AAReST Real-time Software & Test

Vilma Wilke

Year 3 Project

from the

University of Surrey

Department of Electronic Engineering

Faculty of Engineering and Physical Sciences

University of Surrey

Guildford, Surrey, GU2 7XH, UK

May 2016

Supervised by: Dr Christopher Bridges

DECLARATION OF ORIGINALITY

Aarest Real-Time Software & Test

Vilma Wilke

Author Signature Vilma Wilke Date: 17/05/2016

Supervisor's name: Dr. Christopher Bridges

WORD COUNT

Number of Pages:	46
Number of Words:	13124

ABSTRACT

The AAReST mission is an ambitious project undertaken by SSC and CalTech to demonstrate that an array of satellites can be flown and docked in space to construct setups larger than what the launch vehicle diameter would permit. This requires that the many individual components are able to act autonomously to locate each other at a high enough speed to allow accurate docking. In order to improve the speed at which relative positions of other spacecraft are determined, optimisations to the existing software set up are to be undertaken. A series of four tasks were determined to select and optimise an operating system and the existing software, and to perform stress tests on the final software setup to ensure that the changes are as expected.

TABLE OF CONTENTS

LIST OF FIGURES

1 INTRODUCTION

AAReST is a mission intended to demonstrate the viability of assembling an array of satellites autonomously after launch in order to configure themselves into a large primary aperture. The complexity of this main objective requires that range and pose data of all components be read and interpreted at a pace fast enough for safe docking to occur.

In order to accomplish this task, the Raspberry Pi has been selected as the on-board computer for the mission due to its low cost and the wide array of software available to it [1]. Though low cost, the Raspberry Pi is able to meet the specifications for hardware and software of a small satellite mission. This project aims to select an appropriate operating system and optimise the entire software stack to improve the determinism and operational speed at which existing drivers for AAReST will be able to process input.

1.1 Background and Context

With current technology, the maximum size of a satellite's primary aperture is limited to that of the launch vehicle. As an example of how this limit can be expanded, the Autonomous Assembly of a Reconfigurable Space Telescope (AAReST) mission was designed with the intent to demonstrate that assembling a larger primary aperture could be accomplished by assembling an array of smaller mirrors after launch. Developed by CalTech/NASA Jet Propulsion Laboratory and the Surrey Space Centre (SSC), the mission will consist of multiple CubeSat's with specialized mirrors, using multiple technologies to be able to cooperate and reconfigure themselves in space autonomously to assemble and reconfigure themselves to a larger functional mirror [2].

Assembling an array of multiple smaller satellites to function as a single large unit has benefits, but also requires that many issues be addressed to handle the complexity of the mission.

The AAReST mission is composed of multiple mirror components ("MirrorSats") as well as a central component ("CoreSat"). With multiple elements autonomously docking and reconfiguring after launch, an important ability for each individual portion to is be able to determine where all other satellites are. This has to be precise enough for the docking to succeed automatically, and as such has to be fairly precise. The method chosen to handle this task in the AAReST mission is by means of LIDAR. Selected for the task was originally specified as a Softkinetic DS325 LIDAR camera. [3].

As an alternative to a more expensive satellite-specific on-board computer, the Raspberry Pi is a small computer capable of a variety of tasks. Initially designed as a teaching tool, the variety of Raspberry Pi models have proven to be approachable and versatile enough to be useful in a wide array of applications. This popularity has resulted in an extensive community generating material and new software, resulting in an accessible repository of reference information. The Raspberry Pi has been demonstrated to be reliable on the ground for many projects, and as such can be reliable for the mission specifications set out by AAReST.

1.2 Scope and Objectives

The scope of the project was ultimately to optimize the LIDAR data stream by several means in order to reduce the memory consumption and ultimately provide a platform for a higher-frequency input processing than the existing setup. Two main ways identified for this goal was to identify and adjust an operating system to be smaller in size than the default installation used previously. A second way would be to develop an improved code for the existing LIDAR drivers and libraries to be ported to, where future developments could be computed. This software based task encompassed a variety of potential points where optimization could occur. This section will briefly outline the four main tasks in order, and how they aim to achieve the final desired result of faster processing of LIDAR data.

1.2.1 Selecting a suitable RTOS for the mission

One of the first aims within the scope of the project was to investigate if an existing operating system would perform better than the generally used operating system for the Raspberry Pi (Raspbian). This will be done by performing benchmark tests on a variety of suitable operating systems in order to narrow down which one is the most suitable operating system for the task. The tests will be selected to test a variety of tasks related to the requirements of the AAReST mission and ranked on how critical it is that the values are as low as possible in a given test. A list of operating systems selected from research, possessing qualities, are all subjected to similar tests to determine which one is best for the project.

Once all operating systems are selected and benchmarked, the test results can be compared in a numerical fashion to determine which operating system is the most optimal for the task.

1.2.2 Optimize the operating system for the task

Once selected, the operating system will be analysed in detail as to what is included in the source code. From there, decisions will be made as to what will not be needed for the mission and the source code will be modified and trimmed down accordingly to a smaller size. Afterwards, the same benchmarking tests will be performed and compared to the original tests to demonstrate the expected improvements in performance.

1.2.3 Develop kernel-level software to complement existing code

After the operating system has been optimized, the mission-specific software is the next task. Improving the processing speed at which LIDAR data is handled will improve the autonomous navigation and docking by providing more accurate range and pose data during the manoeuvre. This is in part going to be accomplished by rewriting the existing code for general speedup, as well as make use of kernel space to speed up the calculations. This will require understanding the kernel of the selected operating system to make a solution, which processes LIDAR data at an increased rate.

1.2.4 Perform tests on final product

When the kernel has been optimized to a degree seen as a reasonable extent within the scope of the project, tests will be performed on the final product to determine the stability and functionality of the final software, as well as detect any crucial bugs that might have been missed during the development stage. This will be done in part with CalTech to test the behaviour of the software with the commands for the satellite, and measuring the feedback data. The specific testing procedures will be determined with the requirements set out by both parties.

1.3 Overview of Dissertation

Starting with a review of existing literature, the first section of the report details the research and rationale behind the operating systems chosen for further testing. Reviews of the requirements set out by prior work on AAReST are analysed, as well as requirements for future and current development. From research, emphasis was placed on selecting a real-time operating system that would be compatible with existing drivers and software used for the AAReST mission. These requirements narrowed down the research to a select handful of operating systems available on the Raspberry Pi.

Afterwards, the practical work will be looked into involving benchmarking tests and rationale behind narrowing down the potential operating system choices. The test method is described and explained, and existing data is commented on. Initial results and testing methods will set the groundwork and reference points for the stages of development that follow.

2 LITERATURE REVIEW

2.1 Introduction

In this section, the rationale and research behind the selection process for the final operating system is focused on. Before any modifications to the existing code could be done, an operating system for the satellites had to be chosen. Background into many of the operating system components were analysed, as well as reasoning which critical areas could be seen as an important requirement for the AAReST mission's software. The final OS had to comply with existing work and suit the specifications of the mission, and it had to be proven numerically how the selected operating systems performed in comparison to each other. Research into the required functionality and the available operating systems had to be performed in order to make an informed decision for a suitable OS. This task involved research on many open-source projects due to the nature of the Raspberry Pi community.

Prior work of the AAReST mission is analysed first in order to determine a list of qualities to adhere to, both in the interest of the hardware and software used and developed for the mission. These requirements could then be used to select what operating systems would be likely to perform well within requirements. In addition, the parameters for benchmark testing derive from the requirements of the AAReST mission and determine what parameters will be analysed in the testing phase of the project.

2.2 Prior work on the LIDAR software for AAReST

Prior work by R. Duke on the software had been conducted when the LIDAR camera was selected and drivers were created and modified for the initial testing setup. These tests were performed with a default Raspbian install to connect the initial choice of LIDAR camera, and the modified OpenNI2DS325 Driver used to create a testing framework and base the final software on. This driver uses a USB connection to communicate between the two pieces of hardware. In addition, other input is handled through the USB ports as well such as LIDAR operation and Wi-Fi communications. With the final setup of the dissertation, the LIDAR setup was able to achieve a maximum rate of about 20 frames of data per second. Though this exceeded the 1 frame/second that the design was initially working at with manufacturer's setup, it was suggested that future improvements could be done by removing existing parts of the setup that were not needed, and streamlining the overall workflow. Not discussed in the report is the possibility of optimizations from making use of available kernel space [4].

2.3 Operating systems and drivers

To run existing software on the Raspberry Pi, it was decided that an operating system suited for the task was needed. An operating system is generally used to decrease the overall complexity of developing new software and provide an environment to run existing solutions. The use of an operating system (OS) provides the opportunity to decrease the development time and provides a more reliable system from use of existing code that has been proven to be stable.

2.4 Purpose of an operating system in the AAReST mission

An operating system is software programmed at low-level intended to provide an interface between the hardware and user-level software, as well as provide an architecture to manage both hardware and software facilities. This makes it possible for developers to write software with less concern about the details and complexity of the hardware that the software is running on. Tasks such as performing calculations and instructions, loading memory appropriately, and managing files and users are all tasks handled by the operating system. Taking these lower-level issues out of software development allows for software to be managed safely and allows for higher-level software to be more portable, as opposed to being bespoke for any variation in hardware. As existing work on AAReST made use of software at a higher level, an operating system was seen as beneficial to the project for both retaining compatibility with existing work, and for using existing software at future points in development [5].

2.5 UNIX derivatives and Linux operating systems

An example of an operating system design philosophy that influenced many modern operating systems is UNIX. Designed to be possible to run to a wide array of computers with varying applications, UNIX is portable and can run on almost any hardware without trouble from any differences in the machines. It is also possible to use in a wide variety of solutions and is equally capable in desktop and embedded applications [6].

In general, the UNIX design lends itself to a smaller operating system better suited for resource-intensive or embedded solutions. Importantly for AAReST, it is able to run multiple processes simultaneously, allowing for multiple critical processes to be run at the same time without interruption. As the in-orbit reconfiguration requires both position feedback from the LIDAR at the same time as each module manoeuvres itself with respect to this information, it is important to allow multiple processes running at the same time in any given Raspberry Pi.

Linux is a UNIX-like operating system that is notable for being open source. Unlike many common commercial operating systems, the source code for Linux is available freely for anybody to view and modify [7]. In conjunction with an active user base, this has resulted in the development of a stable series of kernels with any user freely able to repair any bugs they come across. As such, there is a wide user base active in improving and modifying the source constantly. Being freely available for anybody to alter, it has been the operating system of choice for the Raspberry Pi. The most common distributions used for the Raspberry Pi are derivatives of desktop Linux versions ("Debian Wheezy" and "Debian Jessie") altered to function on the Raspberry Pi. These Raspbian-based operating systems are able to make use of existing software created for Linux. These points are both useful for the AAReST mission, as it makes it possible to obtain the source code and modify it to better suit the mission. In addition, it makes it possible to use software developed for other versions of Linux to perform tests or compile code in a desired manner.

2.6 System image files

The method of choice for installing a new operating system was by use of system image files (With file extension ".img"). A system image file is a copy of the entire system is read and translated in a serial manner that can be easily stored in a single file. This file can be used to recreate an exact copy of the system on other devices by installing the image with the appropriate software tools. In addition, compiling source code results in an image file that can be installed either on the host system that the image was compiled on, or cross-compiled to be suited for different hardware. Cross-compiling in this manner is appropriate for compiling faster by using a more powerful system than the intended hardware.

2.7 Device drivers

To provide an interface between hardware and software, a device driver is needed to define how new hardware is operated, as well as provide functions that can be used for user-level software. These are treated as black boxes in many ways, and can be loaded and removed as needed for the purposes of the operating system. Like the operating system itself, it makes it simpler to program higher-level functions and benefits from the OS by not requiring the same level of complexity and can make use of existing software structures to access hardware [8].

For the AAReST mission, important device drivers include drivers enabling communication over Wi-Fi and USB channels, as well as a driver to use the LIDAR camera effectively. However, many images for Raspbian kernel variations contain default drivers that serve no purpose for the objectives of AAReST. By default, the generally available Raspbian OS is built with general use drivers and software intended for use in an educational environment, which will not be used in an embedded environment [9]. It has been proposed that removing these will increase the performance by speeding up general response time and freeing up more of the available RAM for use in processes more important to the mission.

2.8 Selecting an operating system

From the specifications of the AAReST mission, it was clear that the operating system for the project had to be for the Raspberry Pi, and able to make use of the existing software and drivers used in the project. In addition, the autonomous reconfiguration sets fort an additional requirement that any time-critical operations are handled consistently, as a delay in a vital command could severely impact the satellite operation.

2.9 Device drivers

One property highlighted as a requirement at the beginning of the task was that the final operating system had to function in real time. A Real-Time Operating System (RTOS) is different from most operating systems in that there is a hard time limit for when tasks must be completed internally [10]. In a soft real-time system, data is discarded if the time requirement is not met. This is not suitable for a complicated system, and hard real-time is preferred in cases such as a satellite mission. In a hard real-time system, the time requirements of internal communication must be met. In general, this sort of hard real-time behaviour is not needed and thus are by default, general-use operating systems are not usually designed as hard real-time operating systems. This property becomes problematic when any time-sensitive operation is not given the required priority within a standard OS, potentially delaying a task significantly. As such, it was important to decide on this property of potential operating systems. This defined whether internal deadlines could be missed or delayed, as can be shown for many time-sensitive software behaviours.

This property of an RTOS operating system is important for AAReST mission because during the automated reconfiguration and docking, missing an internal software deadline would result in less accurate results. This in turn will not produce satisfactory results in such a delicate operation. Especially so in the LIDAR, where incorrect data will result in inaccurate final decisions when trying to calculate the navigation needed to dock. As this is crucial to the mission, ensuring all internal deadlines are met via use of an RTOS is key to the success of the mission objectives.

2.10 Existing operating systems for the Raspberry Pi

The Raspberry Pi is generally run with a LINUX-based operating systems such as Raspbian [11]. As such, the majority of documentation and derived operating systems are similar to previous operating systems. Despite Linux on its own not being an RTOS, there are patches to accomplish this in a way that could be used for most purposes. For both examples investigated, real-time functionality is added with the inclusion of a custom kernel designed for the purpose. The Xenomai kernel is a freely available patch that has been made for a variety of systems, including the popular Raspberry Pi operating system Raspbian Wheezy [12]. Another solution is available from Raspberry Pi based autopilot Navio+ developer EMLID, being a modified Raspbian 3.18 kernel which relies on lowering the latency using the PREEMPT functionality available in the default release [13]. These custom kernels result in an RTOS that is still a Linux derivative, and thus has a fairly expansive documentation and community behind any questions or problems coming up in development. However, not being designed initially as an RTOS means that these solutions may not offer the most optimal final product.

An alternative to the standard operating systems generally endorsed for the Raspberry Pi would be an RTOS designed from the start to function in real time. One popular alternative is FreeRTOS [14], a popular operating system for RT purposes, and is commonly installed as a default OS on CubeSat-specific on-board computers being sold. The popularity of this RTOS has resulted in active development for a functional port of the OS to the Raspberry Pi. Other popular RTOSs have ports to the Raspberry Pi such as Chibios and RTEMS, but there are not as complete as FreeRTOS is. Despite that, these operating systems have the benefit of being generally described as being smaller than Linux-based operating systems by virtue of being initially designed as an RTOS, and not relying on a kernel to run a non-RTOS.

One additional option that was discussed was KubOS, an open-source RTOS designed with CubeSats in mind [15]. During the development, discussions took place with the developers to discuss a potential port to the Raspberry Pi that would benefit both parties. This had the benefit of perhaps resulting in an optimised system alongside communications with the development team when designing any on-board software. However, at the time a decision for the project had to be made there was no such port to develop with. As such, KubOS was not considered with the rest of the selected operating systems.

With the knowledge of requirements for the new operating system from research, the final decision came down to analysis of the selection of operating systems to the requirements of the mission. Restrictions from prior software and future developments and requirements must be considered to make a decision of the final operating system for the task.

2.11 Summary

From analyzing existing work and the specifications of the AAReST mission, research delved into determining what sort of operating systems were available for the Raspberry Pi. A variety of ports and modified kernels for real-time operating systems were identified. However, more practical work would be required in order to narrow down the wide range of available choices to a smaller selection more suited for the next stage, as will be discussed as the benchmarking parameters are laid out and refined.

3 BENCHMARKING OPERATING SYSTEMS

3.1 Introduction

The testing phase of the operating systems is the first task that is addressed. Prior to testing, the operating system selection is narrowed down to a set of three different solutions that suit the workflow. Detailed are the tools selected and the test setup for the phase, as well as data that has been obtained and analysed. Existing information and the decisions behind the test setup are discussed.

Following the initial testing, the next objectives and the methods to approach the goals are detailed and discussed in order. Changes to the chosen operating system, the modified LIDAR driver, and the final testing phase are discussed in order.

3.2 Operating system selection

With knowledge from research, a set of criteria for the final operating system could be defined. This knowledge could be used to take the diverse list of options available for the raspberry pi and narrow down the options to a smaller selection. With a manageable list refined, the final operating system can be selected from analysis and benchmark testing.

3.2.1 Criteria for the final operating system

Using the Raspberry Pi as the on-board computer in the AAReST mission resulted in a particular selection of available operating systems for the task. The final operating system can be selected from a wide variety of ports and specialised operating systems for the Raspberry Pi, which have the benefit of generally being open-source and having the source code readily available. This is beneficial for future development, as it was determined that modifying the source code would result in a final system more suited for the mission. The particular changes to be considered to the final source code are discussed in further detail in section 3.2.1.

In order to narrow down the wide variety of options, knowledge of requirements for the AAReST mission highlighted certain important aspects and requirements of the OS. In order for all proposed functionality to run as expected, the chosen operating system must have USB and WiFi drivers. This is important both for Wi-Fi communication between individual segments during flight, but also for the LIDAR camera connected via USB. As the Raspberry Pi has USB ports already built in and is supported very well by default, it can be reasonably expected that users will demand these capabilities of any Raspberry Pi operating system [16].

The operating system must be an RTOS for the Raspberry Pi, a specification noted during the literature review. In addition, the operating system should consume as little memory as possible, whilst still keeping development time reasonable.

3.2.2 Selection process and final operating system

Initially, four potential solutions to the operating system problem were chosen from the research that are already ported to the Raspberry Pi. These four were:

- Raspbian Wheezy with RTLinux RT kernel patch
- Raspbian Wheezy with Xenomai RT kernel patch
- FreeRTOS
- ChibiOS

Two patched Raspbian Wheezy distributions were chosen because of the larger amount of documentation and information freely available for development, especially with future developments to the operating system itself and the addition of kernels in mind. FreeRTOS was seen as reliable for spaceflight, and ChibiOS was included as well as it was an existing port of an established RTOS for the Raspberry Pi.

Before any initial testing, discussions concerning the AAReST mission took place in order to determine which of these operating systems would be considered viable and beneficial to the project. At this point, existing progress was analysed. From existing expertise on previous projects within the SSC and an active online community, a Linux-based OS was preferred. Adhering to this decision would likewise make use of existing drivers for networking and USB communications, as well as prior work on LIDAR system drivers and software.

With the decision down to two different Raspbian Wheezy installs with a selection of patches for real-time operation, the tests performed were done with a non-patched install as well to provide a comparison. Despite not being a viable operating system for the task without a kernel for real-time, this was used as a baseline to compare the difference made by the operation of the kernel. As such, future developments could be done with knowledge of the impact that the RT kernel has on the system.

The final selection of operating systems and distribution tests was ultimately:

- Raspbian Wheezy Version 3.18 (To compare the impact of the RT kernels)

- Minibian Jessie version (To compare with the patched Minibian Xenomai kernel)
- Raspbian Wheezy patched with EMLID RT kernel version 3.18.9-rt5-v7+
- Raspbian Wheezy Version 3.10 patched with Xenomai version 2.6.3
- Minibian Wheezy Version 3.10 patched with Xenomai version 2.6.3

Of note is the existence of a patched Minibian install, which is a minimal requirement installation of Raspbian. Minibian is specifically compiled in in a way that reduces the size, and is designed for more embedded tasks than the default Raspbian [17]. This was chosen late in the process as the potential benefits of Xenomai on the smaller Minibian might outweigh any potential drawbacks sue to the larger size of Xenomai in comparison to EMLID's kernel. However, whether the optimal processing would come from the minimal install or not had to be determined from benchmark testing.

One of the first requirements was to analyse the size of the source code for the operating system. As optimising the memory use and consumption of the final software would be beneficial to the speed of LIDAR processing and other tasks, an operating system that is smaller from the start is preferable. In order to determine how the operating systems would compare, they were compared before install (both the source code and the image) as well as the size consumed on a clean install to the Raspberry Pi's memory card.

	Image size (GB)	Installed size (MB)
Raspbian Wheezy 3.18 (Calibration)	3.05	18.9
Wheezy w. EMLID's kernel	3.07	30.7
Wheezy w. Xenomai kernel	3.69	21.1
Minibian w. Xenomai kernel	0.97	20.5

Figure 1. Size of the source code and images of the initially chosen operating systems.

From initial image size, it can be seen that the Xenomai kernel increases the image size itself by several GB, whereas the kernel from EMILD solution appears to be much more streamlined in comparison in only increasing the size by a few GB. However, this difference is inverted after installation, making Xenomai the option that consumes less memory. It can also be seen that despite the large size difference on the images, the final Minibian write does not retain the size difference after installation.

The initial size gives a basic indication of the impact, but not much insight into the operational speed of the install. In order to compare this, a benchmarking tool was used to compare a selection of operations that could be compared numerically after testing to determine the most practical solution for the task.

3.3 Benchmarking

In order to determine the most suitable kernel for the task, a series of benchmarking tests were derived to compare the different setups in a numeric fashion. This can be accomplished by using a software that runs a set of tasks to measure the performance of the operating system, recording desired parameters that can be graphed and compared numerically to view any differences in performance. For most of the tests used in selecting an operating system for the AAReST mission, the execution time for a given benchmark task is used as the parameter to compare performance as well as demonstrate the impact of optimizations.

The benchmarking tool chosen for the task was SysBench, which is a benchmarking tool capable of performing tests on various aspects of system performance [18]. Though benchmarking is generally performed to test database performance, the tools offered by SysBench were applicable to analysing the performance of an operating system. The parameters that were of interest to the project were fairly general file I/O and memory allocation performance. It was likewise important to check the file I/O over USB, to ensure that there was no major negative impact on the performance of a fairly crucial communication channel. As there was no existing test for the purpose of testing communication over a USB channel, the performance of file I/O was tested both with the testing point being in the main directory and over USB, which made it possible to compare the difference between the two different locations on the same operating system.

Installing sysbench required that additional libraries were installed alongside sysbench itself. The tests will all be performed with the following libraries installed:

- Sysbench-0.5
- Automake-1.12
- Autoconf-2.69
- Libtool-2.4.4

The four tests that were selected tested a variety of operations to demonstrate the capabilities of the kernel in comparison to the unmodified operating system.

3.3.1 CPU Test

```
sysbench --test=cpu --cpu-max-prime=2000 run
```

CPU is one of the more basic modes where the CPU is tested by calculating prime numbers until the specified maximum value is met to determine general response time. The selected maximum prime for this test was set to a smaller value than the suggested amount, as the Raspberry Pi lacks the processing power associated with the larger desktop PCs and databases that the standard value of 20000 is intended for.

This test was not particularly critical for overall performance, but provided insight into performance during calculations.

3.3.2 FileIO Test

```
sysbench --num-threads=16 --test=fileio --file-total-size=3G --file-test-mode=rndrw prepare sysbench --num-threads=16 --test=fileio --file-total-size=3G --file-test-mode=rndrw --max-time=300 run sysbench --num-threads=16 --test=fileio --file-total-size=3G --file-test-mode=rndrw cleanup
```

In order to test file read and write, the FileIO test requires that test files are generated beforehand to read and write from at random during the test. As such, the prepare and clean-up settings are used to both generate and clean up a set of 128 randomised test files to add up to the maximum size for the test. One prepared, the *rndrw* mode is used to test random read/write operations on these files in the span of 300 seconds, a time frame set by the max-time option.

Due to the fact that the test files are all generated in the current directory when called, tests can be performed at any point in the system. Though changing the directory for the most part will not change the results much, one condition where it will be is when the current directory is a USB drive.

	Raspberry Pi main					**USB files**				
		Response time (ms)					Response time (ms)			
	Requests/sec	Min	Avg	Max	Appx. 95th	Requests/sec	Min	Avg	Max	Appx. 95th
Raspbian Wheezy 3.10 (Calibration)	4.17	0.06	0.15	9.6	0.25	1.86	0.08	1241.17	22960.22	9423.66

Figure 2. Comparing FileIO tests performed on a USB compared to the main directory of the Raspberry Pi

As can be seen in Fig.2, there is a highly noticeable slowdown when tests are performed on a USB drive instead of in the main directory. This implies that there is some overhead that should be considered and monitored across all tests. As USB communications are an important part of the workflow within the AAReST satellites, it is not satisfactory to only perform the FileIO test in the main directory. As such, two FileIO tests must be run on every prospective OS to ensure that the USB communications remains at a satisfactory speed.

Though FileIO are not necessarily a time-critical task on the AAReST mission, the ability to perform the same tests both at root and on a connected USB drive demonstrated the comparative performance of operations over USB in comparison to the same operation with no USB communication involved. As AAReST will rely on USB, it is valuable to know the impact of data transfer speed over USB.

3.3.3 Threads Test

```
sysbench --test=threads --thread-locks=1 run
```

The CPU test is designed to benchmark the performance of the scheduler when there are a large amount of threads present, determined by manipulating a large amount of active while keeping one thread active at the time and switching between the various threads. This gives insight into the processing speed that the OS handles in general with multiple threads being managed. Though informative, AAReST may not necessarily have more than one thread running at any point.

3.3.4 Mutex Test

```
sysbench --test=mutex --num-threads=64 run
```

Acting as an inverse to the “threads” test, mutex tests the response when multiple threads are handled concurrently with each thread only being locked occasionally to increment a test variable. Mutex, short for mutual exclusion, is a property of the kernel used to regulate concurrent processes so that multiple processes do not attempt to use the same critical resources at the same time. Instead, the threads are regulated in order to ensure no conflicts arise. [19]

Of all the benchmarking tests, the Mutex test was used as one of the more important tests. Having reliable response times from the MUTEX is critical in time-sensitive operations.

3.4 Summary

From prior research, a final selection of candidate operating systems were selected for the benchmarking test phase. In line with requirements that would be important to know for the AAReST mission, a series of benchmarking tests within SysBench were identified and designed. With an approach that could be used at multiple points in the course of the project, the benchmarking tests could take place to analyse the operating systems and determine if any performed better than the default Raspbian.

4 BENCHMARKING TEST RESULTS

4.1 Introduction

Once the benchmarking tests had been designed, the testing phase could take place. Afterwards the resulting data was then graphed and analysed in order to note trends in performance for the range of tests presented. Individual test results were examined and an overall overview could be determined from the data. It was possible to determine from the given data which operating system would be a more suitable option for the AAReST mission. The trends observed and the operating system selected for the next stages of development will be summarized in this section.

4.1.1 Mutex test

	Mutex					
	Response time (ms)					
	Min	Avg	Max	Appx. 95 percentile	Min-max	Avg-95th
Raspbian Wheezy 3.10 (Calibration)	1435.97	2625.33	2951.54	2932.34	1515.57	307.01
Wheezy w. Xenomai kernel	1638.55	2457.44	2878.12	2752.04	1239.57	294.6
Minibian	1379.09	2346.61	2704.73	2683.7	1325.64	337.09
Minibian w. Xenomai kernel	1651.25	2549.56	2888.34	2869.82	1237.09	320.26
Wheezy w. Emild's kernel	11938.78	12303.93	12477.71	12470.99	538.93	167.06

Figure 3. Table of Mutex test results

Of the four tests, the mutex test results were the most critical in determining the suitability of the operating system. As with all the tests, the overall performance was not considered as important as the stability. This parameter was determined by the min-max difference and the average-95th percentile difference. As the final source code was to be modified and recompiled at a later stage to improve overall performance, this was not seen as a critical concern at this stage.

In most tests, the variation is fairly consistent at above 1000 ms in the min-max gap, with the notable exception of the Emlid kernel. In both tests, it performs with less than half of the range variation. While the numerical performance was very poor in comparison, the reliability proved to be far more important and it was thus deemed to have the best performance in the mutex benchmarking tests.

4.1.2 Threads Test

	Threads					
	Response time (ms)					
	Min	Avg	Max	Appx. 95	Min-max	Avg-95th
Raspbian Wheezy 3.10 (Calibration)	3.05	3.25	4.97	3.31	1.92	0.06
Wheezy w. Xenomai kernel	2.21	2.25	6.37	2.31	4.16	0.06
Minibian	2.62	2.67	3.97	2.73	1.35	0.06
Minibian w. Xenomai kernel	2.58	2.67	7.84	2.74	5.26	0.06
Wheezy w. Emild's kernel	5.26	5.74	9.78	5.88	4.52	0.07

Figure 4. Table of thread test results

In general, the performance of the threads tests remained within a similar range, usually within a range of around 4 ms. Of all the operating systems, Raspbian Wheezy and the Minibian kernel performed better with the min-max stability being less than 2 ms difference in values, less than half of the range of the next closest RT kernel. This implies that any real-time operation has a negative impact in this regard. From observation, the minimum was consistent, and the maximum test value was the main factor in the final performance comparisons. Though not as critical as the mutex tests, the results provide insight into the operation of the kernels

4.1.3 FileIO test

	Raspberry Pi main				
	Response time (ms)				
	Requests/sec	Min	Avg	Max	Appx. 95 percentile
Raspbian Wheezy 3.10 (Calibration)	4.17	0.06	0.15	9.6	0.25
Wheezy w. Xenomai kernel	4.97	0.1	701.61	15995.19	4641.27
Minibian	4.3	0.08	1037.36	13664.57	6691.11
Minibian w. Xenomai kernel	4.91	0.11	761.55	12193.64	4545.03
Wheezy w. Emild's kernel	4.24	0.17	902.1	13879.28	5761

Figure 5. Table of FileIO test result at root

	USB files				
	Response time (ms)				
	Requests/sec	Min	Avg	Max	Appx. 95 percentile
Raspbian Wheezy 3.10 (Calibration)	1.86	0.08	1241.17	22960.22	9423.66
Wheezy w. Xenomai kernel	1.81	0.08	932.66	18826.7	6761.58
Minibian	1.96	0.09	727.37	14849.84	6743.39
Minibian w. Xenomai kernel	1.84	0.1	981.32	20263.25	6709.16
Wheezy w. Emild's kernel	1.6	0.24	825.67	14870.25	4103.99

Figure 6. Table of FileIO test results on USB

	Raspberry Pi main			
	Response time (ms)			
	Min-max	Avg-95th	Min-max	Avg-95th
Raspbian Wheezy 3.10 (Calibration)	9.54	0.1	22960.14	8182.49
Wheezy w. Xenomai kernel	15995.09	3939.66	18826.62	5828.92
Minibian	15995.09	3939.66	18826.62	5828.92
Minibian w. Xenomai kernel	13664.49	5653.75	14849.75	6016.02
Wheezy w. Emild's kernel	12193.53	3783.48	20263.15	5727.84

Figure 7. Table of comparative FileIO test results

As a comparison of performance over USB, the same benchmarking operation was performed in identical conditions, with the only difference in the location of the test file. For these tests, a USB drive was used across all tests with an identical setup for all operating systems. The comparative difference, as well as the overall operation, were used to analyse the performance.

What could be noted was how in all cases except for the original Raspbian Wheezy kernel, there was a significant increase in response time from all FileIO operations. While not a concern on its own, it can be noted that in all cases, there is an increase in performance when the same operation is performed over a USB channel. This implies that there is a mild to moderate overhead in any operations performed over USB. This is of note, as the AAReST mission requirements involve communication over USB for many operations. Reducing this overhead should be a point of concern during future stages of optimization.

4.1.4 CPU test

	CPU (Primes)					
	Response time (ms)					
	Min	Avg	Max	Appx. 95th	Min-max	Avg-95th
Raspbian Wheezy 3.10 (Calibration)	6.3	6.38	8.19	6.41	1.89	0.03
Wheezy w. Xenomai kernel	6.29	6.38	11.71	6.45	5.42	0.07
Minibian	5.35	5.43	11.78	5.48	5.42	0.07
Minibian w. Xenomai kernel	5.34	5.41	10.78	5.45	6.43	0.05
Wheezy w. Emild's kernel	9.38	9.65	16.09	9.75	5.44	0.04

Figure . Table of CPU test results

Though not critical to any operation on the AAReST mission, the CPU primes test was performed to further demonstrate the comparative operation of the different operating systems. As seen in many tests, the overall performance is negatively impacted by the RT kernel additions. This is a factor that is anticipated to improve with the completion of the next stage of the project, and in a less critical test was not seen as too much of a deciding factor to not use one of the RT kernels.

4.2 Benchmarking result data

In order to determine the stability of an OS, two parameters for each test were considered outside of the numerical values of each of the four categories. The difference between the minimum and maximum value as well as the difference between the average and the approximate standard 95th percentile values were used as indicators of performance as opposed to the numerical values. These parameters gave insight into how reliable the output of a given operating system was.

In real-time operations, the consistency of operation is an important parameter, as a large range of operation could cause problems in time-sensitive functions. As such, it was possible to determine the maximum variance as well as how much would normally be expected in terms of operation. It was important to select an operating system that would perform the same operation with as little variation it time as possible in order to ensure the operation is as reliable as possible.

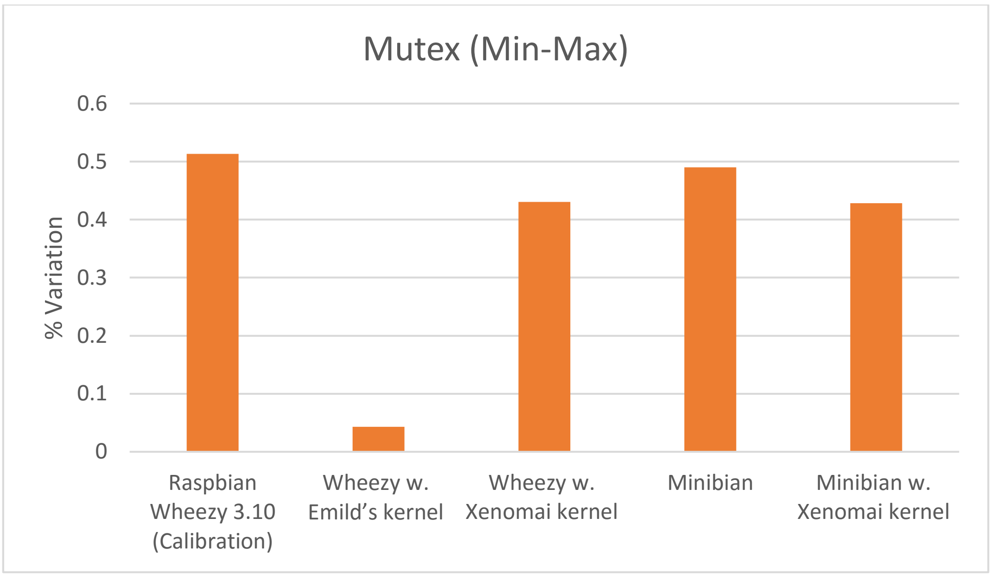


Figure 8. A graphical representation of the Mutex comparative min/max difference, demonstrating the percentage variability in all data

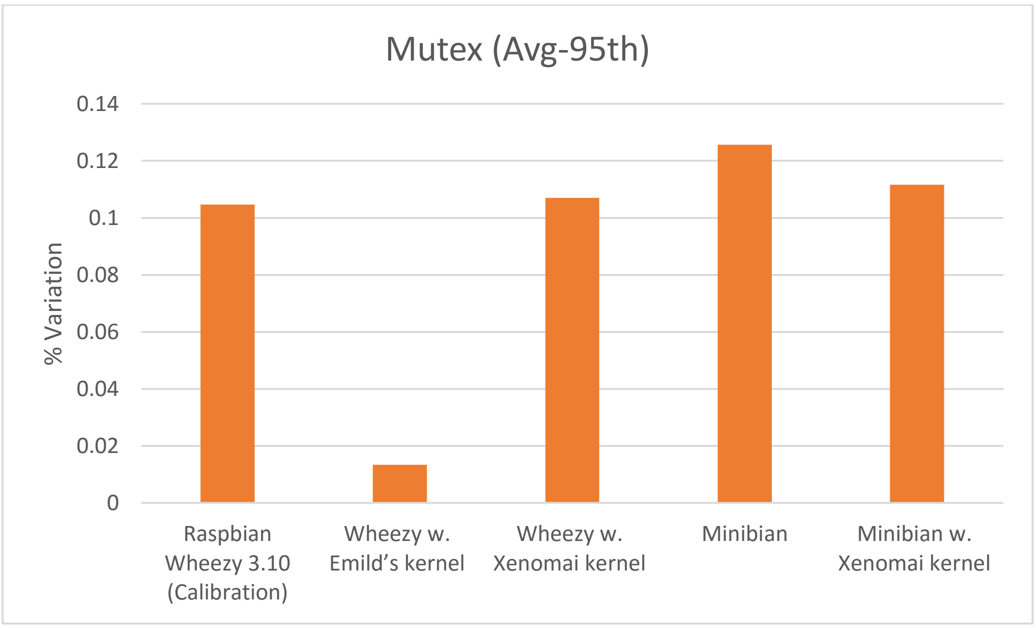


Figure 9. A graphical representation of the Mutex comparative average/standard 95th percentile difference, demonstrating the percentage variability in the average data

Final values used to make a decision were obtained by finding the percentage of error in the cases, with the mute tests being used as the primary goal, with threads and USB performance being secondary. The data gathered shows that Emlid's kernel performed far superior in the mutex test than any other operating system, and never performed worse than average in other critical tests. As such, from the benchmarking tests it was determined that the operating system to be used was the Emlid kernel.

4.3 Summary

Through the tests, several trends in general operation of real-time operating systems could be observed. In some benchmarking tests, such as CPU and FileIO performance tests, there was evidence that the RT operation had a negative impact on performance in these aspects. In addition, all operating systems had overhead when communicating over USB. This test data will provide a reference point for future developments, serving as a comparison as to how modifying the source code will alter the behaviour.

Overall, the RT pre-emptible kernel from Emlid performed consistently well, and had the most reliable performance in the mutex test by a significant margin. As such, it was decided that the next stages would make use of this particular kernel. It is expected that optimising the Emlid kernel will reduce the slower performance experienced in several tests.

5 MODIFYING THE OPERATING SYSTEM

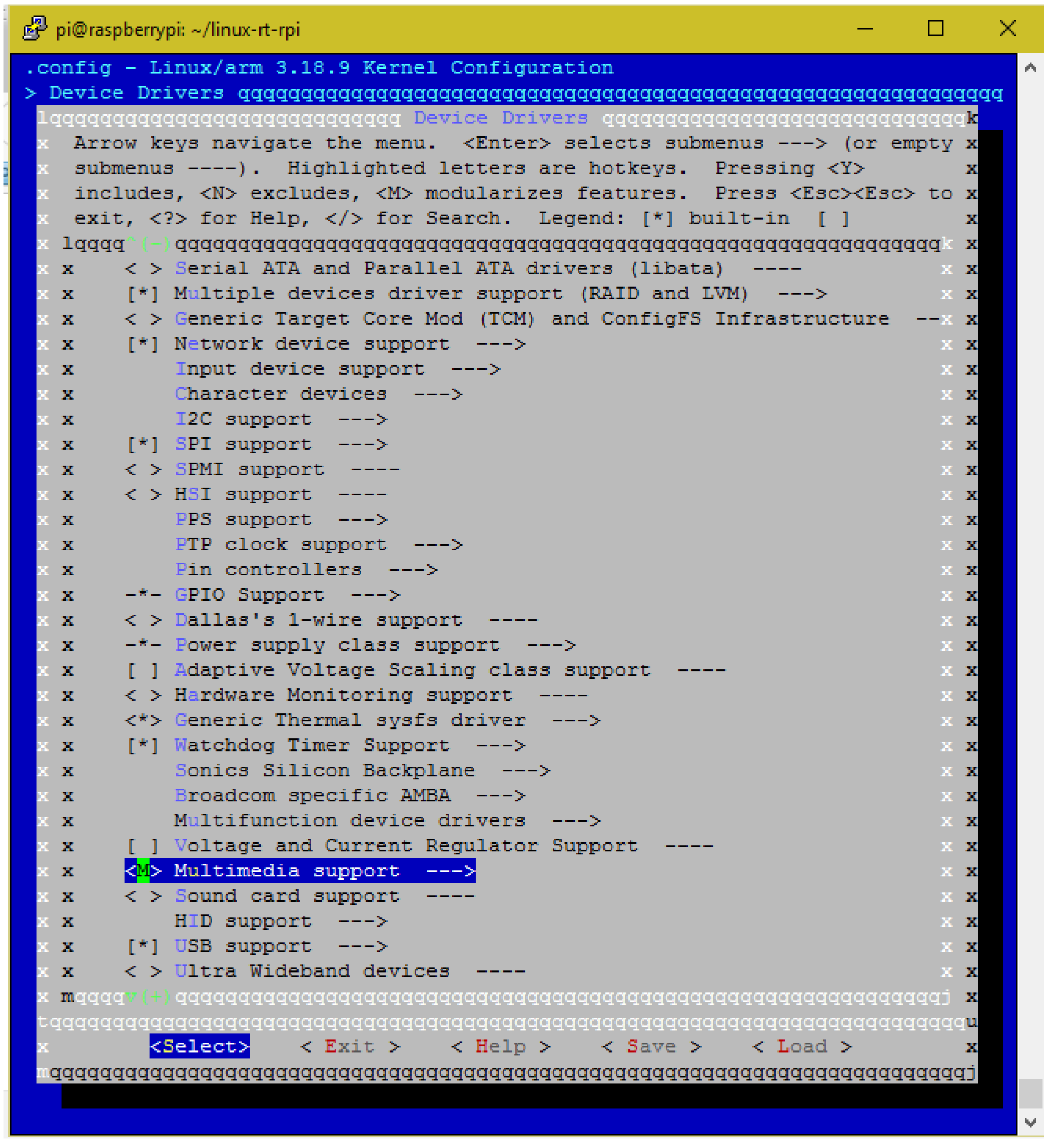
5.1 Introduction

Not all drivers and software components in the base install are needed for the AAReST mission. USB and Wi-Fi are required for LIDAR and inter-satellite communications, but many other existing drivers can be removed to reduce the size of the final operating system. In addition, much of the default software included in a general install are not needed for the AAReST mission [20]. Removing this software frees up memory for more time-sensitive tasks and further improves the speed up. This can be accomplished by carefully modifying the source code to remove unneeded functionality and generate a smaller and more optimal OS customized to the needs of the AAReST mission.

5.2 Modifications to the source code

Many drivers included in a general install are included as part of the image. In an embedded application, many of these drivers are not necessary. As such, any drivers for systems not used in the AAReST mission could be uninstalled to decrease the side of the final operating system. This can be accomplished by modifying the source code and compiling a custom image. This results in an image file that can be used repeatedly to get the same reduced operating system on multiple devices.

Compiling an image was accomplished using the instructions provided by the developer [21], and could be done both on-board or cross-compiled on a more powerful PC. This setup is designed for use on pure Linux systems, and tests confirmed that trying to install the image from a windows system failed to properly install the new operating system. However, using the win32diskimager an image could be generated that could be used in a manner identical to the operating systems tested. A benefit to this was that it was possible to do further uninstallations after modifying the source code to reduce the size additionally.



```
pi@raspberrypi: ~/linux-rt-rpi
.config - Linux/arm 3.18.9 Kernel Configuration
> Device Drivers qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqq
 lqqqqqqqqqqqqqqqqqqqqqqqqqqqqq Device Drivers qqqqqqqqqqqqqqqqqqqqqqqqqqqqqk
 x  Arrow keys navigate the menu.  <Enter> selects submenus ---> (or empty x
 x  submenus ----).  Highlighted letters are hotkeys.  Pressing <Y>       x
 x  includes, <N> excludes, <M> modularizes features.  Press <Esc><Esc> to x
 x  exit, <?> for Help, </> for Search.  Legend: [*] built-in  [ ]        x
 x lqqqq^(-)qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqk x
 x x     < > Serial ATA and Parallel ATA drivers (libata)  ----         x x
 x x     [*] Multiple devices driver support (RAID and LVM)  --->       x x
 x x     < > Generic Target Core Mod (TCM) and ConfigFS Infrastructure --x x
 x x     [*] Network device support  --->                               x x
 x x         Input device support  --->                                 x x
 x x         Character devices  --->                                    x x
 x x         I2C support  --->                                          x x
 x x     [*] SPI support  --->                                          x x
 x x     < > SPMI support  ----                                         x x
 x x     < > HSI support  ----                                          x x
 x x         PPS support  --->                                          x x
 x x         PTP clock support  --->                                    x x
 x x         Pin controllers  --->                                      x x
 x x     -*- GPIO Support  --->                                         x x
 x x     < > Dallas's 1-wire support  ----                              x x
 x x     -*- Power supply class support  --->                           x x
 x x     [ ] Adaptive Voltage Scaling class support  ----               x x
 x x     < > Hardware Monitoring support  ----                          x x
 x x     <*> Generic Thermal sysfs driver  --->                         x x
 x x     [*] Watchdog Timer Support  --->                               x x
 x x         Sonics Silicon Backplane  --->                             x x
 x x         Broadcom specific AMBA  --->                               x x
 x x         Multifunction device drivers  --->                         x x
 x x     [ ] Voltage and Current Regulator Support  ----                x x
 x x     <M> Multimedia support  --->                                   x x
 x x     < > Sound card support  ----                                   x x
 x x         HID support  --->                                          x x
 x x     [*] USB support  --->                                          x x
 x x     < > Ultra Wideband devices  ----                               x x
 x mqqqqv(+)qqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj x
 tqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqu
 x        <Select>    < Exit >    < Help >    < Save >    < Load >        x
 mqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqqj
```

Figure 10. **The menuconfig software used to modify the makefile**

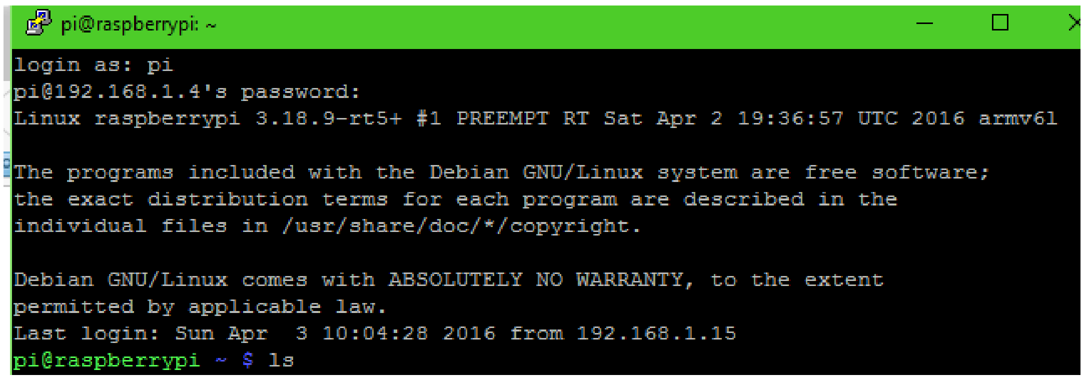
To get a modified installation, the makefiles were used to define what drivers to omit as the image was compiled. Makefiles are used during compiling to direct the compiler to include or exclude drivers without the need to manipulate the more complicated underlying software, or running into problems with not removing all references. Instead, it allows for straightforward customization of kernels from the same source code. The provided makefile was used in conjunction with the installed menuconfig software to create a version that omitted drivers that would not serve any purpose within the specifications of the AAReST mission.

5.3 Removed drivers and impact on performance

By revising the mission requirements for AAReST, it was possible to identify drivers that could be removed from the custom kernel. As all use of the Raspberry Pi during the project was using Ethernet, most input devices were not seen as necessary even for the development phase. Future development can be performed both using Ethernet and by USB communication. As such, driver support for other input devices (Such as pointing devices and keyboards) were removed.

Many devices used in conjunction with the Raspberry Pi make use of the USB port, and general distributions commonly include several of the more common devices within the default installation. Devices such as USB monitors, printers, and controllers were removed as they would not be used at any point in the mission. General USB functionality was left intact for use as a communication channel during the mission.

Support for many sound devices were included in the default installation for speakers and sound cards. These serve no purpose on-board a satellite and were removed along with a variety of minor settings that would not benefit the AAReST hardware in any way. Drivers for specific purposes such as analogue TV support and NFC support were uninstalled to minimize the size of the new image. These broad categories of drivers were used to narrow down the final selection. The list of removed drivers is included in full in Appendix 1.



```
login as: pi
pi@192.168.1.4's password:
Linux raspberrypi 3.18.9-rt5+ #1 PREEMPT RT Sat Apr 2 19:36:57 UTC 2016 armv6l

The programs included with the Debian GNU/Linux system are free software;
the exact distribution terms for each program are described in the
individual files in /usr/share/doc/*/copyright.

Debian GNU/Linux comes with ABSOLUTELY NO WARRANTY, to the extent
permitted by applicable law.
Last login: Sun Apr  3 10:04:28 2016 from 192.168.1.15
pi@raspberrypi ~ $ ls
```

Figure 11. The new image freshly installed. 3.18.9-rt5+ PREEMPT is not a default kernel

As demonstrated in figure 11, the new installation was verified by the version number presented upon login. The kernel image 3.18.9-rt5+ PREEMPT is not a default kernel, further reinforced by the date and time being of when the image was compiled. After the new system image had been compiled and installed on the Raspberry pi, some additional clean-up was needed. By default, many Linux-based kernels keep copies of prior images after the installation of new versions in the "/lib/modules" directory. These can take up a significant amount of space, and removing them is crucial to keeping the freshly installed image at the intended size.

Initial benchmarking tests at this point proved to be promising, with significant decrease in overall response time in all tests. Mutex and USB FileIO tests showed the most improvements out of the tests when compared to the original Emlid kernel.

5.4 Further size reductions

Due to the image being compiled on the same system t was installed on, any software from previous images would remain. Many of these were removed and uninstalled through use of various clean-up commands, while at the same time allowing for further size reductions that were difficult to perform in the makefile stage. As the final image is to be compiled separately to provide a more stable image, this can be done at this stage while keeping a system image with the desired changes.

Much of the software removed was related to drivers that had been uninstalled, such as sound software and libraries. GUI setups and sound file format information could be uninstalled safely, as well as keyboard-related software.

Most of the uninstalled software was related to software languages and libraries outside of the C language used for the software on the AAReST mission. The default installation included many common languages such as Python, Java, and Ruby. Purging these and the included libraries freed up a notable amount of the on-board memory.

	Raspberry Pi main					**USB files**				
		Response time (ms)					**Response time (ms)**			
	Req/sec	**Min**	**Avg**	**Max**	**Appx. 95th**	**Req/sec**	**Min**	**Avg**	**Max**	**Appx. 95th**
Removed drivers	3.82	0.08	1.77	1792.28	0.29	1.91	0.08	95.82	18875.95	5.27
Uninstalled software	4.15	0.08	0.65	56.48	0.48	1.98	0.08	271.62	18875.95	3.95

Figure 12. Final FileIO results at the end of each stage

Despite the large amount of additional uninstallations, further tests proved that there was little to no impact on the overall performance of benchmark tests when compared to omitting drivers from the image. The FileIO tests at root had the only significant change in maximum response time as is shown in figure 12, where a reduction in average and maximum response time is observed. However, the 95th percentile was slightly above the previous average. Fluctuation in average ant 95th percentile values were not seen as critical, as the numbers remained within the range of the original values and were unlikely to be an indication of actual worsened performance in some regard. The same can be said for the slight change in USB FileIO performance, which had a slight increase in average but a decrease in 95th percentile. If performed for far longer, these test variations are likely to average out to be indistinguishable.

5.5 Comparison of old and new operating system performance

By performing the same initial benchmarking tests at multiple points of development, it was possible to see what impact the two stages had on the final performance. As most sets had only minimal improvements after the majority of unneeded drivers, this development was given more attention than the benefits of removing software.

	Image size (GB)	Installed size (MB)	RAM Usage (Kb)
Wheezy w. Emild's kernel	3.07	30.7	52256
Custom minimal Emlid kernel	7.4	23.2	47676

Figure 13. Comparison of overall size of the new kernel compared to the original

As shown in some initial comparisons, there was a definite change in available space on the final kernel. The RAM usage when installed shows that the kernel and associated drivers use up much less of the available space. While the image size is larger, this has not shown itself to be an indication that the new image is larger after installation.

One noted trend was how the response time parameter would not necessarily decrease in a linear fashion. While the average and 95th percentile performance showed a significant decrease, the maximum response time would not necessarily experience the same decrease. This problem was particularly noticeable on the FileIO tests performed over the USB channel.

	USB files				
	Response time (ms)				
	Requests/sec	Min	Avg	Max	Appx. 95th
Wheezy w. Emild’s kernel	1.6	0.24	825.67	14870.25	4103.99
Custom minimal Emlid kernel	1.98	0.08	271.62	18875.95	3.95

Figure 14. Final FileIO results performed over USB

As demonstrated above in figure 14, the maximum value is slightly higher than the initial state. However, the 95th percentile shows that the majority of tests resulted in a response time far smaller than the original. When compared with all operating systems, the response time in this regard is so low that it outperforms all other operating systems tested significantly. As the maximum and average values are influenced by a small number of instances experiencing a long response time (consisting of less than 5% of all signals) the change in performance is still seen as successful due to the majority of FileIO operations being performed at a very high speed.

Outside of the USB FileIO test, the improvement in response time for other tests were closer to what was expected, with a more linear decrease in all parameters. However, the decrease was not even, and the maximum value would not decrease at the same rate as the minimum and average values. The comparative values were therefore not impacted much, but the general improvement in response time resulted in a final image that performed faster and more reliably than the other operating systems on average.

	Mutex					
	Response time (ms)					
	Min	Avg	Max	Appx. 95th	Min-max	Avg-95th
Wheezy w. Emild’s kernel	11938.78	12303.93	12477.71	12470.99	538.93	167.06
Custom minimal Emlid kernel	2449.9	2818.07	3115.82	3113.26	665.92	295.19

Figure 15. Final Mutex test results

Whilst the range of possible response time has increased by around 100 ms, this is still at or below average. Especially so in the min-max range, where previous tests demonstrated that other operating systems could expect at least 1200 ms difference in value range. As such, while keeping the stability the source code was chosen for, the overall response time has been reduced significantly, on average almost 10,000 ms.

5.6 Summary

As expected, response time was decreased with the removal of non-critical drivers. The improvements made the overall performance much faster, with shorter response times in all benchmarking tests. At the same time, the stability remained comparable to the original OS. An additional result was that the USB overhead observed in all previous tests was negligible in the modified operating system. This was not expected from testing at the start, but was a beneficial outcome, especially as much of the communication in AAReST is designed to take place over USB. Viewing the complete data, it was clear that all tests had experienced a notable speedup. The full test data is available in Appendix 2. However, the improvements were demonstrated in some of the more notable tests.

6 OPTIMIZING EXISTING DRIVERS

6.1 Introduction

A second approach to improving the processing speed at which the LIDAR operated at was through the modification and development of software specific to AAReST. By locating critical and resource-intensive existing drivers and software previously developed for LIDAR data and adjusting them to use more of the available user and kernel space. By default, the allocated memory is either purely in user or kernel space, which is not optimal to the task at hand. By making use of both of these allocated memory segments and moving parts of the procedure into kernel space will allow intensive code to be run at a lower level, reducing the amount of layers between the computation and the hardware. This is anticipated to greatly impact the overall processing speed of input for the LIDAR cameras.

Upon software completion, the LIDAR workflow was anticipated to be integrated into the kernel, whereas currently it is exclusively in user space like the TCTM and Wi-Fi workflows. The expected final result is graphically envisioned as in figure 3.

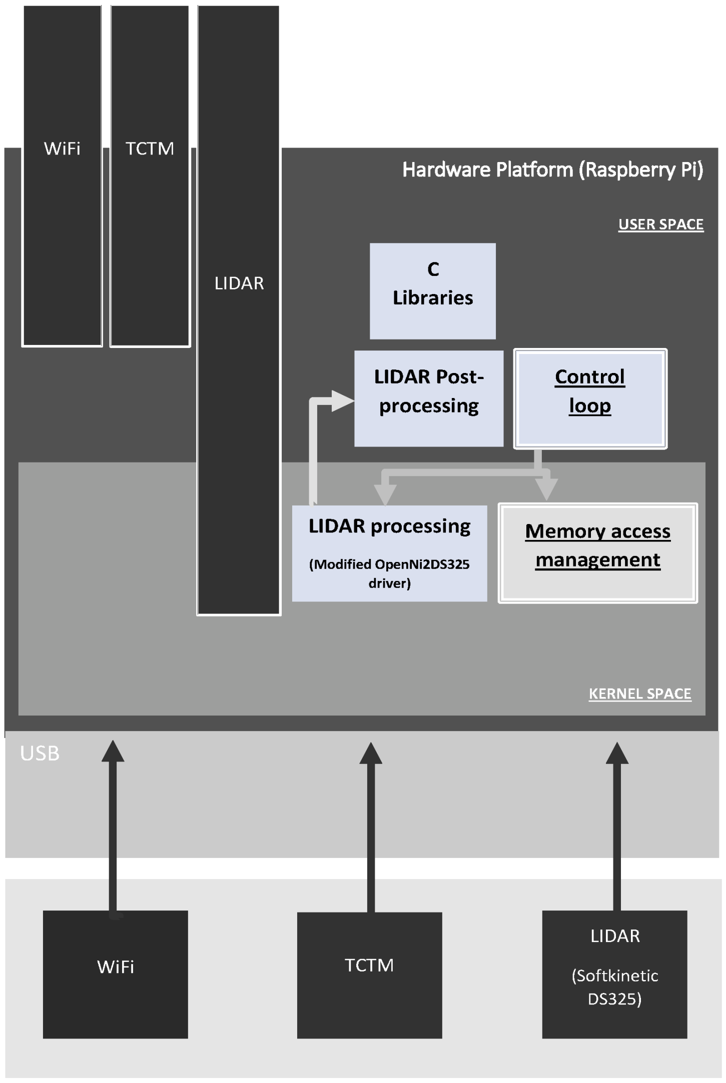


Figure 16. A graphical representation of the software in memory space

6.2 Initial work

Most of the work was expected to be on the existing OpenNI2DS325 driver, as most of the optimization goals were performed in order to improve the LIDAR response and increase the frequency at which new images are generated and processed. If possible, as many of these changes as possible were to be written into a wrapper. This being a separate piece of software that manages the driver, and can be used in future developments to handle telemetry commands to the LIDAR camera.

Initially, the first approach to improving the performance of the OpenNI2DS325 driver was by manually managing the memory usage through the use of a memory map. A memory map process is used to map device memory manually to address space. Though this is generally performed by the kernel and therefore not generally used in most device drivers, this allocation can improve performance in several cases by means of this direct assignment of device memory space [22].

After obtaining the source code for the driver, a method of benchmarking the performance improvement was researched. From the research phase, it was known that a data input of 10 frames/second was possible with the existing driver setup, and this was expected to be exceeded. However, in the time since, the initial camera used in the testing phase was no longer functional. In addition, the Softkinetic DS325 model used was no longer being manufactured. A model identical in specifications had been located, the Softkinetic DS525 being identical in hardware specifications to the Softkinetic DS325 [23]. One was ordered at the same time that software tests were being conducted, but did not arrive in a timely manner for the project. As the expected benchmarking method was to view the frames per second received from the input device, testing in this manner could not proceed as planned.

In order to circumvent this issue with hardware, research into testing software was performed to determine if it was possible to emulate the input to the driver if there was no available hardware. An existing one for the specifications of the project was not located in a timely manner. As such, the performance changes in the LIDAR data frequency from the operating system improvements, and subsequently the impact of the memory map on the existing OpenNI2DS325 driver were not successfully tested or demonstrated by the time of the report.

6.3 Summary

The initial work on the driver was laid out and approaches to how to improve performance was identified. It was identified from research that the use of a memory map to allocate memory space could have a significant improvement in the performance of a driver. From research, a testing parameter was defined in feedback frequency from the LIDAR camera. However, missing hardware at the end of the project and not enough time to develop a software-based solution led to theoretical work not being put into practice before the end of the project.

7 CONCLUSION

7.1 Introduction

At the end of the project, the completed work is compared against the initial aims and objectives set out for the project. A reflection of the tasks, as well as what was accomplished can be identified. In addition, what was not accomplished was investigated, and the reasons for incomplete work were analysed. As such, suggestion for where the project could have been improved as well as suggestions for work in the future follow in the conclusion of the project.

7.2 Aims vs objectives

At the end of the project, the final operating system and software solutions were expected to be much more efficient than the initial setup by having an increased framerate with a smaller memory footprint, due to the overall processing speed increase from the reduced size of the OS. It was also anticipated that real-time functionality will be available for all future functionality, enabling the development of a more accurate final satellite. The main objectives outlined to approach this goal were managed to varying degrees throughout the course of the project.

7.2.1 Selecting a suitable RTOS for the mission

In order to accomplish this task, research into requirements was used to determine important parameters in which to select a range of suitable operating systems that could be benchmarked and compared. The same objectives were used to determine the benchmarks that would be performed in order to numerically analyse the kernels. Using sysbench, a variety of Linux-based kernels for the Raspberry Pi were tested.

Graphing and analysing the numerical outputs made it possible to locate a trend in operation. The RT Pre-emptible kernel developed by Emlid proved to be very reliable in mutex benchmarking tests, and performed average or better in most other critical tests. As such, a suitable ROTS operating system was located for the AAReST mission, expected to improve performance and reliability in future development.

7.2.2 Optimize the operating system for the task

After benchmarking tests were used to successfully find a suitable new kernel, the source code was modified in order to reduce the size further. By recompiling the source code without drivers that were not critical to the AAReST mission, the available RAM and overall response time was expected to be improved.

Returning to the specifications researched, it was possible to generally determine what device drivers could be removed from the default installation setup. By modifying makefiles during compilation, a new system image file was ultimately created with significant improvements in performance when compared to the original kernel. At the same time, the expected stable behaviour mostly scaled down to a reasonable rate, ensuring that the qualities that the kernel was initially selected for were not lost in the process of reducing the size.

7.2.3 Develop kernel-level software to complement existing code

For the next task, the requirements and possible approaches were outlined and research, and discussion of a wrapper for the OpenNI2DS325 driver surfaced both as a way to improve the performance of LIDAR feedback and processing, as well as allow for future telemetry commands to be handled. Before this was done and designed; initial work was aimed at implementing a memory map in order to allocate the necessary memory for the LIDAR driver directly

Due to missing hardware and not enough time left in the project, it was not possible to proceed very far with this stage. It was expected that the input frequency could be used as a benchmark for performance, and this would be improved to some degree by the existing custom software improvements. However, this could not be tested within the time of the project.

7.2.4 Perform tests on final product

At several points in development, benchmarking tests were performed to compare the change in performance. Due to a lack of essential hardware, a full test with the LIDAR camera was not possible within the time frame of the project and the overall performance changes could not be compared. However, the tests done provide several points of comparison.

As driver modifications were not completed to a sufficient degree, any final camera and communications tests would have to be performed after completing the modifications involved. This was initially set as part of the project, but as the driver work was incomplete and no LIDAR camera was available, work could not be completed in this aim.

7.3 Achievements

For the objectives related to locating and modifying an operating system better suited for the AAReST mission, the aims and objectives set out were met. Initial research yielded several operating systems as well as a series of requirements related to the AAReST mission in particular. As such, a final selection of operating systems for benchmarking were successfully identified. A narrowed down selection of Linux-based operating systems was found, with a variety of real-time patches applied.

In addition to providing parameters to identify a selection of operating systems, the research highlighted what requirements these should be benchmarked by. A series of benchmarking tests could be devised in SysBench and used throughout the project to demonstrate comparative performance of both different operating systems, as different stages of development. Once devised, the existing operating systems could be benchmarked and compared in order to select one based on performance. This being a kernel created by Emlid.

From compiling the source, a new image was compiled from the Emlid kernel. This was generated through the use of makefiles in order to remove drivers not needed within the scope of the AAReST mission. Once a new image had been compiled, comparison against the original kernel by use of the previous benchmarking data demonstrated the significant improvements in response time for all tests. This image should help to improve performance in a variety of aspects, including the LIDAR performance.

Initial work was likewise completed on methods to improve the frequency of data form the LIDAR camera further. The initial work was decided to be through applying a memory map to allocate memory as required, but was not implemented or demonstrated to a satisfactory extent. The initial research and suggested test parameter provide a point for future work to continue from.

7.4 Project evaluation

At the end of the project, much of the expected work on operating system research and modification had been completed, yielding data for reference as well as a customized image that improved on an existing operating system. By means of the better-suited operating system with drivers removed freed up more space for mission-critical software.

However, not everything was completed as expected, especially in regards to the driver development. More work could have been completed on these objectives aims could have been met faster and more risks could have been planned for. The lack of camera for testing was not an expected risk at earlier points in the project, and a software solution could have been researched and prepared earlier if the eventuality had been considered. Having a clearer set of goals at the beginning of the project could have helped with the amount of work completed.

Future developments

- *Benchmark new OS for LIDAR performance:* By testing the new image with LIDAR setup and comparing the results with the default image install, a comparison of any performance improvements in the LIDAR workflow can be demonstrated, and compared with future developments in optimisation.

- *Modify the LIDAR driver:* With an existing parameter to compare against, work on implementing a memory map in the existing driver could result in further improvements in performance.

- *Additional improvements and control by use of a wrapper:* By writing a wrapper function for the driver, more control for the overall performance can be applied. In addition, this could provide the framework for future telemetry control commands.

8 REFERENCES

[1] Raspberry Pi Foundation. *What is a Raspberry Pi.* https://www.raspberrypi.org/help/what-is-a-raspberrypi/ [Online: accessed 06-11-2015].

[2] C. Underwood, S. Pellegrino, V. Lappas, C. Bridges, B. Taylor, S. Chhaniyara, T. Theodorou, P. Shaw, M. Arya, and J. Breckinridge, "*Autonomous Assembly of a Reconfigurable Space Telescope (AAReST)-A CubeSat/Microsatellite Based Technology Demonstrator*"

[3] Underwood, C. Pellegrino, S. Lappas, V. Bridges, C. Baker, J. (2015) *"Using CubeSat/micro-satellite technology to demonstrate the Autonomous Assembly of a Reconfigurable Space Telescope (AAReST)*", Acta Astronautica, 114 (2015) 112-122, pp 2-4.

[4] Richard DUKE. "*Demonstration of a Low Cost C.O.T.S. Rendezvous Depth Sensor for Satellite Proximity Operations*".

[5] Liu, Y. Yue, Y. and Guo, L. (2011) "*UNIX Operating System, The Development Tutorial via UNIX Kernel Services*". Beijing: Higher Education Press.

[6] Todino, G. Peek, J. and Strang, J. (2001) "*Learning the Unix Operating System, 5th Edition*". O'Reilly Media.

[7] "*What is Linux"* (2016) Available at: https://www.linux.com/what-is-linux (Accessed: 01-05- 2016).

[8] Corbet, J, Rubini, A. Kroah-Hartman, G. (2005) "*Linux Device Drivers. 3rd Edition*". Available at: http://www.oreilly.com/openbook/linuxdrive3/book/ (Downloaded: 21/11/2015).

[9] Raspberry Pi Foundation. Raspbian (2016) https://www.raspberrypi.org/downloads/raspbian/ (Accessed 18-12-2015)

[10] National Instruments. "*What is a Real-Time Operating System (RTOS)?*". http://www.ni.com/whitepaper/3938/en/ [Online: accessed 06-11-2015].

[11] Raspbian. About Raspbian. https://www.raspbian.org/RaspbianAbout [Online: accessed 11-01-2016].

[12] Xenomai. About Xenomai. https://xenomai.org/about-xenomai/ [Online: accessed 18-12-2015].

[13] Emlid Limited. Real-time Linux for RPi2. https://docs.emlid.com/navio/Downloads/Real-time-Linux-RPi2/ [Online: accessed 17-12-2015].

[14] Real Time Engineers Ltd. FreeRTOS. http://www.freertos.org/ [Online: accessed 11-01-2016].

[15] Kubos. 'KubOS' http://www.kubos.co/kubos/ [Online: accessed 09-01-2016].

[16] Raspberry Pi Foundation. Raspberry Pi USBDocumentation.

https://www.raspberrypi.org/documentation/hardware/raspberrypi/usb/README.md [Online: accessed 0901-2016].

[17] Minibian. About. https://minibianpi.wordpress.com/about/ [Online: accessed 01-05-2016].

[18] A. Kopytov, "Sysbench manual, "https://github.com/nuodb/sysbench/blob/master/doc/manual.xml/".

[19] Corbet, J, Rubini, A. Kroah-Hartman, G. (2005) "*Linux Device Drivers. 3rd Edition*". Available at: http://www.oreilly.com/openbook/linuxdrive3/book/ (Downloaded: 21/11/2015), pp. 109-114.

[20] Raspberry Pi Foundation. Download Raspbian. https://www.raspberrypi.org/downloads/raspbian/ [Online: accessed 11-01-2016].

[21] linux-rt-rpi . https://github.com/emlid/linux-rt-rpi [Online: accessed 30th of April 2016]

[22] Corbet, J, Rubini, A. Kroah-Hartman, G. (2005) "*Linux Device Drivers. 3rd Edition*". Available at: http://www.oreilly.com/openbook/linuxdrive3/book/ (Downloaded: 21/11/2015), pp. 412-422.

[23] SoftKinetic. DepthSense® 525. http://www.softkinetic.com/Store/ProductID/36 [Online: accessed 18-03-2016]

9 APPENDIX 1 – REMOVED DRIVERS

9.1 Summary

This appendix contains all of the drivers and driver categories removed through the testing phases. Contained in the list is the location of each removed item or category removed within the scope of this project.

9.2 Full list of removed drivers

- Networking support -> bluetooth subsystem support
- Networking support -> NFC subsystem support
- Device drivers -> block devices -> Packet writing on CD/DVD
- Device Drivers -> Misc devices (Ensure only videocore is there)
- Device Drivers -> Input device support
- Device Drivers -> Input device support - HardwareI/O ports -> Gameport support
- Device Drivers -> Multimedia Support -> Analog TV

- Device Drivers -> Multimedia Support -> Digital TV
- Device Drivers -> Multimedia Support -> Radio receivers/transmitters
- Device Drivers -> Multimedia Support -> Remote controller support
- Device Drivers -> Graphics support -> Backlight & LCD device support
- Device Drivers -> Bootup logo
- Device drivers -> Sound card support
- Device drivers -> USB support -> USB Monitor
- Device drivers -> USB support -> USB Printer support
- Device drivers -> USB support -> USB Modem support
- Device drivers -> USB support -> EMI 6|2 USB Audio interface
- Device drivers -> USB support -> EMI 2|6 USB Audio interface
- Device drivers -> USB support -> 7-segment display
- Device drivers -> USB support -> Diamond Rio500
- Device drivers -> USB support -> Lego infrared tower
- Device drivers -> USB support -> LCD driver
- Device drivers -> USB support -> LED driver
- Device drivers -> USB support -> Cypress USB thermometer
- Device drivers -> USB support -> Mouse fingerprint sensor
- Device drivers -> USB support -> CardBus adapter
- Device drivers -> USB support -> Apple Cinema display
- Device drivers -> USB support -> USB LD driver
- Device drivers -> USB support -> Playstation 2 Trance Vibrator
- Device drivers -> USB support -> IO Warrior
- Device drivers -> USB support -> iSight firmware
- Device drivers -> Auxiliary Display support

10 APPENDIX 2 – FULL TEST DATA

10.1 Introduction

In this appendix, the full data from benchmarking is laid out for reference. Of note is how the new kernel performs with comparable stability to the original kernel from Emlid, but at a lower overall response time in most tests.

10.2 Numerical data

	Threads				**Mutex**			
	Response time (ms)				Response time (ms)			
	Min	Avg	Max	Appx. 95th	Min	Avg	Max	Appx. 95th
Raspbian Wheezy 3.10 (Calibration)	3.05	3.25	4.97	3.31	1435.97	2625.33	2951.54	2932.34
Wheezy w. Xenomai kernel	2.21	2.25	6.37	2.31	1638.55	2457.44	2878.12	2752.04
Minibian	2.62	2.67	3.97	2.73	1379.09	2346.61	2704.73	2683.7
Minibian w. Xenomai kernel	2.58	2.67	7.84	2.74	1651.25	2549.56	2888.34	2869.82
Wheezy w. Emild's kernel	5.26	5.74	9.78	5.88	11938.78	12303.93	12477.71	12470.99
Custom minimal Emlid kernel	3.97	4.27	7.62	4.46	2449.9	2818.07	3115.82	3113.26

Figure 17. Full test data of threads/mutex tests

	Raspberry Pi main				
		Response time (ms)			
	Requests/sec	Min	Avg	Max	Appx. 95th
Raspbian Wheezy 3.10 (Calibration)	4.17	0.06	0.15	9.6	0.25
Wheezy w. Xenomai kernel	4.97	0.1	701.61	15995.19	4641.27
Minibian	4.3	0.08	1037.36	13664.57	6691.11
Minibian w. Xenomai kernel	4.91	0.11	761.55	12193.64	4545.03
Wheezy w. Emild's kernel	4.24	0.17	902.1	13879.28	5761
Custom minimal Emlid kernel	4.15	0.08	0.65	56.48	0.48

Figure 18. Full test data of FileIO at rood

	USB files				
	Response time (ms)				
	Requests/sec	Min	Avg	Max	Appx. 95th
Raspbian Wheezy 3.10 (Calibration)	1.86	0.08	1241.17	22960.22	9423.66
Wheezy w. Xenomai kernel	1.81	0.08	932.66	18826.7	6761.58
Minibian	1.96	0.09	727.37	14849.84	6743.39
Minibian w. Xenomai kernel	1.84	0.1	981.32	20263.25	6709.16
Wheezy w. Emild's kernel	1.6	0.24	825.67	14870.25	4103.99
Custom minimal Emlid kernel	1.98	0.08	271.62	18875.95	3.95

Figure 19. Full test data of FileIO on USB

	CPU (Primes)			
	Response time (ms)			
	Min	Avg	Max	Appx. 95th
Raspbian Wheezy 3.10 (Calibration)	6.3	6.38	8.19	6.41
Wheezy w. Xenomai kernel	6.29	6.38	11.71	6.45
Minibian	5.35	5.43	11.78	5.48
Minibian w. Xenomai kernel	5.34	5.41	10.78	5.45
Wheezy w. Emild's kernel	9.38	9.65	16.09	9.75
Custom minimal Emlid kernel	5.34	5.55	13.54	5.68

Figure 20. Full test data of CPU

10.3 Comparative data

	Threads		Mutex	
	Response time (ms)			
	Min-max	Avg-95th	Min-max	Avg-95th
Raspbian Wheezy 3.10 (Calibration)	1.92	0.06	1515.57	307.01
Wheezy w. Xenomai kernel	4.16	0.06	1239.57	294.6
Minibian	4.16	0.06	1325.64	337.09
Minibian w. Xenomai kernel	1.35	0.06	1237.09	320.26
Wheezy w. Emild's kernel	5.26	0.07	538.93	167.06
Custom minimal Emlid kernel	3.65	0.19	665.92	295.19

Figure 21. Full comparative data for threads and mutex tests

	Root		**USB**	
	Response time (ms)			
	Min-max	Avg-95th	Min-max	Avg-95th
Raspbian Wheezy 3.10 (Calibration)	9.54	0.1	22960.14	8182.49
Wheezy w. Xenomai kernel	15995.09	3939.66	18826.62	5828.92
Minibian	15995.09	3939.66	18826.62	5828.92
Minibian w. Xenomai kernel	13664.49	5653.75	14849.75	6016.02
Wheezy w. Emild's kernel	12193.53	3783.48	20263.15	5727.84
Custom minimal Emlid kernel	56.4	0.17	18875.87	267.67

Figure 22. Full comparative data for FileIO tests at root and USB

	CPU (Primes)	
	Response time (ms)	
	Min-max	Avg-95th
Raspbian Wheezy 3.10 (Calibration)	0.230769231	0.004680187
Wheezy w. Xenomai kernel	0.462852263	0.010852713
Minibian	0.462852263	0.010852713
Minibian w. Xenomai kernel	0.545840407	0.009124088
Wheezy w. Emild's kernel	0.504638219	0.00733945
Custom minimal Emlid kernel	0.605612999	0.022887324

Figure 23. Full comparative data of CPU tests