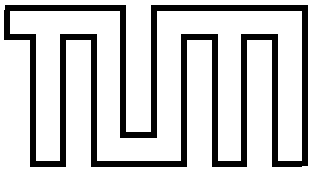


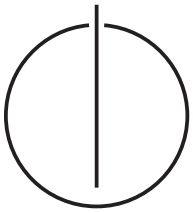
FAKULTÄT FÜR INFORMATIK

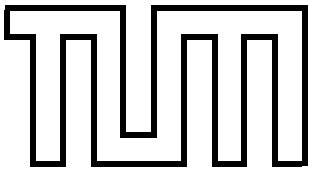
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Validating the Real-Time Capabilities of the ROS Communication Middleware

Jonas Sticha





FAKULTÄT FÜR INFORMATIK

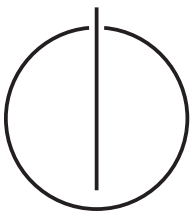
DER TECHNISCHEN UNIVERSITÄT MÜNCHEN

Bachelor's Thesis in Informatics

Validating the Real-Time Capabilities of the ROS Communication Middleware

Validierung der Echtzeitfähigkeit der ROS Kommunikationsmiddleware

Author:	Jonas Sticha
Supervisor:	Prof. Dr. Uwe Baumgarten
Advisors:	Daniel Krefft M. Sc. Dr. Lukas Bulwahn (BMW Car IT GmbH)
Submission date:	July 15th, 2014



Declaration

I confirm that this bachelor's thesis is my own work and I have documented all sources and material used.

München, den 15. Juli 2014 Jonas Sticha

Abstract

As the complexity of modern real-time systems increases steadily, they often consist of multiple software components interacting with each other through a middleware. Such a middleware, amongst others, has to be capable of meeting certain real-time requirements. To evaluate whether the Robot Operating System (ROS) might be suitable for such use, different components of ROS were validated concerning their real-time capabilities. Therefore, three test suites have been developed, and executed repeatedly. The hereby gained insights, the used test environment, as well as the corresponding theoretical background, are summarized in this bachelor's thesis.

Kurzfassung

Die stetig wachsende Komplexität moderner Echtzeitsysteme hat zur Folge, dass sich diese immer häufiger aus mehreren Softwarekomponenten zusammensetzen, welche über eine Middleware miteinander kommunizieren. Solch eine Middleware muss unter anderem in der Lage sein, gewisse Echtzeitanforderungen zu erfüllen. Um bewerten zu können, ob das Robot Operating System (ROS) für solch einen Einsatz in Frage kommt, wurden mehrere Komponenten von ROS bezüglich ihrer Echtzeitfähigkeit validiert. Zu diesem Zweck wurden drei Testsuites entwickelt und wiederholt ausgeführt. Die Erkenntnisse die dabei gesammelt wurden, die verwendete Testumgebung, sowie das nötige theoretische Hintergrundwissen sind in dieser Bachelorarbeit zusammenfassend dargestellt.

Contents

List of Figures

1 Introduction

Computer science has enabled many ground-breaking and world-changing inventions and developments over the past few decades. Probably one of the currently most interesting topics is autonomously driving vehicles. While there are frequent news updates from Google on their newest progress also other companies, like the vehicle manufacturer BMW, are working on that topic. Even though autonomously driving vehicles might still sound very futuristic, it is only a matter of time until they will become reality. However, at the moment researchers still face a lot of unresolved issues.

Piloting a vehicle constantly demands for a lot of decisions to be made. These decisions depend on influences of the environment, e.g., the course of the road, the weather, or the path that other vehicles are headed. A computer making these decisions therefore has to perform an enormous amount of calculations on an even greater amount of data coming from sensors scattered throughout the vehicle. Handling this complexity of necessary computations requires a distributed target system where different tasks are implemented as individual software components. To allow these software components to exchange information with each other, the use of a communication middleware is required. The information that such a middleware transmits is in many cases critical to safety. Imagine a scenario where a software component detects an obstacle on the vehicle's current route. This software component now has to pass this information to another component that can manipulate the steering of the vehicle accordingly to prevent any damage. If this information reaches its recipient too late, an accident might occur. To prevent this cause for an accident, the used middleware must guarantee certain real-time capabilities.

A prevalent middleware implementation that can comply with the functional requirements of such a system is the ROS Communication Middleware. So far there is however no statement concerning its real-time characteristics. This thesis is a first step towards validating whether ROS is also appropriate for use in real-time critical systems such as the one exemplified earlier on. For this purpose, three test suites were developed that investigate the real-time behavior of different components of ROS. For the execution of these test suites, a test environment was set up on that a series of test runs were conducted, with different parameters and under different conditions, to gain insights on the real-time behavior of ROS.

The thesis is structured as follows: Chapter 2 gives an overview of the necessary theoretical background. Chapter 3 describes the relevant hardware and software parts of

the test environment. Chapter 4 gives a detailed description of the three test suites, and summarizes the insights that were gained based on the gathered measurement results. Chapter 5 concludes the thesis with an illustration of the long-term behavior of the test results and an interpretation of the overall real-time capabilities of the ROS communication middleware.

2 General Information

This chapter gives an overview of the theoretical background for this bachelor's thesis, including real-time, ROS, and real-time capable Linux.

2.1 Real-Time

A system is called real-time critical if its work has to be completed before a certain deadline is reached. Depending on the consequences of exceeding a deadline, real-time systems can be classified into three categories [3, p. 321]:

Hard The deadline of a hard real-time system must not be exceeded under any circumstances as this leads to a total system failure.
Example: a vehicle's braking system. If it takes the brakes too long to respond, the vehicle might crash.

Firm The result of a firm real-time system is useless and can be deleted once the deadline is exceeded, but instead of inducing a total system failure it only lowers the systems overall quality of service.
Example: voice over IP. Here, if a sound information package does not reach the recipient in time, this will result in a short loss of sound for several milliseconds. Once that has happened, the corresponding sound information package becomes useless and can be destroyed as soon as it arrives. However, as long as this does not occur too frequently, this is hardly even noticeable and therefore does not implicate a total system failure, but only a decrease in the overall sound quality.

Soft In a soft real-time system, a result can still be used after the deadline was exceeded; the quality of the result is however reduced. Here, the deadline can also be interpreted as a guideline.
Example: video streaming. If video data reaches the user too slow, he will be confronted with a loading screen while watching the video, which implies a decrease in quality. However, once the missing data becomes available it can still be used and the playback of the video can continue.

Furthermore, it is important to understand the difference between real-time and performance. The single goal of a performance-optimized system is to get the average response time as fast as possible. Rare peak values are acceptable, as long as they occur rarely enough to not influence the system's average performance.

Real-time systems accomplish a different objective. Here, not the average response time is of importance, but the occurring peak values. As mentioned earlier, in a system with hard real-time requirements a single peak value that exceeds the specified deadline causes a total system failure. Therefore, the important characteristic of a real-time system is not its average performance but a deterministic behavior. Even if a certain task is on average accomplished faster by a high-performance system without real-time properties, its lack of deterministic behavior makes it impossible to guarantee a certain maximum response time.

2.2 ROS Communication Middleware

The *Robot Operating System* (ROS) originates from different research projects at Stanford University and is now developed as an open source project [4]. Despite the term "operating system" in its name, ROS is actually a software framework implemented on top of conventional operating systems such as Linux or Windows. It consists of a variety of tools and libraries designed for the development of robot software. Therefore, it has two functionally equivalent implementations for use with Python and C++. This thesis however only refers to the implementation in C++.

The following two components of the ROS framework are subject to this thesis:

Timer The ROS library offers a timer function that can be either used in *cyclic* or in *one-shot* mode [5]. On initialization of a timer, one of the two modes has to be specified together with a function pointer and a timeout/frequency value. After the corresponding amount of time has passed, the specified callback function is called. In cyclic mode, this is repeated until the timer is stopped. Furthermore, the callback function is provided with latency information, which is passed as an argument. This latency information tells the function whether it was called right on time, or if the call happened too early or too late.

Middleware The middleware is a key component of the ROS framework. It allows different ROS processes, also referred to as *nodes*, to interact and exchange information in the form of messages [6]. The message exchange follows the publish-subscribe messaging pattern. This means that there are two kinds of participants involved in the message transmission process: *publishers* and *subscribers*. A publisher publishes messages of a certain type on a corresponding topic; a subscriber that has a subscription on said topic will then receive those messages. A topic is defined only by its name and the message type. This means that on a certain topic, only one type of messages can be broadcast. Furthermore, the participating nodes don't know of the existence of any other nodes. It is therefore possible to subscribe a topic that no messages are published on, as well as to publish messages on a topic that no node has a subscription on. Also, the number of publishers or subscribers on a certain

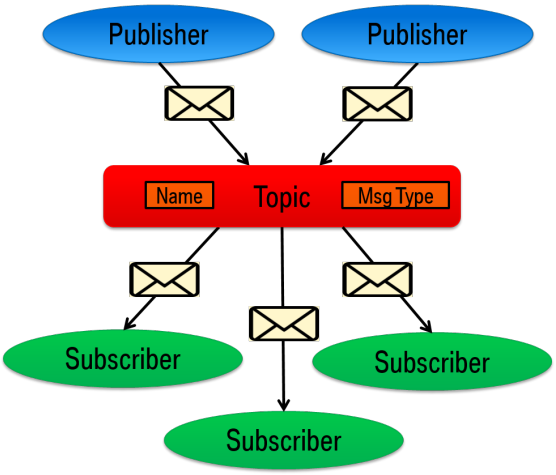


Figure 2.1: ROS Publish-Subscribe Mechanism

topic is not limited and a single ROS node can have multiple subscriptions on different topics or publish messages on different topics or both. See figure 2.1 for an illustration of the described pattern.

2.3 Real-Time Capable Linux

Linux is a widely spread operating system, implemented for a number of different target platforms. In its default version, the mainline Linux kernel however only allows a process to preempt currently executing code if certain conditions are met. This can lead to a scenario where a high-priority process demanding for CPU time has to wait for several milliseconds, e.g., for kernel code to complete execution, until the scheduler actually grants its request. While this is acceptable in most environments, the hereby introduced latency can lead to a total system failure in a real-time critical setup. To prevent this from happening, the CONFIG_PREEMPT_RT patch can be applied to the source code of the mainline Linux kernel, which makes the Linux kernel almost completely preemptible [13]. According to the developers, this reduces the maximum latencies to a scope of only several microseconds.

2.3.1 Cyclictest

Cyclictest is a test suite, developed by Thomas Gleixner, to determine high resolution latencies of the Linux *nanosleep* system call [14]. Its measurement process can be sketched as follows:

1. Switch to real-time priority and FIFO scheduling.
2. Acquire a timestamp *start* and store it.
3. Trigger the nanosleep system call with a certain *timeout* value.

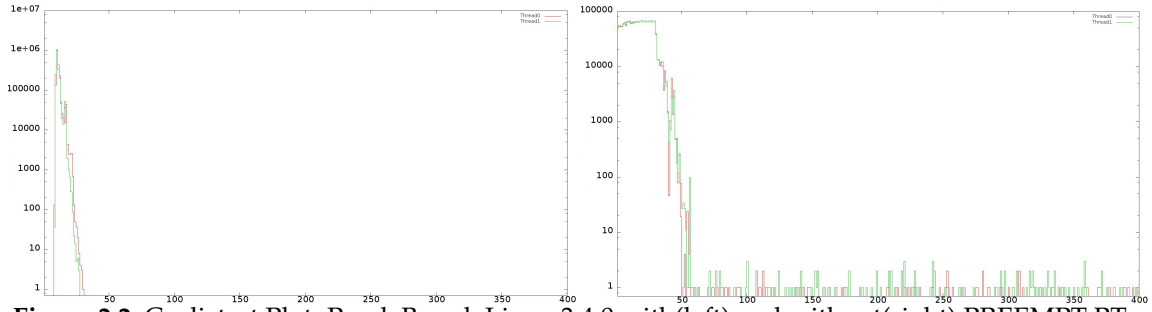


Figure 2.2: Cyclictest Plot: PandaBoard; Linux 3.4.0 with(left) and without(right) PREEMPT RT

4. Acquire a second timestamp *end* and store it.
5. Calculate the timer latency as follows: $latency = end - timeout - start$.
6. Repeat steps 2 - 5 for a specified number of times.

The measured latencies are then evaluated and minimum, average, and maximum values are displayed. See figure 2.2 for an example plot of Cyclictest measurement results on a Linux system with and without full kernel preemption. All of the measured latencies on the PREEMPT RT Linux system (left) were within a range of 8 to 30 microseconds; on the normal Linux system (right) on the other hand, peak values of up to 15 milliseconds occurred. Besides the two different flavors of the Linux operating system, the conditions for the two test runs were exactly the same.

2.3.2 High-Precision Timestamps

For measurements as described in the previous section, a program has to acquire timestamps with very high precision. On Linux systems, this can be accomplished with the *clock_gettime* system call:

```
1 clock_gettime(clockid_t clockid, struct timespec* timestamp);
```

The first argument specifies the clock that shall be used. Here, CLOCK_MONOTONIC_RAW should be specified as it gives access to bare hardware time, which cannot be manipulated by any other program. Depending on the system, those timestamps can reach a precision of up to 1 nanosecond. For further information see the Linux manual page of the clock_gettime system call [2].

2.3.3 Real-Time Processes

To tell the Linux operating system that a certain process is real-time critical, the process priority has to be adjusted and a real-time scheduling policy has to be specified. Both can be done with the *sched_setscheduler* system call [2]:

```
1 sched_setscheduler(pid_t , int policy, const struct sched_param*);
2 //struct sched_param {int sched_priority;};
```

However, there are a few parameters that have to be taken into account when developing a real-time program on Linux:

Multithreading Linux treats threads similar to processes. Therefore, all sub-threads of a single process can have different priorities or scheduling policies. To ensure that all threads run under the same conditions, the *sched_setscheduler* system call should be conducted right at the beginning of the program, before any additional threads are initialized. Future threads will then by default inherit priority and scheduling policy of the calling process unless explicitly specified otherwise (more details on that can be found in the Linux library function manual pages of pthread_create and pthread_attr_setinheritsched [1]). Disregarding this fact can in a worst-case scenario lead to a deadlock with 100% CPU workload.

Scheduling Policy The mainline Linux kernel implements two real-time scheduling policies: *FIFO* and *Round-Robin* (RR). The decision for one of those two has to be based on the design of the corresponding application.
Note: Since Linux kernel version 3.14, *Earliest Deadline First* (EDF) scheduling is also available for use.

Priority Value The decision of the scheduler, which process to give CPU time next, is based on the priority value. The process with the highest RT priority will always get CPU time first. Therefore, the priority value has to be chosen depending on the priority values of other applications. Possible priority values for real-time processes are in the range of 1 (lowest) to 99 (highest). It is however discouraged to use a priority value of 99, as this priority is used by management processes such as watchdog threads [15].

Page Faults Page faults that require I/O activity to be handled can cause significantly increased latencies. Therefore, real-time critical programs should ensure that their virtual address space is locked to physical RAM at all times. This can be done by calling the *mlockall* system call [2] in the program's initialization process:

```
1 mlockall(MCL_CURRENT|MCL_FUTURE);
```

This ensures that the complete current address space of the process is locked to physical memory, as well as any address space that the process might allocate in the future.

For further information on creating real-time processes on Linux, visit the real-time Linux Wiki [15].

3 Test Environment

This chapter describes the relevant hardware and software components of the test environment.

3.1 Hardware Platform

The hardware platform for the testing system is the PandaBoard ES. It is a single-board computer based on the OMAP4460 processor from Texas Instruments, with 1 GB DDR2 RAM (see figure 3.1 [9]).

Data carrier is a SanDisk Extreme Pro SDHC UHS-I memory card with a capacity of 8 GB. All software components, including the operating system, are installed on it.

The used power supply unit provides 4.0 Ampere with a voltage of 5 Volts, which meets the exact specifications of the platform manufacturer.

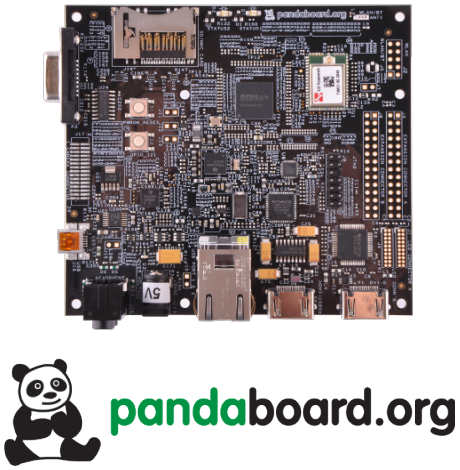


Figure 3.1: PandaBoard ES

3.2 Operating System

ROS is primarily developed for the Linux operating system. Hence, Linux is also employed as operating system for the testing platform. However, instead of using a conventional Linux distribution, the Poky platform builder was used to build a customized one that only contains software components that are mandatory for our purposes.

3.2.1 The Poky Platform Builder

The Poky platform builder is an infrastructure with a collection of tools that can be used to build and cross-compile custom Linux distributions [11]. It is maintained as a part of the Yocto Project, which is an open-source collaboration project focused on embedded Linux developers with members from many different companies, including, e.g., Intel and AMD [12]. The infrastructure it provides is organized through different *layers*; each of them providing different components for the build process of a Linux image. A layer is a collection of so called *build recipes* that provide all the necessary parts for a certain software package, as, for example, the GNU nano text editor, to be included into a Linux image. Alternatively, a layer can also provide support for a specific target platform. The latter type of layer is also referred to as a *Board Support Package* (BSP) layer. By default, the Poky platform builder provides multiple layers that are sufficient for building a minimal Linux image for the QEMU virtual machine.

Besides the default layers, the following two layers were of particular importance to build the Linux image for the testing platform:

meta-ti The meta-ti layer is a BSP layer developed by Texas Instruments [8]. It enables hardware support for a number of target platforms that are based on processors from Texas Instruments, as, for example, the PandaBoard ES. For this purpose, it provides the appropriate machine configurations and Linux kernel build recipes to build a bootable Linux image for any of the supported target platforms. In case of the PandaBoard ES, the meta-ti layer provides a recipe for version 3.4.0 of the Linux kernel.

meta-ros The meta-ros layer was initialized by the BMW Car IT GmbH and is now maintained as an open-source project on GitHub [7]. It provides build recipes for ROS and a continuously growing choice of many popular ROS packages. The currently supported version of ROS is *Hydro Medusa*, which was released in 2013.

3.2.2 Building Steps

The following instructions sketch the build process of the Linux image:

1. Download the Yocto Project, which includes the Poky platform builder, from the official Yocto Project website[1].
2. Switch to the root directory of the Poky platform builder and install the following three layers into it: meta-openembedded, meta-ros, and meta-ti.
3. The *core-image-ros-roscore.bb* build recipe, which is located in the meta-ros layer, can be used to build a GUI-less Linux image that only contains the core ROS compo-

[1] https://www.yoctoproject.org/downloads

nents. Any components that shall additionally be included in the final image can be specified here. In this case, the following components were added:

- *htop*, for inspection purposes.
- *rt-tests*, which adds the Cyclictest test suite (see section 2.3.1).
- *oneshot-timer-tests*, *cyclic-timer-tests*, and *communication-tests*, which adds the three test suites that are described in the following chapter.

4. Invoke the build process with *bitbake core-image-ros-roscore*, which builds the complete Linux image and stores it in the *deploy* directory.
5. Download the source code of the Linux kernel from the location specified in the meta-ti layer and apply the corresponding CONFIG_PREEMPT_RT patch (see section 2.3).
6. Manually rebuild the Linux kernel with a kernel configuration that has full preemption support enabled and use the resulting kernel to replace the one in the deploy directory of the Poky platform builder.
7. Install the image in the deploy directory of the Poky platform builder to the memory card.

The exact versions of all relevant software components that were involved in the build process are listed in table 3.2.1.

For detailed step-by-step instructions of this process, visit the documentation of the ROS real-time tests GitHub project under the following URL[2].

Software	Version
Yocto Project	1.5.1
Poky Platform Builder	10.0.1 “Dora”
meta-ros	SHA1 ID “f6ce8dddab4e8c7bedf5adb31c541adbc3941f94”
meta-openembedded	Branch “dora”, SHA1 ID “40e0f371f3eb1628655c484feac0cebf810737b4”
meta-ti	SHA1 ID “aacecf01794687af7c78cf98a979a3db8b69938a”
kernel-ubuntu	Branch “ti-ubuntu-3.4-1485”, SHA1 ID “b3d5eeb10553e4bc0c3f250a4d06d43c4ab397a9”
ros_realtime_tests	SHA1 ID “a3f87bb3c259a9482a104b7346f38a1126be2821”
gcc [Poky build]	Version 4.8.2 20131212 (Red Hat 4.8.2-7)
gcc [RT kernel build]	Version 4.7.3 (Ubuntu/Linaro 4.7.3-1ubuntu1)
gcc-arm-linux-gnueabi	Version 4:4.7.2-1

Table 3.2.1: Linux Build Software Versions

[2] http://bmwcarit.github.io/ros_realtime_tests/

4 Validation through ROS Real-Time Tests

The validation of the real-time capabilities of ROS focuses on three functionalities of the ROS framework: the one-shot timer, the cyclic timer, and the publish-subscribe communication mechanism (see section 2.2). Hence, three test suites were developed, named oneshot_timer_tests, cyclic_timer_tests, and communication_tests, each of them testing one of the previously mentioned functionalities. By repeatedly executing these test suites in the specified test environment (see chapter 3), with different conditions and with different test parameters, many insights concerning the real-time behavior of the different components could be gathered.

Furthermore, to encourage the use of these test suites, they have been developed with a focus on easy usability within any existing ROS system and are publicly available on GitHub under the BSD 3-clause license. There the test suites can be found in the *ros_realtime_tests* git repository, which is located in the user space of the BMW Car IT GmbH where it can be found under the following URL[1]. Furthermore, to allow for an easy inclusion of the test suites into an embedded ROS system, the meta-ros layer (see section 3.2.1) by default contains all of the necessary build recipes to do so. Therefore, if the Poky platform builder in conjunction with the meta-ros layer is used for the construction of the operating system, any of the test suites can be included by simply adding the corresponding test suite name, underscores replaced through dashes, to the system's build targets (as illustrated in section 3.2.2).

The test suites allow for all of the relevant test parameters to be freely specified on program startup, and on completion of a test run the measurement results are stored in a log file, which can directly be plotted with gnuplot. The resulting plot is then stored as a JPEG image, which has the exact same name as the corresponding log file. These plots are two dimensional and illustrate how often a certain latency or jitter value occurred during the respective test run. On the X-axis are the measured values in microseconds, while the corresponding number on the Y-axis states the frequency with that a certain value occurred. For example, if a value of 100 microseconds on the X-axis has an according frequency of 42 on the Y-axis, this means that the value of 100 microseconds was measured for 42 times during the corresponding test run.

[1] https://github.com/bmwcarit/ros_realtime_tests

Note that while the X-axis has a linear scale, the Y-axis is scaled logarithmically to a base operand of 10. This is done to make infrequent peak values, which might only occur once during a test run, clearly visible in the final plots. See figure 4.1 on page 16 for an example plot of a log file from the oneshot_timer_tests test suite.

In the following sections, each test suite is explained individually and the insights the corresponding test results gave are summarized. Further information concerning the use of the test suites can be found on GitHub[1].

4.1 The One-Shot Timer

The first test suite, named oneshot_timer_tests, that was developed during the course of this project validates the real-time capabilities of the ROS one-shot timer function (see section 2.2). It repeatedly triggers the one-shot timer with a certain timeout value and every time the timer expires, it measures the latency by which the actually specified timeout value was missed. When a one-shot timer expires, it calls a callback function which it provides with information about the delay of the call. This latency value is also stored, together with the latency value measured by the test suite.

The following test parameters can be specified:

Timeout value The timeout value (in microseconds) that the ROS one-shot timer is triggered with during the measurement process.

Number of repetitions The number of repetitions defines the total amount of measurements to be conducted during the test run.

Scheduling policy The scheduling policy with that the test suite is executed. This can be either round-robin or FIFO for a test run as real-time process, or "0" which indicates that the test suite shall run as normal user process with default process priority and scheduling policy.

After a certain number of measurement repetitions, the collected data is analyzed to get the following information, which are then displayed on the shell and stored in the log file.

Measured timer latency The timer latencies measured by the test suite are analyzed for minimum, maximum, and average. Furthermore, the ten highest and the ten lowest measured values are listed, along with the loop index of their occurrence.

Reported timer latency As mentioned above, the ROS timer provides latency information itself. Those are also analyzed for minimum, maximum, and average values.

Difference Theoretically, the above two values should be very similar. To validate this assumption, the differences of every two corresponding latency values, measured and reported, are calculated and analyzed for minimum, maximum, and average.

The output after the completion of a test run looks similar to the following sample:

```
1 root@pandaboard:~# oneshot_timer_tests -s RR
2 INFO: Performing ROS Timer latency measurements...
3 INFO: Running in default mode
4 INFO: Measurement results with a loop length of 1000 and a timeout of 1000
      us:
5 INFO: Measured: MIN: 60us AVG: 101us MAX: 138us
6 INFO: Reported: MIN: 70us AVG: 110us MAX: 147us
7 INFO: Difference: MIN: 8us  AVG: 9us MAX: 29us
8 INFO: Indices of top values(|latency:index|):
      |138:151|128:146|127:69|125:152|124:276|123:692|122:557|121:70|...
```

```
9 INFO: Indices of lowest values(|latency:index|):
      |60:358|64:293|65:757|67:513|67:528|70:301|71:610|71:791|72:366|...
```

4.1.1 Design Choices

For the ROS timer function to become available, the test suite had to be implemented as a ROS node. Therefore, before the measurement process is started, the test suite registers itself within the ROS system. If specified by the user, the process is executed as a real-time process, which is implemented in consideration of the constraints described in section 2.3.3.

The measurement algorithm is implemented similarly to that of the Cyclictest test suite (see section 2.3.1) and can be sketched as follows:

1. Initialize ROS one-shot timer and start an additional thread, executing the *ros::spin()* method.
2. Acquire timestamp *start* and store it.
3. Trigger one-shot timer with the specified timeout value.
4. Wait for the timer to call the callback function.
5. On expiration of the timer:
   a) Take a second timestamp *end* and store it.
   b) Store reported latency value.
6. Calculate the measured timer latency as follows: $latency = end - timeout - start$.
7. Repeat steps 2 - 6 for specified amount of times.
8. Analyze collected data, display results on shell, and create log file.

4.1.2 Test Results

The test suite was run for multiple times in the testing environment specified in chapter 3, with alternating test parameters and under different conditions. The results of these test runs led to the following insights:

No timeout values underneath one millisecond The online documentation of the ROS timer function does not specify any boundaries for timeout values. However, for any timeout value below one millisecond the timer still behaves as if one millisecond was specified. This becomes evident if the test suite is executed with a timeout value of , for example, 800 microseconds. In this case, the latency values increase by 200 microseconds compared to a test run with a timeout value of one millisecond. So, for timeouts shorter than one millisecond the ROS one-shot timer cannot be used.

Timers Randomly ignored without RT priority In some cases, it can occur that a triggered one-shot timer never calls the callback function and seems to be completely ignored. This however so far only occurred when the test suite was running as a normal user process with the respective process priority and scheduling policy. With real-time priority and scheduling policy, this behavior could not be witnessed so far. The ROS community is already informed about this and it is further investigated when this occurs.

Reported latency values fairly accurate In most cases, the reported latency values that are passed to the callback function are quiet accurate. The average difference between reported and measured latencies never exceeded 10 microseconds, and the highest measured peak difference was below 80 microseconds. However, this is only the case if the test suite runs with real-time priority and scheduling. Otherwise, reported and measured latency values drift apart by up to multiple milliseconds.

400 microseconds deadline only exceeded rarely If the test suite is running with real-time priority and scheduling policy, timer latencies that exceed a value of 400 microseconds only occur very rarely. The deadline of 400 microseconds was chosen, as this is the deadline used by the developers of the CONFIG_PREEMPT_RT patch (see section 2.3) for Cyclictest test runs on their testing systems [10]. During all the test runs that were performed during this project, which sum up to approximately 100 million measurement repetitions, this only occurred one time (with a latency value of 525 microseconds). This leads to a probability for a deadline miss of approximately 10^{-7} - 10^{-8}.

Timer latency correlates with timeout value Depending on the specified timeout value, the average timer latency changes perceptibly. When the test suite is, for example, running with real-time priority, round-robin scheduling, and a timeout value of one millisecond, the average measured latency is around 100 microseconds. If the timeout value is changed to 10 milliseconds, the average measured latency increases to approximately 150 microseconds. A similar behavior can also be witnessed if the test suit runs with default priority and scheduling policy.

Peak values randomly distributed Maximum as well as minimum peak values seem to be randomly distributed over the whole test run. Comparing the loop indices of peak values from different test runs does not indicate any pattern.

Overall, the measured latencies show a quite deterministic behavior. The results of multiple test runs with the same parameters always look nearly exactly the same, and significant peak values almost never occur. See figure 4.1 for an example plot of a test run with a timeout value of 1 millisecond, real-time priority, and round-robin scheduling. In this case, all measured latencies were within a range of 55 microseconds to 177 microseconds, with an average latency of 98 microseconds. However, if we compare this plot with

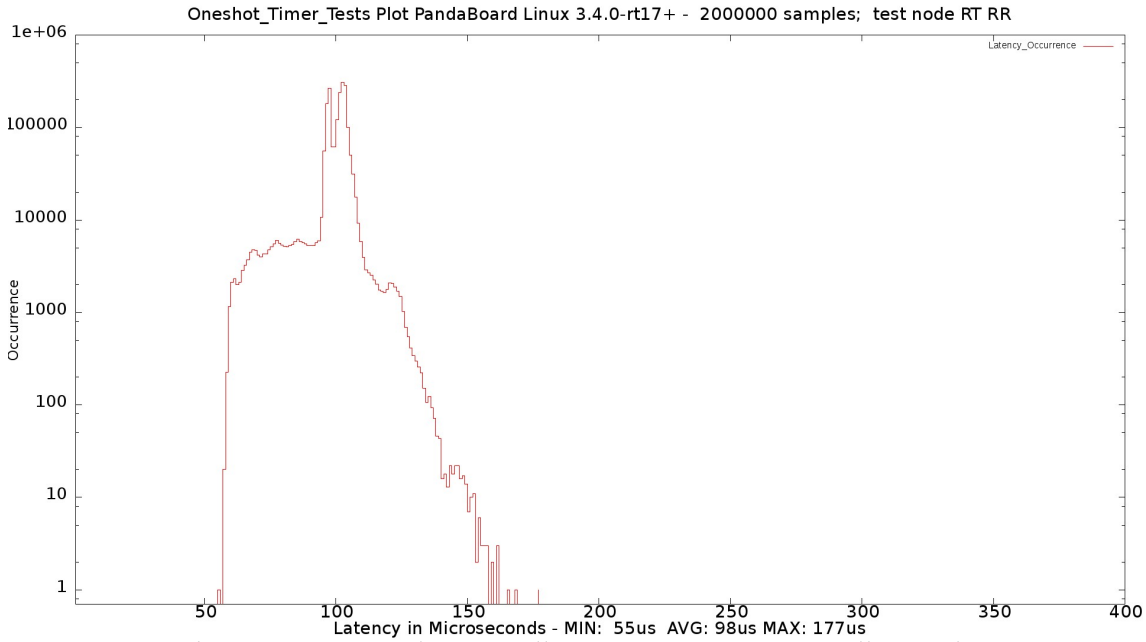


Figure 4.1: Oneshot_Timer_Tests Plot: 2 million Measurements; 1 Millisecond Timeout; RT-Priority; Round-robin Scheduling.

the left plot in figure 2.2 on page 6, which shows the Cyclictest (see section 2.3.1) results of a similar scenario, it is obvious that the nanosleep system call shows a much better real-time behavior than the ROS one-shot timer, as it can meet a much lower deadline and during approximately 40 million measurement repetitions on the testing system, a latency of 50 microseconds wasn't exceeded even once. This indicates that there is still a lot of potential for possible future improvements of the implementation of the ROS one-shot timer to minimize the occurring latencies.

An overall summary of the long-term behavior of the one-shot timer will be depicted in the conclusion (chapter 5).

4.2 The Cyclic Timer

As the name suggests, the cyclic_timer_tests test suite validates the real-time capabilities of the cyclic ROS timer function (see section 2.2). Therefore, it triggers the cyclic timer with a certain frequency and measures the time that passes between every two consecutive calls to the callback function. After the callback function was invoked for a specified amount of times, the measured time intervals are analyzed. Theoretically, the time that passes between every two consecutive callback calls solely depends on the specified frequency and should therefore always be the same. For a frequency of, e.g., 100 hertz, this time interval should be 10 milliseconds. The test suite calculates the jitter of the actual time interval between every two callback function calls. Therefore, it calculates the differences between the measured time intervals and the actually expected value, as derived from the selected frequency.

The following test parameters can be specified:

Frequency The frequency (in hertz) to which the cyclic ROS timer is set during the measurement process.

Number of repetitions The number of repetitions defines the total amount of measurements to be conducted during the test run.

Scheduling policy The scheduling policy with that the test suite is executed. This can be either round-robin or FIFO for a test run as real-time process, or "0" which indicates that the test suite shall run as normal user process with default process priority and scheduling policy.

After a test run, minimum, maximum, and average of the collected data are determined, as well as the loop indices of the ten highest and lowest marginal values. These information are then printed to the shell and stored in the log file.

The output after the completion of a test run looks similar to the following sample:

```
1 root@pandaboard:~# cyclic_timer_tests -s RR
2 INFO: Performing latency measurements...
3 INFO: Measurement results with a total of 1000 measurements and a frequency
      of 1000Hz:
4 INFO: Jitter: MIN: -961us AVG: 0us MAX: 170us
5 INFO: Indices of top values(|latency:index|):
     |170:101|150:690|145:131|141:204|140:241|138:351|138:791|138:901|...
6 INFO: Indices of lowest values(|latency:index|):
     |-961:138|-960:203|-958:532|-958:560|-958:689|-958:835|-958:853|...
```

4.2.1 Design Choices

Similar to the ROS one-shot timer (see section 4.1), the cyclic timer function of ROS is only accessible for ROS nodes. Hence, the test suite is implemented as such and registers

itself within the ROS system before the measurement process is started. Also, similar to the implementation of the oneshot_timer_tests test suite, the user can choose to run the test suite as real-time process. This is implemented in consideration of the constraints described in section 2.3.3.

The measurement process can be sketched as follows:

1. Initialize cyclic ROS timer instance.
2. Trigger timer with specified frequency.
3. Execute *ros::spin()* function.
4. On every invocation of the callback function, store current timestamp.
5. After a certain amount of callback function calls, stop measurements.
6. Calculate the frequency jitter between every two consecutive calls of the callback function. This is done as follows:
   $jitter[i] = timestamp[i] - timestamp[i-1] - \frac{1}{frequency}$
7. Analyze collected data, display results on shell, and create log file.

4.2.2 Test Results

Multiple executions of the test suite on the testing system (see chapter 3) gave the following insights:

Cluster of negative jitter values around -900 microseconds Besides positive jitter values, the test results also show a significant amount of negative jitter values (about 12% of the total) around approximately -900 microseconds. In the plots, the cluster of negative jitter values looks similar to the cluster of positive values, but with an offset on the X-axis by -1 millisecond and a compression in height and width (see figure 4.2). The offset of -1 millisecond is thereby independent of the selected frequency.

Significant negative peak values in first two measurements Some frequencies induce negative peak values of up to multiple milliseconds. These peak values however only occur in the first two measurements. A frequency inducing this behavior is, for example, 100 Hz, with the first and second jitter value being around -9.9 and -4 milliseconds. Otherwise, the occurrence of peak values doesn't suggest a pattern as they are randomly distributed throughout the runtime of the test suite.

Overall frequency kept Besides maximum and minimum jitter values, the (signed) average jitter value was always 0. This means that overall the amount of calls to the callback function during a certain period of time is what we would expect it to be. For example, with a frequency of 10 Hz and a runtime of 1 second, the callback function is actually called for a total of 10 times.

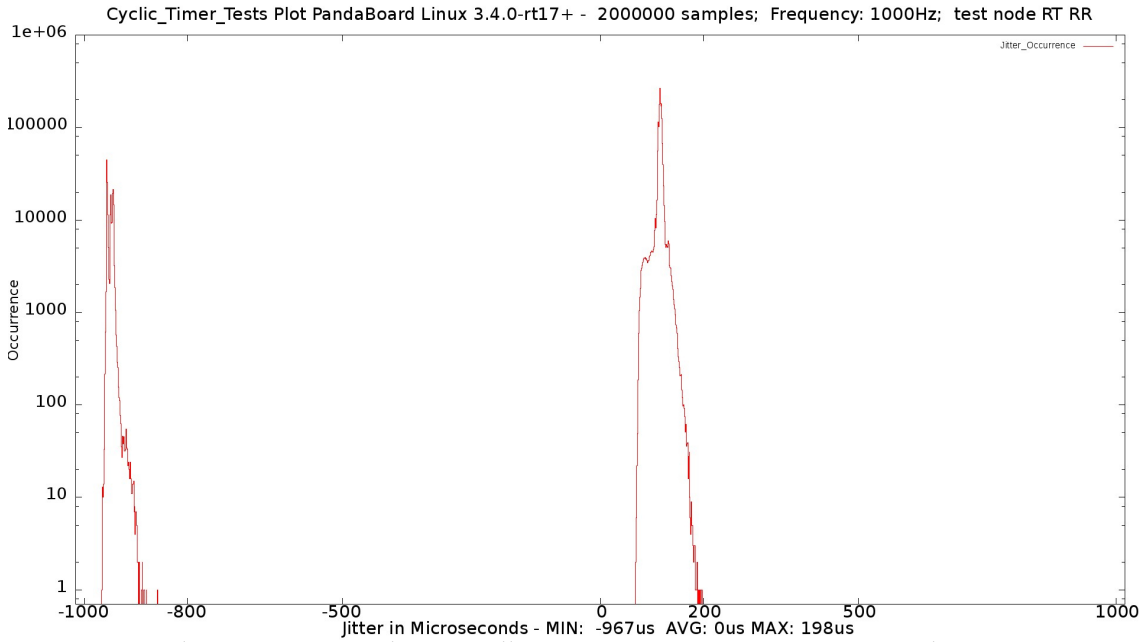


Figure 4.2: Cyclic_Timer_Tests Plot: 2 million Measurements; Frequency: 1 kHz; RT-Priority; Round-robin Scheduling.

Positive jitter values similar to one-shot timer latencies Positive jitter values occur when the delay between two callback function calls is longer than expected. The measured positive jitter values show a behavior similar to the latencies of the one-shot timer (compare figure 4.2 and figure 4.1) concerning the following two points, which are explained in more detail in section 4.1.2:

- Correlation of specified timeout value (derived from the frequency) with measured latency values
- Random distribution of peak values throughout the test run

400 microseconds deadline never exceeded so far During all the measurements that have been conducted so far, a deadline of 400 microseconds has never been exceeded.

Significantly increasing jitter for frequencies over 1 kHz If a frequency of more than 1 kilohertz is specified, the measured jitter values, both negative and positive, start to increase significantly. Therefore, the cyclic ROS timer should not be used with frequencies over 1 kHz.

As mentioned above, the percentage of negative jitter values is about 12%, which results in a ratio of negative to positive jitter values of approximately 1 to 7. This, and the fact that nearly all negative jitter values are between -1000 and -600 microseconds suggests that the cyclic ROS timer is implemented on top of the one-shot timer and simply reschedules it with a timeout value derived from the frequency. Once the overall delay of callback function calls (caused by the latencies of the one-shot timer) adds up to a value of about one millisecond, the next one-shot timer is simply rescheduled with a one millisecond

shorter timeout value, to keep the overall frequency to the specified value. With a frequency of 1 kHz, this results in about every 8^{th}callback function call to take place almost immediately after the previous one (sometimes time intervals of under 50 microseconds while it should be 1000). However, this ensures that overall the specified frequency is kept. But, due to this behavior the cyclic ROS timer, in its current state, should not be used in real-time critical systems that operate at a high frequency of around 1 kHz.

Overall, the behavior of the cyclic ROS timer is quiet deterministic. So far, no jitter values outside of the two described clusters at around -900 and +100 microseconds were measured, besides the two peak values in the first two measurements that occur with certain frequencies. However, an ideal implementation should achieve only one cluster of jitter values with the center at 0 microseconds.

In the conclusion (chapter 5), the long-term behavior of the cyclic ROS timer gets further illustrated and interpreted.

4.3 The Publish-Subscribe Communication Mechanism

The real-time capabilities of the publish-subscribe communication mechanism of ROS (see section 2.2), are validated by the communication_tests test suite. For this purpose, the end-to-end message transmission latencies, which arise from the publication of a message until its reception by an appropriate subscriber node, are measured.

The following test parameters can be specified:

Start delay Amount of seconds to wait after the initialization, before starting the measurement process.

Publication frequency Frequency in hertz with that the messages are published during the measurement process.

Message payload length Size (in Bytes) of the additional message payload that is appended to each test message (if a value >0 is specified).

Number of repetitions Total amount of messages to send throughout the measurement process.

Scheduling policy The scheduling policy with that the test suite is executed. This can be either round-robin or FIFO for a test run as real-time process, or "0" which indicates that the test suite shall run as normal user process with default process priority and scheduling policy.

The measured latencies are analyzed for minimum, maximum, and average, as well as for the loop indices of the top ten highest and lowest peak values. Furthermore, the amount of messages that didn't reach the subscriber is counted. This is also done for the amount of messages that arrived out of order. This information is then printed to the shell and stored in the log file.

The output after the completion of a test run looks similar to the following sample:

```
1 root@pandaboard:~# communication_tests_subscriber -s RR &
2 root@pandaboard:~# communication_tests_publisher -d 3 -s RR
3 INFO: Done publishing...
4 INFO: Measurement results:
5 INFO: Amount messages: 1000; Messages out of order: 0
6 INFO: MIN: 200us AVG: 205us MAX: 517us
7 INFO: Indices of top values(|latency:index|):
      |517:0|242:97|241:10|229:1|226:463|222:863|220:16|219:504|218:35|...
8 INFO: Indices of lowest values(|latency:index|):
      |200:241|200:409|201:7|201:85|201:86|201:87|201:88|201:94|201:96|...
```

4.3.1 Design Choices

To measure the message transmission latencies, the test suite has to exchange messages under a certain topic. Therefore, it consists of a publisher and a subscriber ROS node.

Both nodes can be executed with RT-priority and scheduling policy, which is implemented as specified in section 2.3.3.

Publisher node The publisher node works as follows:

1. On startup:
   a) Adjust priority and scheduling policy.
   b) Register within the ROS system.
2. If a start delay is specified, sleep for the according amount of seconds.
3. Acquire current timestamp and publish it in a message under the "*communication_tests*" topic.
4. Wait for a certain amount of time, derived from the specified frequency.
5. Repeat steps 3 - 4 for the specified amount of times.
6. Call *ros::shutdown()* to ensure that all messages are transmitted before the publisher process dies.

Subscriber node The subscriber node is responsible for measuring the actual transmission latency. It is implemented as follows:

1. On startup:
   a) Adjust priority and scheduling policy.
   b) Register within the ROS system and subscribe the *"communication_tests"* topic.
2. For every message that is received, calculate the corresponding transmission latency and store it.
3. After receiving the last message, analyze the measured data, display the results on the shell, and create the log file.

The messages that are exchanged during the measurement process have the following structure:

```
1 uint32 sec
2 uint32 nsec
3 uint32 seq
4 bool last_msg
5 uint8[] payload
```

The first two 32 bit integer values contain a *publication* timestamp that the publisher node acquires right before publishing the message. This enables the subscriber node to calculate the transmission latency by simply taking a timestamp *reception* as soon as it receives

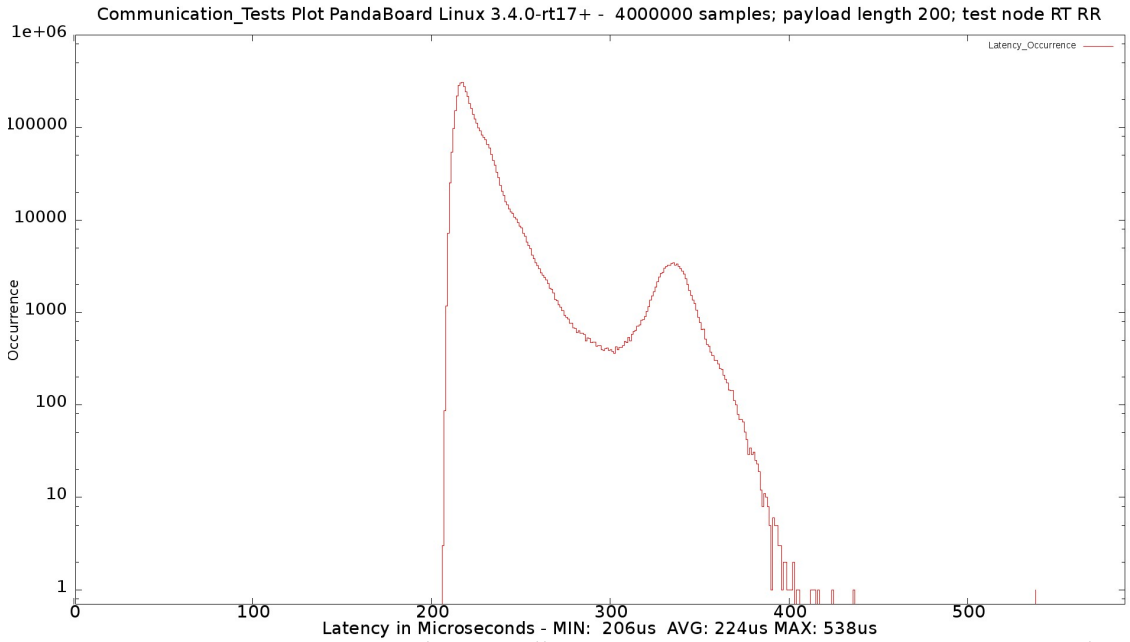


Figure 4.3: Communication_Tests Plot: 4 million Measurements; Frequency: 100 Hz; Payload Length: 200 Bytes; RT-Priority; Round-robin Scheduling; Only 1 Subscriber Node.

the message. With those two timestamps, the transmission latency is calculated as follows: $latency = reception - publication$. The payload array of 8 bit integer values is only attached if a payload length bigger than zero is specified on startup.

Furthermore, the design of the test suite allows for an unlimited amount of subscriber nodes to run concurrently. If multiple subscriber nodes are used during a test run, each of them will independently measure the occurring transmission latencies and generate a respective log file. By default, the test suite supports the automatic start of 1 or 5 subscriber nodes.

4.3.2 Test Results

Multiple executions of the test suite on the testing system (see chapter 3) gave the following insights:

High transmission latencies for first message The first transmitted message usually has the highest transmission latency. In figure 4.3 for example, the maximum latency of 538 microseconds was caused by the first message. In this example, all other measured transmission latencies were below 437 microseconds. A possible explanation for this might be that the actual connection between publisher and subscriber is first established when a message gets actually published. The thereby introduced overhead might cause the increased latency.

Otherwise randomly distributed peak values All minimum and maximum transmission latencies, besides that of the first message, are randomly distributed through-

out the complete runtime of the test suite. Comparing the indices of the peak values from multiple test runs does not show any pattern.

Increased amount of RT subscriber nodes broadens range of measured latencies Increasing the amount of real-time nodes with a subscription on the same topic results in a broader range of occurring transmission latencies. With only one real-time subscriber, all measured latencies, besides that of the first message, usually were within a range of approximately 200 microseconds in width (see figure 4.3). When the amount of subscriber nodes was increased to 5, this range however broadened significantly. See figure 4.4 for the measurement results from a test run where a total of 5 subscriber nodes were running at the same time (with otherwise the exact same test parameters as before). In this case, the range of occurring transmission latencies has a width of approximately 500 microseconds. This behavior might be caused by the fact that once a message is published on a topic with 5 equally prioritized subscriber nodes, all of them request CPU time at the same moment, while the testing system only provides two CPU cores. Therefore, some of the subscriber nodes have to wait for others to finish first, which would explain the increased latencies.

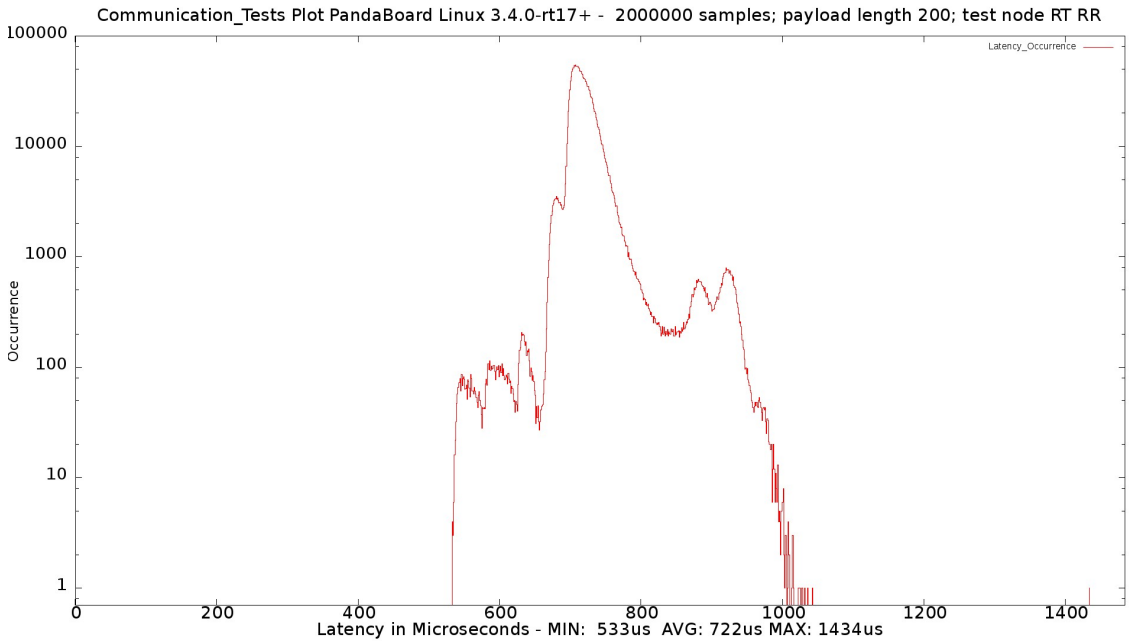


Figure 4.4: Communication_Tests Plot: 2 million Measurements; Frequency: 100 Hz; Payload Length: 200 Bytes; RT-Priority; Round-robin Scheduling; 5 Subscriber Nodes.

Increase in frequency improves measured latencies An increased frequency results in lower transmission latencies. In the example in figure 4.3, where the test suite was run with a frequency of 100 hertz, the measured latencies were in the range of 206 microseconds to 436 microseconds with an average transmission latency of 224 microseconds. A respective test run with a frequency of 1 kilohertz resulted in latency values within the range of 193 microseconds to 340 microseconds and an

average transmission latency of only 203 microseconds. Note that the transmission latencies of the first messages were not taken into account for this statistics.

Overall, as will be further depicted in chapter 5, the transmission latencies show a deterministic behavior over multiple test repetitions. So far, no unproportionally high latencies have occurred during any of the conducted test runs. However, there is so far no insight on the influence that other, independent ROS nodes (communicating on different topics at the same time) might have on the transmission latencies of the real-time critical ROS nodes. For that purpose, further test runs within pre-existing ROS systems have to be conducted.

4.4 General Insights

Even though the three test suites all have their focus on different functionalities of ROS, the corresponding test results did not only lead to insights concerning the respective components, but also to insights that apply to PREEMPT_RT Linux and ROS in general. The following are the most important ones:

Same behavior with round-robin and FIFO scheduling Running the test suites with round-robin or FIFO scheduling does not have any impact on the measurement results. This is probably due to the time slice length of the round-robin scheduling policy on the testing system. As long as the test suites never execute longer than the specified time slice before going to sleep again, the scheduler never has to interfere and therefore the specified scheduling policy doesn't influence the test results.
The value of the round-robin time slice length can be found out with the *sched_rr_get_interval* system call:

```
1 int sched_rr_get_interval(int pid, structu timespec* tp);
```

On the testing system, the value this system call returns for a real-time process is 100 milliseconds.

RT ROS node on single CPU platform can cause stuck system Running the test suites with real-time priority and scheduling policy on platforms with single-core CPUs, revealed a problem with the *ros::shutdown* call. If ros::shutdown is called in such a scenario, it will get stuck in a busy loop, waiting for a response from the ROS master process. This happens due to the fact that the node process has a higher priority than the ROS master process. The busy loop in the ros::shutdown call hence never gets preempted by the ROS master process, which can therefore never send the answer that the node process is waiting for. Currently, the only option to circumvent real-time ROS nodes from getting stuck in the shutdown procedure is to execute the ROS master process with at least the same priority as the highest prioritized ROS node. If FIFO scheduling is used, the priority of the ROS master process even has to be higher than that of all real-time ROS nodes. This is the only way to ensure that the ROS master process gets the necessary CPU time to answer the nodes.
Besides running the ROS master process with real-time priority and scheduling policy, the current state of ROS has no possibility to avoid this problem. It is therefore currently discouraged to execute real-time ROS nodes on single-core platforms. However, the ROS community is already informed about this behavior (see [2] for the discussion). Therefore, it can be expected that the problem will be resolved in the future, allowing the execution of ROS nodes with real-time priority both on single- and multi-core platforms.

[2] https://groups.google.com/d/msg/ros-sig-embedded/5KyHU_zu6_E/h75Z1Z2OqWYJ

Unpredictable behavior without correct RT process setup If the test suites are executed as default user processes (without real-time priority and scheduling policy), the measurement results become rather unpredictable. This shows that a key element for the development of real-time processes is respecting the requirements stated in section 2.3.3. See figure 4.5 for example plots of test runs where the test suites were executed with default priority and scheduling policy. Compared to the plots from the previous chapters, where the test suites were executed with real-time priority and scheduling policy, the measured latency values increase drastically and become a lot less deterministic.

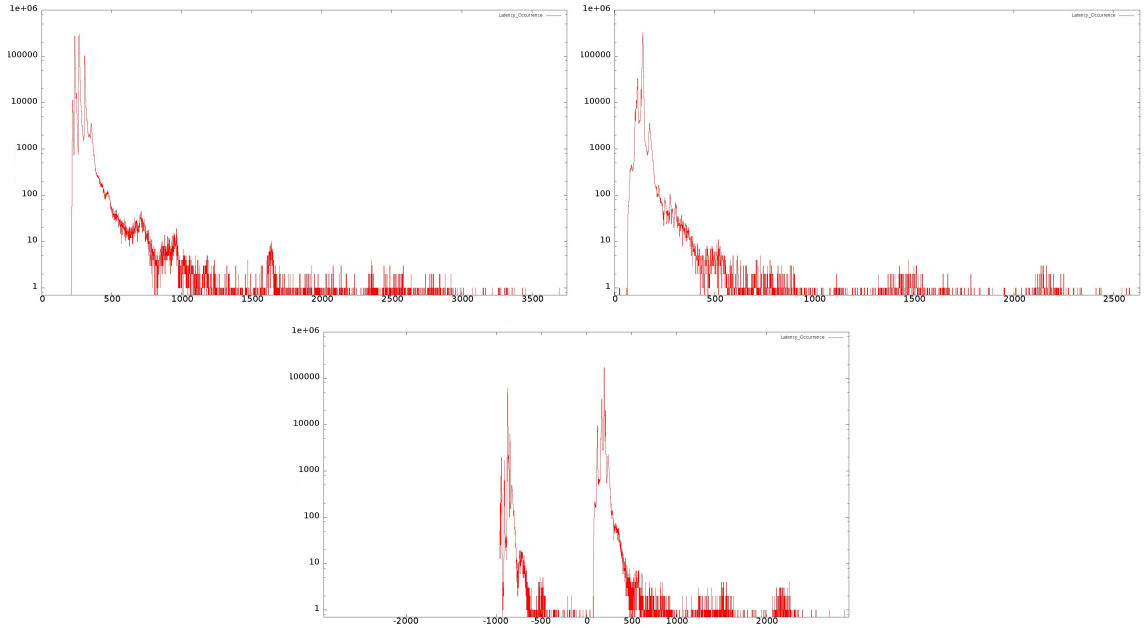


Figure 4.5: Plots: Test Results without Real-Time Priority and Scheduling: Communication_Tests (top left), Oneshot_Timer_Tests (top right), Cyclic_Timer_Tests (bottom).

5 Conclusion

To decide whether ROS can be used for real-time critical tasks, we have to take a look at the following long-term latency plots that illustrate all of the test results that were obtained up to now, with an individual plot for each of the three test suites. All of these test runs were conducted with varying test parameters and under different conditions, for example, with the CPU in an idle state or with an additional workload. However, what all of the illustrated test runs have in common is that the test suite was running on the test environment depicted in chapter 3 and with real-time priority and FIFO or round-robin scheduling policy. The long-term plots are three-dimensional with their X-, Y-, and Z-axes aligned as depicted in figure 5.1. Along the X-Axis are, depending on the test suite, the jitter or latency values in microseconds with a linear scale. The Z-axis depicts the frequency of the corresponding latency or jitter value. If, for example, a latency value of 250 microseconds has a corresponding value of 100 on the Z-axis, it means that the latency value of 250 microseconds was measured 100 times during that test run. The Z-axis is scaled logarithmically with a base value of 10 and a range from -1 ($10^{-1} = 0.1$) to 5 ($10^5 = 100,000$). In direction of the Y-axis this representation is repeated, but with the measurement values from different test runs. Basically, X- and Z- axis of the three-dimensional long-term plots correspond to the X- and Y-axis of the two-dimensional plots that can be generated from the log files of the test suites (see chapter 4). Therefore, we can imagine the long-term plots as many of the two-dimensional plots being placed behind each other in direction of the Y-axis, thereby adding the third dimension.

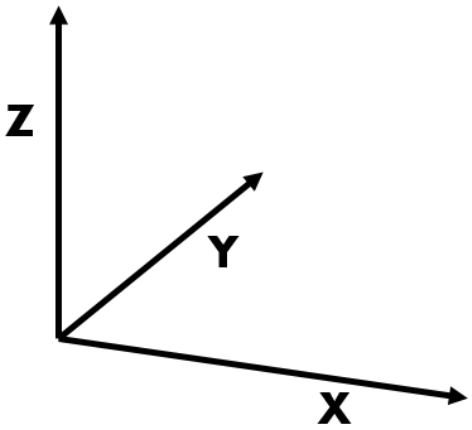


Figure 5.1: Directions X-, Y-, and Z-Axis

The first long-term plot (figure 5.2) is from the oneshot_timer_tests test suite (see section 4.1). It illustrates the test results from 47 test runs with a total of 94 million latency measurements. These test runs were conducted with timeout values of 1 or 10 milliseconds, with FIFO or round-robin scheduling, and with the system in an idle state or with the system being exposed to additional CPU workload. We can see that besides a few peak values above 300 microseconds, the overall distribution of measured values looks quite

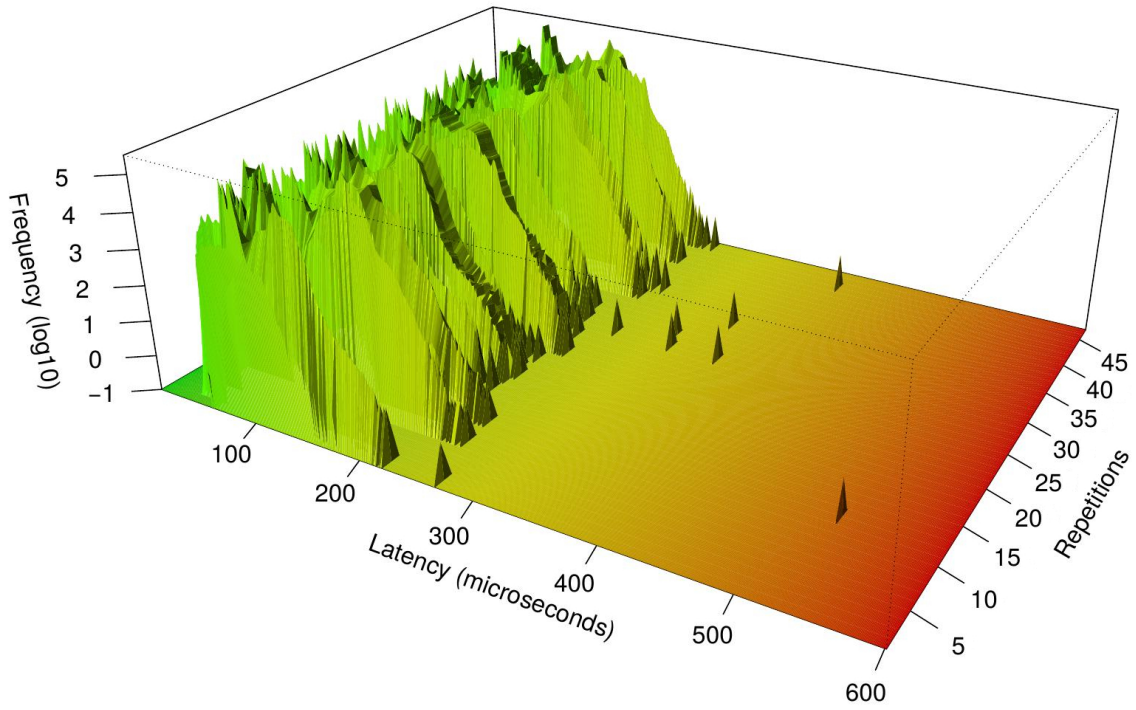


Figure 5.2: Oneshot_Timer_Tests 3D long-term Plot; 47 Test Runs with a Total of 94 million Measurements

deterministic. A value of 400 microseconds was even only exceeded once, by a latency of 525 microseconds. Therefore, based on these measurements the probability for latencies above 400 microseconds can be approximated to a value between 10^{-7} and 10^{-8}.

Overall, the one-shot timer shows a quiet good real-time behavior. However, there are two major issues that should be further investigated:

1. The one-shot timer function can't handle timeout values under 1 millisecond correctly. This should either be resolved in the implementation, or a fixed range of possible timeout values should be specified and noted accordingly in the corresponding documentation.

2. If a ROS node using the one-shot timer is running as a normal user process, it can occur that a triggered one-shot timer doesn't fire and is simply ignored. As mentioned, this doesn't affect nodes that run with real-time priority, however this behavior should still be resolved as it might lead to a faulty behavior of ROS nodes that are running with default priority.

A more detailed explanation of those two issues can be found in section 4.1.

Next, we can have a look at figure 5.3 for the long-term plot of the measurement results from the cyclic_timer_tests test suite (see section 4.2). Here, 16 test runs have been conducted, which sum up to a total of 32 million measurements. For the conducted test runs, the cyclic timer has been triggered with a frequency of 100 Hz or 1 kHz, and, similar to

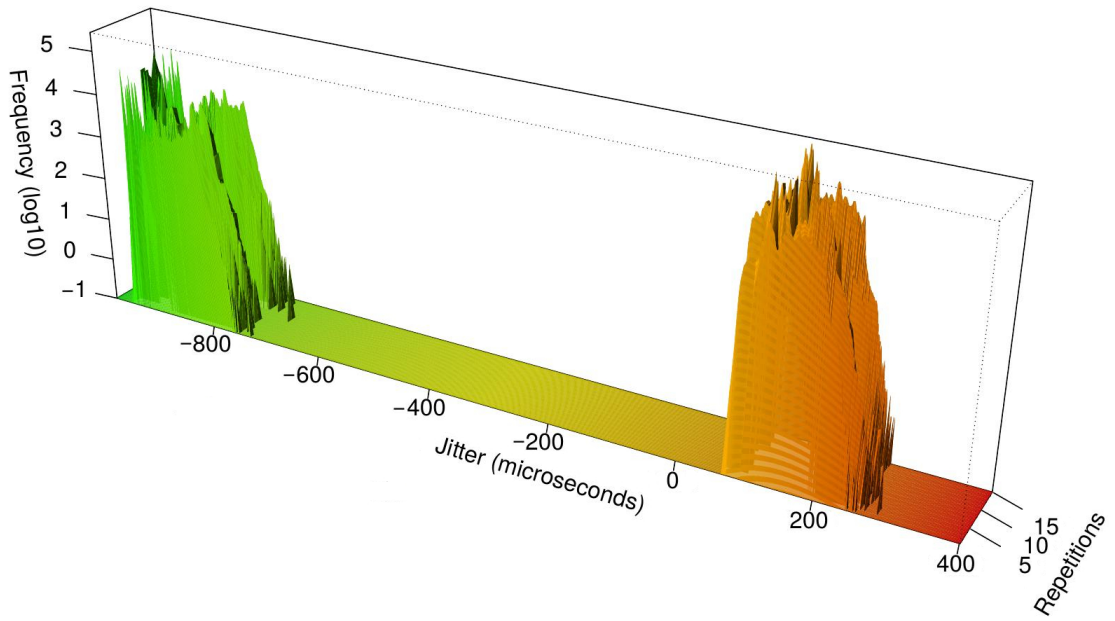


Figure 5.3: Cyclic_Timer_Tests 3D long-term Plot; 16 Test Runs with a Total of 32 million Measurements

the long-term measurements with the oneshot_timer_tests test suite, with the system in an idle state or with the system being subject to additional CPU workload. All measured jitter values are in the range of -1000 microseconds to +400 microseconds. (Note that the large negative jitter values, as described in section 4.2, that can occur in the first two measurements of a test run are not taken into account here.) To be more precise, this rather large range of jitter values can actually be divided into two smaller ranges with a width of 400 microseconds each, from approximately -1000 microseconds to -600 microseconds and from 0 microseconds to +400 microseconds. Even though (due to the logarithmically scaled Z-axis) the amount of measured values in the negative range looks very similar to the amount of values in the positive range, the amount of negative jitter values only accounts for approximately 12% of the total. A possible explanation for this behavior is described in section 4.2.2.

As we can see, independent of the test parameters and the system's condition, the measured values in each of the test runs look very similar and show a very deterministic pattern. Also, the overall specified frequency was always kept, so the total amount of timer expirations within a test run was always as expected. Therefore, in terms of real-time constraints also the cyclic ROS timer function shows a good behavior. It is however very important to take into account that approximately 12 % of timer expirations will happen about 1 millisecond too early. This should be completely acceptable for timer frequencies of around 100 Hz or slower. For a frequency of 1 kHz this however means that 12% of the timer expirations will occur immediately after the previous one. There-

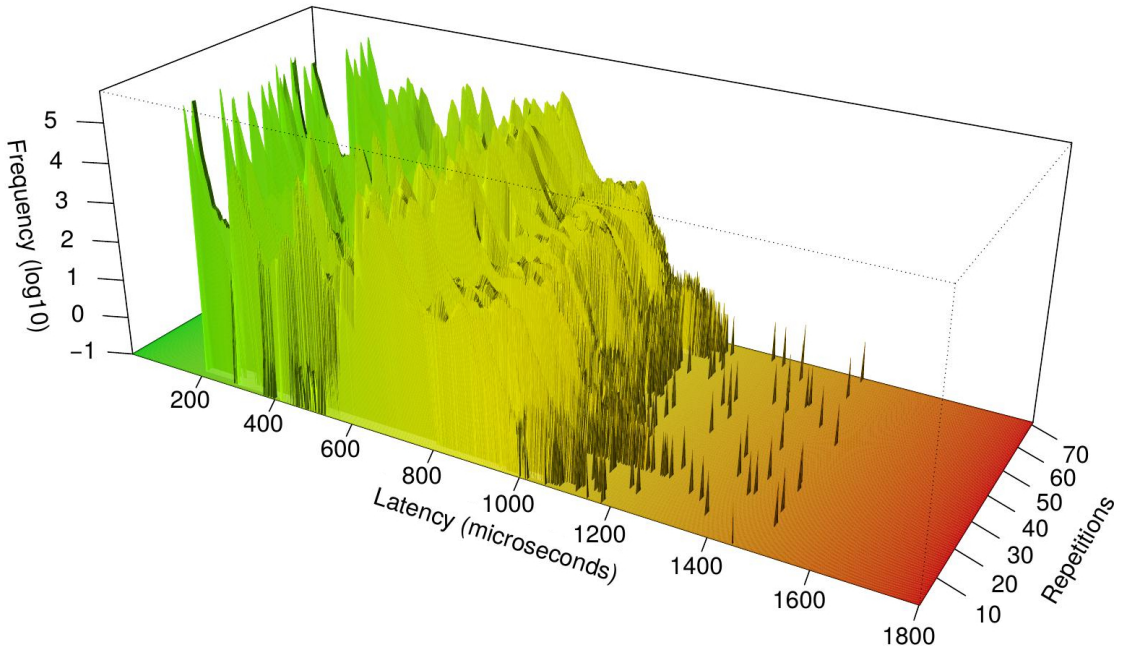


Figure 5.4: Communication_Tests 3D long-term Plot; 71 Test Runs with a Total of 184 million Measurements

fore, this behavior should become a subject for possible future improvements of the ROS implementation.

The third part of ROS that we still have to regard is the publish-subscribe communication mechanism, which was investigated with the communication_tests test suite (see section 4.3). Figure 5.4 shows the corresponding long-term plot, illustrating the results of 71 test runs which collected a total of 184 million latency values of the end-to-end message transmission process. Besides varying the message publication frequency and the system's state in the same manner as previously described for the long-term measurements with the cyclic_timer_tests test suite, also the number of concurrently running real-time subscriber nodes was varied between 5 and only 1. All measured latencies were in a range of approximately 200 microseconds to 1.6 milliseconds.

As we can see in the plot, the distribution of the measured latency values looks quiet similar for all the test runs, which indicates a rather deterministic behavior. Also, in all of the test runs a latency value of 1.6 milliseconds was never exceeded. As already described in section 4.3, the first message of every test run is most times also the one with the highest transmission latency. In 5.4, all latencies above 1.4 milliseconds were caused by the first messages of the corresponding test run. Ignoring the first message from every test run would therefore even leave us with a maximum latency underneath 1.4 milliseconds. With only one subscriber, the maximum measured latencies did not even exceed a value of 500 microseconds.

Overall, the publish-subscribe communication mechanism shows a good real-time behavior, as there were no unproportionally high transmission latencies and the overall

distribution of these measurement values looks rather predictable. It is however important to keep in mind that the first message that a subscriber receives has a rather high transmission latency, and that an increasing number of real-time subscribers on the same topic also induces significantly increasing transmission latencies. Furthermore, it is important for the future to conduct further test runs with the communication_tests test suite, within pre-existing ROS systems to see whether other ROS nodes that run independently from the real-time nodes of the test suite, also using the publish-subscribe communication, have any impact on the measured transmission latencies.

All in all, we can conclude that the overall real-time capabilities of ROS are good, however certain restrictions have to be taken into account, such as the faulty behavior of the one-shot timer for timeout values below 1 millisecond as described earlier. It is also important to note, that the statements made in this thesis can only refer to the exact setup as described in chapter 3. Any discrepancies of any of the described components might induce a completely different behavior. It is therefore important for the future to gather further test results from varying test scenarios on different target platforms. Due to the fact that the test suites are publicly available, anyone who is interested in this topic can contribute by conducting test runs himself and sharing the results with the ROS community. By keeping up this work, the current insights can be further strengthened and broadened, enabling future improvements to the real-time capabilities of the ROS communication middleware.

Bibliography

[1] Linux library function manual pages. http://linux.die.net/man/3/, June 2014.

[2] Linux system call manual pages. http://linux.die.net/man/2/, June 2014.

[3] Heinz Wörn; Uwe Brinkschulte. *Echtzeitsysteme*. 2005.

[4] Open Source Robotics Foundation. Ros history homepage. www.ros.org/history/, July 2014.

[5] Open Source Robotics Foundation. roscpp/overview/timers. http://wiki.ros.org/roscpp/Overview/Timers, July 2014.

[6] Open Source Robotics Foundation. Ros/tutorials/writingpublishersubscriber(c++). http://wiki.ros.org/ROS/Tutorials/WritingPublisherSubscriber%28c%2B%2B%29, July 2014.

[7] BMW Car IT GmbH. Meta-ros github repository. https://github.com/bmwcarit/meta-ros, June 2014.

[8] Texas Instruments. Meta-ti git repository. http://git.yoctoproject.org/cgit/cgit.cgi/meta-ti/, June 2014.

[9] OMAPpedia. Picture pandaboard es. http://www.omappedia.com/images/1/16/PandaBoard_top_view.png, June 2014.

[10] Open Source Automation Development Lab (OSADL). Website: Qa farm realtime. https://www.osadl.org/QA-Farm-Realtime.qa-farm-about.0.html, July 2014.

[11] Yocto Project. Poky platform builder website. http://www.pokylinux.org/, June 2014.

[12] Yocto Project. Yocto project website. https://www.yoctoproject.org/about, June 2014.

[13] Real-Time Linux Wiki. Config preempt rt patch. https://rt.wiki.kernel.org/index.php/CONFIG_PREEMPT_RT_Patch, May 2014.

[14] Real-Time Linux Wiki. Cyclictest. https://rt.wiki.kernel.org/index.php/Cyclictest, July 2014.

[15] Real-Time Linux Wiki. How to build an rt-application. https://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application, May 2014.