazyness reasons I coupled this on migrate\_disable/enable so some other mechanisms get the same treatment (e.g. get\_cpu\_light).*



*Now on the scheduler side instead of setting `NEED_RESCHED` this sets `NEED_RESCHED_LAZY` in case of a `SCHED_OTHER/SCHED_OTHER` preemption and therefor allows to exit the waking task the lock held region before the woken task preempts. That also works better for cross CPU wakeups as the other side can stay in the adaptive spinning loop.*

*For RT class preemption there is no change. This simply sets `NEED_RESCHED` and forgoes the lazy preemption counter.*





---

## 5. Cyclist Test Result

The Linux kernel is a very complicated piece of software, so it is not always easy to prove the exact latency characteristics of a given system. Usually, the latencies also highly depend on the workload and selected kernel configuration. The combination of these and many other parameters are endless, but practical measurements can give a good insight into latencies that would also be experienced in practical real-world applications.

This chapter presents the latency measurement results of this thesis. First, in [General Functionality \(on page 21\)](#), the general functionality of the measurement systems is discussed. Then, in [Latency Measurements](#), the measured latency table and relevant characteristic numbers are given from both kernels.

### 5.1. General Functionality

There really are not any quantitative measurements that could be done to evaluate the general functionality and correctness of a system. However, normal usage, comprehensive testing, and latency measurements already gave a good estimate of the overall system functionality. An incorrectly working system would have caused noticeable problems, such as wrong calculation results or kernel panic messages. The system functionality was constantly observed during the testing and development, which proved the general system to be working very reliably. But other, more thorough experiments were not executed to evaluate the system functionality.

Throughout all latency measurements, the system was completely stable and usable without any signs of problems. The kernel did not show any instability even when different areas were loaded heavily. During all the testing no system misbehavior was observed, which gives good confidence that the PREEMPT\_RT patched Linux was not experiencing any severe problems on RISC-V architecture.

### 5.2. Latency Measurements

The latency measurements were started by compiling appropriate kernel images, filesystems, bootloaders, and other requisite files to the local development machine. Both used kernels were configured as presented earlier in [Load Generation \(on page 15\)](#). Once all files were flashed to the SD card and the system was successfully booted, it was left idle for 3 minutes before any testing. During that time, the kernel had time to initialize internal data structures such as the random number generator to fully complete the boot process. This way, any of the internal kernel initializations did not affect the measurement results.

After this procedure, the kernel was ready for the actual measurements. For the VisionFive 2 contain DVFS driver. The CPU frequency can be changed in the test. To get the accuracy test result, set the CPU frequency to maximum(1.5 GHz)



1. Set maximum CPU frequency.

```
echo performance > /sys/devices/system/cpu/cpu0/cpufreq/scaling_governor
```

2. Set different load (cpu, os, memory or storage).

```
cpu: stress-ng --matrix 4
```

```
os: stress-ng --set 4
```

```
memory: stress-ng --vm 4
```

```
storage: stress-ng --dir 4
```

3. Run cyclitest.

```
cyclitest -m -S -p 99 -i 200 -q -D 10m -H 200
```

(The cyclitest arguments detail can see in [Latency Measurement \(on page 16\)](#))

Each latency measurement lasted for a total duration of 10 minutes resulting in approximately 1.2 million latency samples each. The same process was repeated for every load category using the mainline and the PREEMPT\_RT patched kernels without experiencing any problems. Altogether, there were a total of 10 different measurement combinations that were executed.

The dataset is presented in [Table 5-1 : Latency Measurement Results \(on page 22\)](#). It is easier to do a comparison between different categories and kernels using the tabular format. Most importantly the table contains the absolute maximum observed latencies from each measurement. Other important

calculated numbers are the average latency and respective standard deviation. Also, the minimum observed latencies are presented for completeness. The *cyclitest* tool reported measured practical clock resolution to be 1  $\mu$ s, so all presented calculations and measurements are rounded to the nearest microsecond.

Stdev is standard deviation of the latency. It reflects the jitter of the latency. The less the value of stdev, the better real-time system. The methods of computation and an example is list in Appendix B.

**Table 5-1 Latency Measurement Results**

Stress	Real-Time				Mainline			
	Avg( $\mu$ s)	Stdev( $\mu$ s)	Min( $\mu$ s)	Max( $\mu$ s)	Avg( $\mu$ s)	Stdev( $\mu$ s)	Min( $\mu$ s)	Max( $\mu$ s)
Idle	6	2	5	37	8	4	6	66
CPU	8	4	6	39	10	6	6	88
OS	9	8	6	46	20	16	6	9162
Mem	12	6	6	95	12	14	6	14125



Table 5-1 Latency Measurement Results (continued)

Stress	Real-Time				Mainline			
	Avg( $\mu$ s)	Stdev( $\mu$ s)	Min( $\mu$ s)	Max( $\mu$ s)	Avg( $\mu$ s)	Stdev( $\mu$ s)	Min( $\mu$ s)	Max( $\mu$ s)
Storage	10	7	7	49	16	14	7	3025

For the mainline kernel, the lowest maximum latency was measured when the system was idling, as would be expected. Also, in this particular case, the average latency and standard deviation were the smallest. When the CPU load was introduced to the system, it caused the latencies to rise, but the maximum latency was kept under 100  $\mu$ s. However, with every other type of load, noticeably longer latencies were experienced.

The OS and memory stress test in main line kernel seemed to much worse than real-time Kernel. With this type of load, the maximum latencies were in the order of milliseconds, and the average latency was also much higher with a significant standard deviation when compared to Real-time system.

With real-time kernel, the observed latencies were very similar to each other, across all different categories. All measured maximum latencies, except for the error category, were well below 200  $\mu$ s. Also, the observed standard deviations were consistently small throughout the tests, suggesting that most of the experienced latencies were close to the average numbers. Therefore, applying any of the selected loads to the system did not affect the system responsiveness by a significant amount.



---

## 6. Analysis

The PREEMPT\_RT patch improves the worst-case latency numbers in RISC-V when compared to the current mainline version. The numbers show that the minimum requirements for a real-time operation are already fulfilled by this system. Also, the studied RISC-V system seems to be capable of achieving very similar latency figures as some other architectures that are already officially supported. This observation confirms that the RISC-V has future potential, but to get this setup feasible for production-level usage there is still plenty of work and verification ahead.

This chapter analyses the measurement results presented earlier in [Cyclist Test Result \(on page 21\)](#) Chapter 4. First, different aspects affecting the latency performance are discussed in [Latency Analysis \(on page 24\)](#). Then, in [Suitability for Real-Time Applications \(on page 25\)](#), the suitability of RISC-V Real-Time Linux in practical applications is analyzed in detail. Finally, future work related to this setup is discussed in [Future Work \(on page 26\)](#).

### 6.1. Latency Analysis

As expected, with the addition of the PREEMPT\_RT patch, overall worst-case system latency was reduced significantly in almost every stress scenario. This is especially relevant in some of the most demanding categories, where the real-time kernel was able to achieve comparable latencies to an unloaded system. In these same test cases, the mainline kernel had significant latencies and could not even be considered to be acceptable for many real-time systems. Most notably, the latency in the out-of-memory error condition was very significant and reached well in the milliseconds' region with the mainline kernel. This was not the case in the real-time kernel where the same latency was still comparable to other categories. Based on these measurements, the reliability and determinism of the real-time kernel are confirmed to hold in terms of experienced scheduling latencies.

The single biggest reason behind the real-time kernel's superior performance seems to be the conversion of spinlocks to sleeping locks that also implement priority inheritance functionality. This is to be expected as the non-preemptible regions inside the kernel are significantly reduced by this change. Also, the conversion of most interrupt handlers to be threaded and thus subject to scheduling with lower priorities might slightly account for the better latencies measured with the real-time kernel. The reason why this change is probably not so relevant with this particular kernel configuration, as the conversion of spinlocks, is that there were not too many sources of interrupts enabled. Those that were required to be enabled seemed to be anyways quite fast to complete compared to the other sources of latencies. In the end, the latency improvements of PREEMPT\_RT probably did not appear to be that impressive when compared to the performance of the mainline kernel which is already quite close to being real-time capable. This is mostly because other significant changes historically introduced by PREEMPT\_RT are already merged, so they also benefit the mainline kernel.

Even if the real-time kernel is better than the mainline, it still has some sections that produce the over 150  $\mu$ s latencies that were consistently measured. Based on the tracing of the kernel internals, some insight into these latencies was acquired. The longest latencies appear to be exclusively



caused by timer interrupts, which are always executed in hard interrupt context, and occasional raw spinlocks that are still required. The other significant peaks in the latency histograms show that there are also some additional causes for smaller, but considerable latencies. However, analyzing these reliably by tracing is hard and probably unnecessary if the longer latencies still occur. Also, some of the latency differences between different categories are probably explained by hardware-specific issues. For example, a more demanding load to the system might cause additional cache misses, etc. that cannot be fully mitigated by the `PREEMPT_RT` patch.

Overall, the findings of this thesis are quite significant, as they show that the `PREEMPT_RT` patch on RISC-V can be made to work without too much additional effort. With the real-time preemption model enabled, the measured latencies are fairly predictable and generally in an acceptable range for a real-time system. In addition, there do not seem to be any fundamental problems with running the `PREEMPT_RT` on RISC-V architecture. However, it is possible that if some more advanced kernel features were enabled from the configuration, there might be some fundamental problems with the current RISC-V specific implementation. At least with more enabled features from the kernel, the latency performance could get worse.

Even though performing exact measurements on a Linux system is challenging, the results are very repeatable and sensible. However, the latency measurements will always have some uncertainty as the system is very asynchronous and mainly driven by interrupts. More exact results could have been observed by running each one of the latency measurements for a longer period, even for days. The general shape of the latency response, however, is still captured with the used shorter test duration. One actual source of possible errors in the presented results is the used `PREEMPT_RT` patch version, as the closest version was for 6.3 kernel. This means, that there is a possibility that up until the official release of this version, some new features could have been added to the kernel which are not handled properly by the `PREEMPT_RT`. However, this is quite an unlikely situation, and also the tracing results suggest that there should not be any problems regarding this. Anyways, using the results presented in this thesis as a conservative estimate is fine. Finally, it is also possible that there is still some oversight in the system configuration. Otherwise, all other sources of error should be mitigated and taken into account in the measurements.

## 6.2. Suitability for Real-Time Applications

Whether the presented system is suitable for practical usage is depended on the application. The measurements overall give a good reference for a wide variety of practical applications that stress the system in different ways. However, slightly differently configured, or selected stressors might have given different results. Likewise, it should be noted that *cyclicttest* can give somewhat optimistic numbers, so that would need to be carefully considered when referencing these numbers for real-world usage. Even the *cyclicttest* utility itself can interfere with the latency measurements. In addition, the used minimal real-time kernel is certainly not representative of practical applications, so changing the configuration might bring up additional challenges.

Regardless, already at this point, without any significant architecture-specific latency optimizations the system appears to achieve acceptable latencies. The current system can be considered soft or



firm but certainly not hard real-time capable. This means that in theory, the presented system would be suitable for industrial machinery, but in practice, official support by PREEMPT\_RT with extensive prior experience would be required. Starting from kernel v6.6, RT-Linux officially supports RISC-V architecture.

Other properties of RISC-V, of course, provide some appealing benefits in comparison to other architectures for industrial applications. It is a completely open ecosystem, and for example, it is easier to develop completely custom instruction-level extensions. The RISC-V ecosystem is also very modern as there is not any legacy burden on the system. All these aspects might very well make a Real-Time Linux running on the RISC-V platform a desirable system for some use cases in the future.

## 6.3. Future Work

This thesis presented only the very first steps of assessing the Real-Time Linux on RISC-V architecture. There is a huge number of possibilities to study this topic further as other related research is practically nonexistent. The future work could include, e.g., testing the effect of some additional kernel options, evaluating more stress categories, and looking into some kernel optimizations such as things related to the multi-core operation. Also, experiments from different hardware platforms, measurements about completely different metrics, and practical PREEMPT\_RT development work would be valuable in the future. Studying these topics further would give important information regarding the possibilities of RISC-V architecture for practical industrial real-time applications.

As stated earlier, the kernel configuration used in this thesis is very minimal and real systems would certainly need some additional features of the Linux kernel to be enabled. Because of this, it would be beneficial to do some further research by enabling some of the most important additional kernel configuration options and testing how they affect the latency performance. Excluding obvious debugging options, a properly behaving Real-Time Linux kernel should not be significantly affected by this. However, some parts of these options could introduce substantial latencies and thus would need some optimization work before they could be considered acceptable to be used in a real-time configuration. Also, simultaneously perhaps even additional errors related to locking and general real-time behavior could be discovered and fixed.

As the Linux systems are very complex, there are plenty of other interesting metrics in addition to the latency that could be measured. Having more statistics from areas like memory usage, power consumption, and system throughput would be very valuable information about other aspects of using a Real-Time Linux on RISC-V. In addition, a wider range of stress categories could be tested and evaluated as this study completely ignores areas such as networking or other sources that cause a significant number of interrupts. Designing these additional stress categories would most certainly have to be done together with the enabling of additional kernel features.

Regarding the current system, it might be possible to do some additional optimization on the settings used in this study and achieve even lower latencies. For example, restricting some interrupts to certain cores by setting CPU affinity bits might bring slight improvement to the experienced latencies on other cores. Also, some code-level optimization would probably be



possible. This optimization work should be targeted to areas amounting to the longest latencies as discussed previously. With these changes, the latencies could be even smaller than presented in this thesis. Also, the RISC-V architecture itself is under continuous development having new features proposed and ratified regularly. For example, there is ongoing work to design and implement a better and more flexible interrupt controller hardware. This proposal for Core-Local Interrupt Controller (CLIC) aims to add completely new features to the current PLIC-based architectures. Significant improvements would include support for interrupt preemption and selective vectoring among others. These features could be useful or even critical for some time-sensitive applications, so in the future, RISC-V would be even more capable of handling different real-time workloads. However, it should be noted that these details are subject to change as the CLIC specification is still in the very early development phase. There will certainly be other similar development of new hardware blocks in the future.

Alongside RISC-V hardware, also the general software support will continue to improve. The most important pieces of software, e.g., compilers, debugging tools, and Linux kernel already have great support for RISC-V but of course, it is still a bit behind other older and more established architectures. For example, every feature of the Linux kernel is not yet fully supported when RISC-V architecture is used. But most certainly the ever-growing community and commercial interest will bring fixes to some of these issues. Overall, the whole RISC-V ecosystem is getting better every day.

In the end, the single most important thing for using Real-Time Linux on RISC-V architecture would be official support. This would guarantee the functionality to a certain degree, as at that point, the community using that product would be considerably larger. Starting from kernel v6.6 LTS, RISC-V has become the official support for RT-Linux, and most of StarFive JH-7110 driver code has been accepted in Kernel 6.6, which is of great advantage for JH-7110 to enter RT-Linux commercial industrial. Based on this article, the RISC-V architecture should be fully suitable for higher demand industrial applications.



---

## 7. Conclusion

As real-time systems become more complex and require more advanced features, the benefits of using complete GPOS to help implement some of these requirements more rapidly become increasingly appealing. Using the PREEMPT\_RT patch with Linux kernel and one of the many officially supported architectures is currently one of the most

viable ways of achieving that goal if the real-time requirements are not too strict. In the meantime, the RISC-V is already getting lots of attention in the industrial space, and in the future, it will probably be a prominent player in the field of embedded devices. Once the availability of commercial RISC-V solutions improves and the Linux kernel support for RISC-V architecture progresses, together they bring a very interesting platform that can compete with other architectures. Inevitably, at some point, there will be genuine demand and support for advanced RISC-V systems that run Real-Time Linux.

With a couple of small tweaks presented in this thesis, the core of the PREEMPT\_RT patch on RISC-V looks to be already fully functional. Most importantly, it does not seem to contain any fundamentally problematic sections of code that produce kernel panics, deadlocks, or other significant issues. Also, the latency measurements show promising results regarding real-time usage scenarios. This unoptimized Real-Time Linux running on a RISC-V platform can already achieve similar results to other officially supported architectures. However, the evaluated system was very minimal and therefore does not yet necessarily represent a practical system.

With PREEMPT\_RT applied, even in high load situations, the maximum observed latency was kept below 150  $\mu$ s. In a severe, system out-of-memory error condition, the maximum latency was well below 250  $\mu$ s. This is significantly lower when compared to the mainline kernel, which experienced latencies well in the milliseconds' region. Overall, the maximum observed latency seems to be very deterministic, so they do not depend on the system load, which is the single most important requirement for the operation of all real-time systems. These numbers are especially impressive as there are not currently any RISC-V architecture or driver-specific latency optimizations done. The main factors contributing to the current latencies still present in the PREEMPT\_RT kernel seem to be related to timer interrupts and occasional raw spinlocks. Optimizing some of these parts could improve the experienced latencies even more.

Based on the latency measurements, the PREEMPT\_RT on RISC-V could be already suitable for some situations that do not need better response times than 200  $\mu$ s. Without any error conditions, the current system should be able to reliably respond within that time limit. However, Like LAZY\_PREEMPT and other feature required to be supported. Besides, this is not yet possible in practice, as there would need to be official support for the RISC-V architecture in the PREEMPT\_RT patch for any serious project. Probably in addition to this, there would need to be some proven track record to give enough confidence to use such systems in industrial projects. At the bare minimum, it will take several years even before this can be considered. It is also worth noting that adding more options to the kernel might cause some longer latencies. This matter would require additional studying in the future to know the different options that might cause problems.



The future of real-time capable Linux systems running on RISC-V architecture, in general, seems to be very promising. Different real-time extensions of Linux will be more prominent than ever before and PREEMPT\_RT will have an important role in that development. Fully mainlining the PREEMPT\_RT patch to Linux kernel source would only accelerate the development as there is exposure to an even bigger community. Starting from kernel v6.6, RT-Linux officially supports RISC-V architecture. With official support, it is possible to better track and resolve RT-Linux kernel issues. Also, the general RISC-V support in the Linux kernel, as well as the architecture agnostic parts of PREEMPT\_RT itself, will develop continuously. There might even be some completely new features implemented to the Linux kernel that will enable even lower latencies in the future. This work will most certainly help with the future adoption of RISC-V as a viable alternative to other existing architectures. Regardless, with this study completed, the RISC-V architecture has started its journey to achieving official support for PREEMPT\_RT patch.



---

## 8. Appendix A: Real-Time Linux Defconfig

1. RISC-V RT patch download link: [0001-riscv-allow-riscv-preempt\\_rt-config.patch](#)
2. Config\_Add\_PREEMPT patch download link: [0002-enable-full-preempt-rt-config.patch](#)
3. LAZY\_PREEMPT patch download link: [0003-riscv-rt-add-riscv-lazy-preempt-support.patch](#)
4. CPUPRI\_workaround\_RT\_test patch download link: [0004-cpupri-a-work-around-for-non-rt-test-panic.patch](#)



### Note:

Please apply the patches in the above order.

To apply the patches, execute the following command:

- With git repository:

```
git am (patch)
```

- Without git repository:

```
patch -p1 < (patch)
```

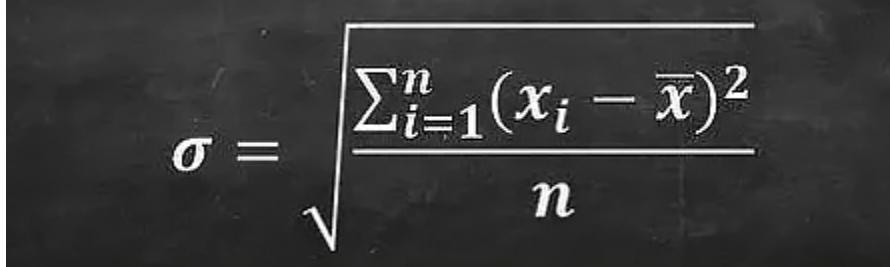


---

## 9. APPENDIX B: Standard Deviation

The formula of standard deviation is listed below:

Figure 9-1 Formula of Standard Deviation


$$\sigma = \sqrt{\frac{\sum_{i=1}^n (x_i - \bar{x})^2}{n}}$$

$n$  is the sample count,  $\bar{x}$  is the mean value of the sample count.

Here is an example of standard deviation calculation.

Total 4 latency counts, 4, 4, 6, 6.  $n$  is 4, the mean value is 5. The standard deviation value is 1.