An Android real-time kernel and system interface for open nano-satellite constellations

Bachelor thesis by:

Marc Marí Barceló

As part of the Bachelor Degree in Informatics Engineering
in the specialization on Computer Engineering

Advisors:
Elisenda Bou Balust
Carles Araguz López

Academic tutor:
Prof. Xavier Martorell Bofill

Facultat d'Informàtica de Barcelona
Universitat Politècnica de Catalunya - UPC BarcelonaTech

Barcelona, June 2016

The #1 programmer excuse for legitimately slacking off:
“My code’s compiling”

RANDALL MUNROE - XKCD 303

Abstract

Satellite development is moving from big and expensive satellites to small and low price nano-satellites based on commercial off the shelf components. These nano-satellites have allowed new universities, companies and hobbyists to access the space.

In this context, the Android Beyond the Stratosphere project pretends to develop a hardware and software platform for nano-satellites that is open-source, standardized, modular and low cost, based on commercial components and open standards. The project is based on an Android mobile phone, connected to an Arduino board for experiment deployment.

The aim of this thesis is to design and implement part of the software architecture for the Android Beyond the Stratosphere project. On one side, the Android Real Time Operating System, which will allow to run the software architecture reliably. On the other side, the System Data Bus, a flexible, modular and robust message distributor that is core to the rest of the software architecture.

Resumen

El desarrollo de satélites está cambiando de satélites grandes y de alto coste a nano-satélites, pequeños y de bajo coste, y basados en componentes comerciales. Estos nano-satélites han permitido el acceso al espacio a nuevas universidades, empresas y aficionados.

En este contexto, el proyecto Android Beyond the Stratosphere pretende desarrollar una plataforma software y hardware para nano-satélites que sea abierta, estándar, modular y de bajo coste, basada en componentes comerciales y estándares abiertos. El proyecto se basa en un teléfono móvil con Android, conectado a una placa Arduino, donde se colocan los experimentos.

El objetivo de este trabajo es diseñar e implementar parte de la arquitectura de software para el proyecto Android Beyond the Stratosphere. Por un lado, un sistema operativo en tiempo real basado en Android, que permitirá el funcionamiento de la arquitectura de software de forma fiable. Por otro lado, el System Data Bus, un repartidor de mensajes flexible, modular y robusto que sirva como núcleo del resto de la arquitectura de software.

Resum

El desenvolupament de satèl·lits està canviant de satèl·lits grans i d'alt cost a nano-satèl·lits, petits i de baix cost, i basats en components comercials. Aquests nano-satèl·lits han permès l'accés a l'espai a noves universitats, empreses i aficionats.

En aquest context, el projecte Android Beyond the Stratosphere pretèn desenvolupar una plataforma software i hardware per a nano-satèl·lits que sigui oberta, estàndard, modular i de baix cost, basada en components comercials i estàndards oberts. El projecte es basa en un telèfon mòbil amb Android, connectat a una placa Arduino, a on es col·loquen els experiments.

L'objectiu d'aquest treball és dissenyar i implementar part de l'arquitectura de software per al projecte Android Beyond the Stratosphere. Per un costat, un sistema operatiu en temps real basat en Android, que permetrà el funcionament de l'arquitectura de software de forma fiable. Per altre costat, el System Data Bus, un repartidor de missatges flexible, modular i robust que serveixi com a nucli de la resta de l'arquitectura de software.

Acknowledgements

First, I would like to show my gratitude to Prof. Eduard Alarcón and Elisenda Bou. They gave me the opportunity to join the Android Beyond the Stratosphere project from the very begining.

I want to show my appreciation specially to Carles Araguz, with whom I have been working for more than three years. Side by side we developed this and the ^{3}Cat-1 software architectures, learning from each other. Side by side, both of us have grown our knowledge and experience.

Thanks to Prof. Xavier Martorell, academic tutor, who adopted both the project and this thesis as one of his own and contributed with his own expertise. And also to Arnau Prat, who contributed in various areas inside the Android Beyond the Stratosphere project, and helped with the first important steps in the kernel development.

My gratitude to Prof. Adriano Camps, father of the NanoSat Lab, and all the people at the NanoSat Lab. My journey up to this point started with them. They are a fantastic team, both in the personal and professional levels. All together we made possible the projects that have come out from this lab. By their side, I learnt how to work in a team, and I learnt about their work, even if it was completely unrelated with mine.

Last, but not least, thanks to all my family. My parents, my grandmother, my brothers and my uncle. All of them supported me in all my decisions. Even if they were worried or unsure, they did not try to stop me. Their advice has always been helpful.

Thanks to everybody, the ones that are mentioned, and the ones that are not. It is because of them that arrived where I am now. And I could not even dream that it would be so high. Thanks again.

Contents

List of Figures

List of Tables

List of listings

1. Android Beyond the Stratosphere project

1.1 Introduction

Computing systems have evolved from very big and monolithic systems in the past, to small and standardized systems in the present. And they are currently moving towards distributed systems with many cores. Satellite missions have evolved in a very similar way. Traditionally, satellites were big and only available to big corporations, but nowadays, trends are moving towards small and standardized systems that can be deployed into multi-node architectures.

Like in computing, these satellite nodes have become more and more affordable, and have introduced new actors in the aerospace industry, such as universities, small companies, or even hobbyists. This evolution represents a democratization of the space.

These satellite nodes are usually nano-satellites. A nano-satellite is a small satellite of little more than 1 kg.

Mini-satellite	100 kg to 500 kg
Micro-satellite	10 kg to 100 kg
Nano-satellite	1 kg to 10 kg
Pico-satellite	100 g to 1 kg
Femto-satellite	10 g to 100 g

Table 1.1: Miniature satellite classification

In 1999, California Polytechnic State University and Stanford University developed a miniature satellite specification: the CubeSat program. This standard defines a 1U CubeSat as a cube-shaped nano-satellite, with 10 cm per side and no more than 1.33 kg [1, p. 5]. Since then, hundreds of universities throughout the world, such as UPC [2], Aalborg University [3] or EPFL [4] have developed satellites following this specification, ultimately becoming a *de facto* standard for low-cost small spacecraft. But not only universities have adopted this standard. There is an increasing number of companies whose activities are focused on the development of nano-satellite platforms and technologies, such as Planet Labs [5], Tyvak [6] or ISIS [7].

The popularization of nano-satellites have come hand-to-hand with the evolution in commercial off the shelf components (COTS). These components have become smaller, more power efficient, more generic, more standardized, and cheaper.

Traditionally, satellites were developed and launched by large corporations or space agencies. This was caused by the long development cycles and, more importantly, the high cost. Missions like GOCE, from ESA, had a total cost of €350 million [8], or TDRS-K, from NASA, had a cost of $450 million [9]. On the other hand, nano-satellite missions such as the ones at University of Surrey had a cost of just around €2 million [10].

There are two other important aspects of nano-satellites that are very important. On one side, they are usually launched in the spare capacity of larger missions, reducing even more the cost and risk. On the other side, they are payload-oriented, which means, the main purpose of the mission is the payloads or experiments in the nano-satellite, which may be an infrared camera, a geiger sensor, or an spectrometer, just to mention a few examples.

But, due to their smaller size, nano-satellites have some inherent problems, as stated in [11]. The most important one is the limitation in power, which has an effect specially in the communication link. This is why more and more nano-satellite missions have on-board Mission Planning Systems (MPS) to manage on-board resources such as power more efficiently. These MPS can also handle contingencies quicker and more intelligently to increase the robustness and resiliency of the mission to unexpected faults. This autonomy may even increase the performance of the system if the MPS can also detect experiment opportunities.

The low cost and low risk of nano-satellites, combined with the shorter development cycles due to its smaller size, makes nano-satellites ideal candidates for distributed space missions. These distributed missions usually fall in one of three categories [12]:

- **Fractionated satellites**: These are satellites whose systems are located in different modules, so it is required to share resources (power, data processing, communication link...) to operate.
- **Satellite swarms** or constellations: The nodes in the infrastructure are independent satellites that do not exchange resources or collaborate.
- **Federated Satellite Systems**: Independent satellites use their underutilized resources for a distributed mission [13].

Several corporations have started to offer and plan product lines based on satellite constellations, such as O3B [14], which is using a constellation of 12 satellites for communication purposes, OneWeb [15], using micro-satellites to provide worldwide Internet access, or Flock [16], with 28 nano-satellites for Earth imaging.

1.2 The Phonesat concept

Current smartphones share a lot of characteristics with what is expected on a nano-satellite. All the components on a smartphone are designed to be small and low power, while staying in a reasonable price.

Nano-satellites	**Smartphones**
On-Board Computer	System on Chip (SoC)
Data storage	Flash memory
Batteries	Batteries
Satellite health sensors	Temperature, voltage, state of charge and other sensors
Attitude determination system	Magnetometer (compass), gyroscope and accelerometer
Earth observation	Integrated camera

Table 1.2: Comparison of nano-satellite and smartphone components

For these reasons, the concept of reusing a smartphone to develop a nano-satellite is not a novel approach. In that sense, the STRaND-1 cubesat [17] demonstrated the use of a Nexus One smartphone as a payload of the nano-satellite. The Phonesat project [18] in its version 1.0 used a Nexus One smartphone as the on-board computer, and in the version 2.0 a Nexus S was used as a core processor as well as using of some of its sensors such as gyroscopes. These projects show the feasibility of operating an smartphone in the harsh conditions of space.

1.3 The Android Beyond the Stratosphere vision

The ABS project, performed at the Nano-Satellite and Payload Laboratory of the Technical University of Catalonia - UPC BarcelonaTech, is aimed to develop an open-source, standardized, modular and low-cost hardware and software nano-satellite platform based on commercial components and open standards.

This platform is based on two commercial systems: on one side, an Android smartphone, used as a processing core and as a sensing and communication subsystem. On the other hand, an Arduino board, an open hardware platform where payloads and experiments can be deployed. These commercial systems contain miniaturized and low-power components, and they are accessible and standardized.

The ABS platform is payload-oriented, being able to host any number of payloads of any kind and for any purpose. All these payloads are hosted on the Arduino board, following the Arduino standards of size or pinout, among others.

All these efforts are oriented towards creating an standard software and hardware platform, that anyone can use or modify to fit their needs. This standard platform, with commercial components reduces the cost to access space even further, allowing the access to new universities, hobbyists or companies, and moving an step further on the democratization of space.

The final goal of the ABS project is to have an Open Space Station (OSS), a distributed system where multiple satellites collaborate and share resources. Anyone can add a satellite to this OSS, or just upload a software experiment that benefits from the hardware of that constellation.

The current version of the ABS project is based on a Nexus 5 mobile phone [19], running Android 4.4 (codename KitKat) with an Arduino board connected through the built-in Micro USB port of the phone.

1.4 Software Architecture

This thesis is focused on the software aspects of this platform. The ABS project requires a modular software platform that can adapt to satellites of any size and also to distributed systems such as the OSS. To follow with the ABS vision, this platform must be ready to use for any kind of satellite, and must be open source and available to any developer that wishes to modify it.

The software architecture for the ABS project is based on the ^{3}Cat-1 software architecture documented in [11] and shown in figure 1.1. Unlike common flight software approaches, it aims for autonomy and reusability. This architecture was designed with 4 layers in mind: the System Core, the Process Manager, the System Data Bus and the Hardware Modules.

This design allows for modular addition of hardware components to the software architecture. To add a new hardware payload it would only be necessary to develop a Hardware Module and define the high level tasks and actions so it can be managed by the high-level scheduler.

The different layers are:

- The System Core (SysCore) is the top-level administrator of the system. Unlike the other layers, it is developed in the Prolog language. It is responsible for the task scheduling from a high-level perspective.
- The Process Manager (ProcMan) is the executor of the system. It translates the high level orders of the System Core into actions for the Hardware Modules. It is also in charge of monitoring and reacting to unexpected events.
- The System Data Bus (SDB) is the message deliverer. It distributes the actions generated by the Process Manager to the lower level, and the answers and events from the Hardware Modules back to the Process Manager.

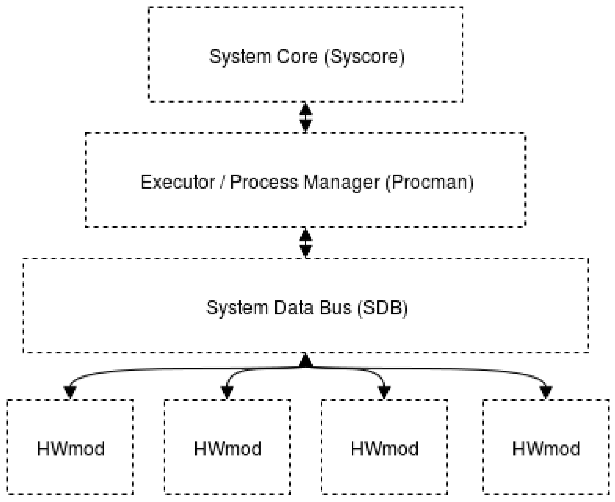


Figure 1.1: Software Architecture for the ^{3}Cat-1 project

- A Hardware Module (HWMod) is the layer between the software architecture and the real hardware. There can be as many as necessary, and each of them translates the System Data Bus messages into actions to the real hardware. This part is hardware specific and would need to be modified when the underlying hardware changes.

This design, oriented to modularity and extensibility, tries to improve system robustness by isolating failures and precluding their propagation through the architectural layers. It is also a very light architecture, having in mind the few components needed to ensure flexibility.

This software architecture for the ^{3}Cat-1 satellite was implemented by Carles Araguz (SysCore and ProcMan), Marc Marí, author of this thesis (SDB and ProcMan telemetry management), and other developers (HWmods). This implementation had some issues that need to be addressed:

- The commands that are used to interact with the SDB were changed during implementation phase, which produced very messy code in the SDB. This makes the addition of new commands very difficult.
- The SDB was not designed with testability in mind. This produced a very opaque layer, which made the debugging of the interaction between ProcMan and HWMods a very arduous job.
- The SDB, as its name says, should be used system-wide, not only between ProcMan and HWMods. This design flaw means that the SysCore lacks a direct communication channel with the HWMods, which could be needed to obtain sensor data for the scheduling process.
- The interface between the HWMods and the SDB was implemented using software interrupts to notify pending actions. This led to having nested interrupts handlers in a single process. When it was decided to rewrite these libraries, the communication protocol was too dependent on that model to make an efficient alternative.

This software architecture has been extended and improved taking into account the issues detected while implementing the system for the ^{3}Cat-1 satellite, and also the particularities of the ABS system (figure 1.2). But it also tries to benefit as much as possible from the Android stack, which is extensively documented [20] and is supported by a large community of developers.

The following modifications from the ^{3}Cat-1 software architecture are applied:

- The System Core has to be updated to allow for new scheduling policies, as explained in [21].
- The Process Manager has to be slightly adapted for the new communication layers, but the functionality remains the same.

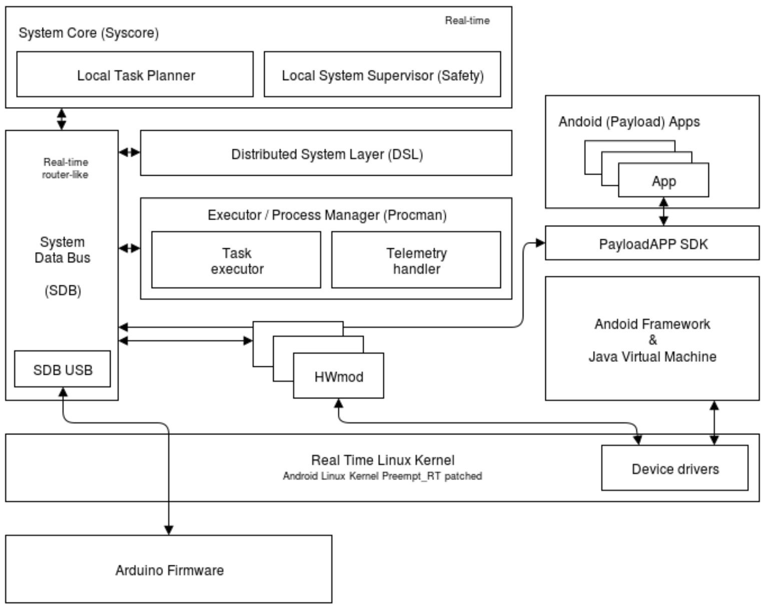


Figure 1.2: Software Architecture for the ABS project

- The System Data Bus has to be completely rebuilt. It is necessary to introduce modular command definitions that enable expansion and contraction of the system capabilities with ease and simplicity to fit any kind of satellite. Moreover, it has to be used between all system layers, including the USB subsection, to communicate with the Arduino board, and the Distributed System Layer, to communicate with other satellites in a constellation.
- The Hardware Modules follow the same structure and are supported both as a regular user-space process and as an Android app managed from the software architecture. They can perform operations and actions on the mobile phone, or on the Arduino platform.

1.4.1 Android Real-Time Operating System

As it can be seen in figure 1.2, the kernel underneath Android has been modified to enable real-time capabilites.

A real-time operating system (RTOS) is an OS that has a deterministic scheduling policy that guarantees a maximum latency, this is, the delay between the expected and the real start of an application. It is usually achieved by using priority based scheduling and preemption of low-priority tasks [22].

This OSs are essential in control systems such as in industry, medicine, or aerospace. In satellites, there are mission critical tasks such as flight control and sensor and environment control that must be run with deterministic latency and deterministic scheduling policy to prevent mission failure.

Most of currently running satellite missions have either a PIC microcontroller as an on-board computer, or a more powerful processor like a 32-bit ARM running a RTOS. The most common RTOSs for satellites are FreeRTOS [23], or a Linux distribution extended with Preempt RT [24] or Xenomai [25].

For example, the AAUSAT3 cubesat [26] is running an AT90CAN128 at 12 MHz and an ARM7 microcontroller at 60 MHz both with FreeRTOS. The ^{3}Cat-1 cubesat [11] is running an AT91SAM9G20 at 400 MHz with a Linux-based distribution extended with Xenomai for running mission-critical tasks.

Linux usage has been growing in the area of nano-satellites hand-to-hand with the increase on the number of COTS components used in those. Linux is already a mature development platform with millions of users around the world, in systems so different as high performance computers [27] or embedded systems [28].

This versatility is partially due to the high number of drivers for commercial systems of any kind. A major issue, though, is the lack of built-in real-time capabilities.

Due to its open-source nature, there are a number of projects developing RT extensions for the Linux kernel. The most important ones are Preempt RT [24] and Xenomai [25].

Android is also an operating system that has grown in popularity [29]. This OS is based on the Linux kernel, with several modifications oriented to be more power efficient and integrate better with mobile devices and the rest of the software components. The user space is developed in Java, on top of a special Java Virtual Machine called ART (the successor of Dalvik) [30]. The use of Java as a programming language has enabled the development of millions of programs or “apps” by developers all around the world.

Android is specifically designed for embedded devices, being mobile phones or tablets, or other devices such as TVs [31] or cars [32]. It is also open source, which makes easier to modify and adapt Android for all these projects. This flexibility has made it ideal for projects in all kinds of areas, such as robotics [33]. In [34], it is explained how a Nexus 5 smartphone running Android has been used in an environment usually reserved for RTOSs: unmanned aircraft systems (UAS). The results obtained in that project show that it is feasible to use the internal sensors of a Nexus 5 to control the the attitude of the system. But the latency uncertainties mentioned in the article are a sign of the necessity of transforming Android into a RTOS.

Introducing real-time capabilities in the Linux kernel below Android would result in an Android Real-Time Operating System (ARTOS), that can be used by anyone in areas where it is not possible to use Android yet, because of its lack of RT capabilities. And, of course, it would allow the ABS project to use an open and standardized platform with a huge community behind.

It is important to note that it would not be possible to reach this goal with other mobile platforms. For example, iOS [35] or Windows Phone [36] are not open source and would not allow for these kind of modifications.

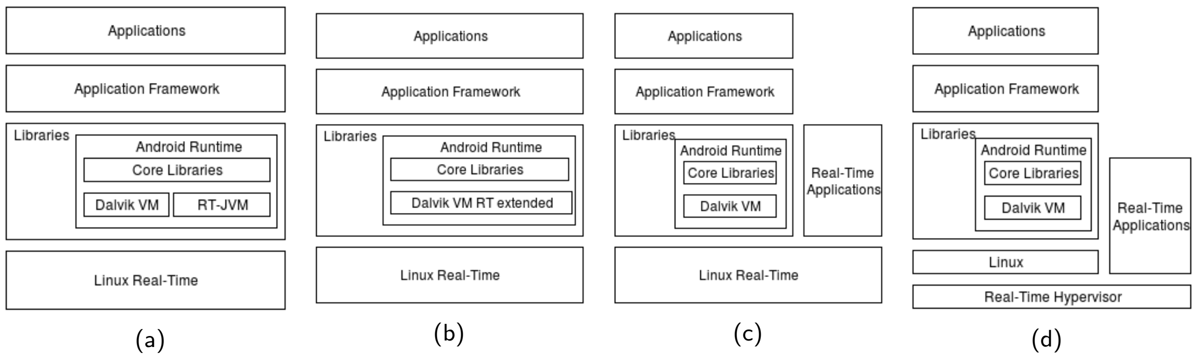


Figure 1.3: Directions to extend Android with real-time capabilities[1]

There are up to four different ways to enable a real-time environment in Android:

- The figure 1.3a: Extend Linux with real-time capabilities and add a real-time Java virtual machine next to Dalvik (Android-specific Java virtual machine).
- The figure 1.3b: Extend Linux with real-time capabilities and extend Dalvik with real-time capabilities.
- The figure 1.3c: Extend Linux with real-time capabilities and run real-time applications directly on top of the kernel.
- The figure 1.3d: Introduce a hypervisor and a real-time kernel to run next to the Linux kernel.

In the context of the ABS project, the first two options have been discarded because all the control software is developed in C directly on top of the kernel, so it can run faster and consume less resources. The other two options are analyzed in chapter 4.

[1]Extracted from [37]

Testing is a crucial part of the development of ARTOS, because it is the only way to check the real-time behaviour of a RTOS. [38] runs latency tests with different system loads on a Zoom Mobile Development Kit with Android. The results show inconsistent latency in every test performed, which means Android *as-is* is not prepared to run real-time applications.

[39] runs several tests to determine the real-time behaviour of a standard Android OS on a Beagleboard-XM development board. Different tests were performed, and the results showed important problems in the Android OS to behave as a RTOS without any modification. For the ARTOS that is explained in the following chapters, this test methodology is not ideal because using a highly documented development board instead of a real Android smartphone allows for better test methodologies. Furthermore, the first goal of the ARTOS implemented in the context of the ABS project is to have a contained latency, which is not analyzed in the article.

In [40] some improvements towards real-time were applied to the Android stack. This implementation corresponds to the implementation in figure 1.3b. The Linux kernel was extended using Preempt RT, and modifications in Dalvik are focused in memory management and garbage collection. These changes were tested on a HTC Dream smartphone. Latency tests show a significant improvement in time execution determinism. But these modifications were done on an Android 2 system, whereas this thesis is based on a more modern Android 4.4 system.

On the other hand, [41] uses an approach very similar to the one decided for the ABS project. It extends Linux with real-time capabilities using the Preempt RT patch, and runs the real-time applications directly on top of the kernel. The Android version in this case is 3.1, and the testing platform is a Motorola Xoom tablet. This work shows some problems that may appear while working on Android 4.4, such as version mismatches. The tests performed are mainly based on hardware interrupt latency, which is not the main goal for the ABS project.

1.4.2 System Data Bus

The System Data Bus (SDB) is the message distributor of the whole ABS software architecture, and therefore it has to be modular and flexible, so it can adapt to any nano-satellite and purpose, and it has to be open, so other developers can contribute and share their vision.

The Core Flight Executive (cFE) [42] by NASA is a complete framework that allows deployment of complex architectures, and which already has some predesigned components and services, including high-level application scheduling. The SDB, as the cFE, would allow the easy deployment of the ABS platform adapted with the particularities of each mission.

Given that the proposed SDB is envisioned as the core component of a modular software platform for nano-satellites it should pursue the system-wide qualities stated in [11]:

- **Modifiability**: The ABS projects aims to develop a generic platform suitable for any kind of mission. The SDB must be able to adapt to new commands and new rules that are mission-specific.
- **Portability**: This aspect is not as critical as the other ones, because the SDB runs on top of a Linux kernel which, by its nature, is already portable.
- **Robustness**: The SDB must be able to detect and recover from its own errors. But, as the component that delivers messages throughout the system, it should also ensure the correctness of the messages that pass through it. This avoids the propagation of errors from one component to another.
- **Computational efficiency**: It is very important to remember the environment where the SDB runs. All the resources in a nano-satellite are scarce and must be used rationally.
- **Autonomy**: The SDB is not the component responsible for the system autonomy. But the SDB is the responsible for sending data to and from the SysCore (the high level scheduler), so it is essential that it works as expected.

1.5 Scope and objectives

The scope of this thesis is the analysis and development of an Android real-time operating system, and the design and implementation of a system-wide communication layer, oriented to nano-satellites and nano-satellite constellations and based on the ^{3}Cat-1 System Data Bus.

Therefore, the objectives of this thesis are twofold:

- *Implement an Android real-time operating system* that is both stable as an OS, and with RT capabilities, the most important one being contained latency.
- *Design and implement the System Data Bus*, and the necessary libraries to interact with it. This SDB must be modifiable, robust and efficient.

2. Methodology

2.1 Android Real-Time Operating System

Different approaches are tested when developing the ARTOS, but all of them share the same methodology:

1. Apply patches to the base code. The patch mismatches are analyzed and corrected.
2. Compile. Solve any compilations errors and warnings, produced by the differences introduced from the patch.
3. Boot. Flash the resulting kernel on the mobile phone and boot. Solve the errors and warnings that appear, until a kernel that can boot the Android user space is obtained. This process is the most time consuming.

It is also very important to define a methodology to test the different results of the process. A valid ARTOS version that is able to load the user space must be tested both in stability and real-time behaviour. Stability of the system is tested with stress tests such as [43]. A working OS must support similar system loads without crashing or hanging.

On the other side, as it can be seen in [38] and in [40], real-time tests are based on calculating the latency of a real-time task (the already implemented cyclictest software [44] is used) over different conditions of periodicity, and load on the system.

If a system with a stable latency is achieved, other aspects of the real-time system should be tested as explained in [39], such as thread switching latency, semaphore timing and mutex behaviour.

When developing the ARTOS, there are multiple limitations and risks that can appear:

- Kernel debugging is a very difficult task because of its low-level nature. This will make this process long and difficult.
- The patches that are applied are complex patches that modify many aspects of the kernel and can introduce bugs that may pass undetected or may trigger an error far from the point where kernel is modified. This may make debugging very difficult.
- Android contains device-specific drivers that are not in the usual Linux kernel. These devices may or may not perform properly in real-time, and may need further modification and work.
- Android devices usually have closed-source binaries to control proprietary features such as CPU frequency, WiFi protocol, or others. These closed-source binaries may or may not perform properly in real-time, but there is no way to adapt them.

2.2 System Data Bus

The SDB specifications and implementations are shared with and reviewed by the rest of the ABS software team through public and collaborative software repositories such as GitHub.

The SDB, like any other component of the ABS software architecture has a set of minimal tests that are capable of testing little parts of it. Thus, the developers can ensure that their changes are not breaking any part of the system. This also helps in the debugging process, and increases the reliability of the final product. The tests developed check that every possible valid packet can be created and transmitted, and also check that system components can connect to the SDB and share messages.

This component also contains an internal Quality Of Service (QoS) logging and reporting interface. This allows all the packets in the system to be tracked for timing purposes and to detect and debug packet collisions. With this internal QoS reporting, the performance of the system can be tracked. The two single most important numbers for the SDB are obtained through this set of tools. These metrics are:

- **Packet processing time**: Time between a packet entering the SDB and the same packet or an answer to that packet exiting the SDB. This metric should be a time as small as possible.
- **Reliability**: Number of packets successfully processed versus number of packets dropped. This metric should be a percentage very close to 100%.

These metrics should be tested in a wide variety of system states, including different loads and bandwidths (number of packets sent to the SDB per unit of time).

Once the SDB is developed and tested in isolation, it is connected to the rest of components of the ABS software architecture, so it can be tested in nominal situations and in high-load situations, always analyzed through the QoS information.

Designing and developing the SDB has several risks and limitations:

- Finding a design problem at the end of the development process can require a redesign and reimplementation of a big part of the SDB.
- A design not oriented to debug and test may create a black box that makes the SDB unreliable.
- Inadequate QoS results may require to redesign of a big part of the SDB.

3. Project management

3.1 Planning

3.1.1 Introduction

The ABS project, including the work described on this thesis, began on September 2013. Work during this period of time has been irregular depending on the personal circumstances of the team members.

The total amount of work expected is around 1000 hours.

As the amount of work is low compared to the period of time, any delays that may occur in the planning can be overcome without further planning modification.

3.1.2 Resources

The resources are human and material:

- Human: the ABS project team working on software:
  - Software developer for ARTOS and the SDB (also thesis author)
  - Software developer for the Procman and Syscore (Carles Araguz)
  - Software developer for Android apps and the hardware controlling the payloads (Arnau Prat)
  - Other software developers with temporary collaborations in areas such as ARTOS research or the MCS generator, among others.
- Material: A Nexus 5 mobile phone, laptop, and a Raspberry Pi or other embedded ARM-based platform.

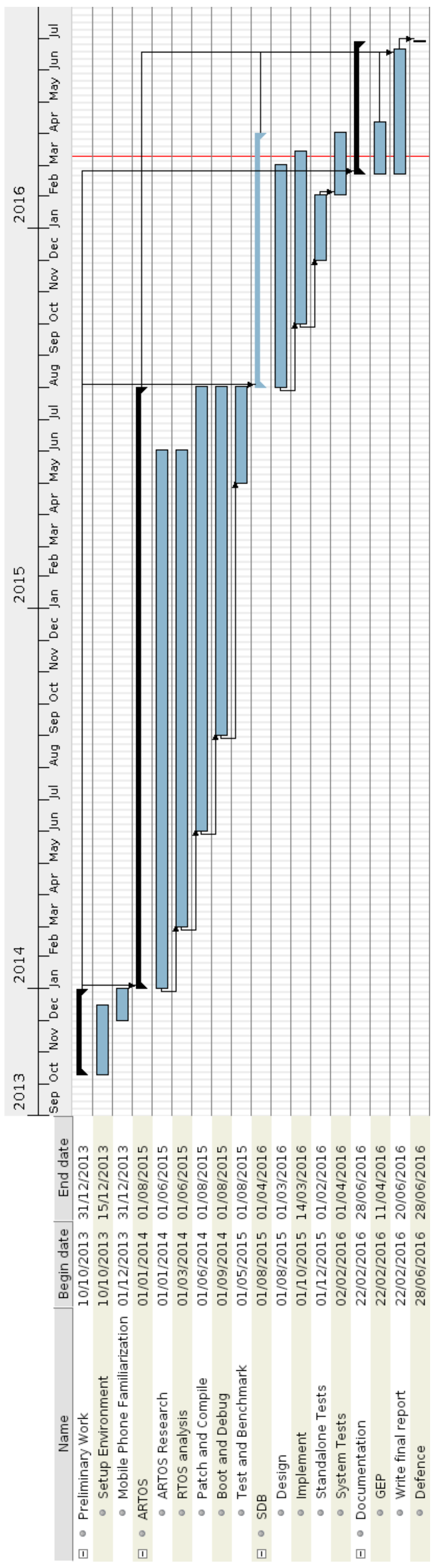
3.1.3 Tasks

Preliminary work

This group contains all the tasks that need to be done before the project really starts. These tasks are well documented, so no big delays are expected.

Setup Environment Collect and prepare the hardware and the software. This includes ordering the mobile phone and the necessary hardware interfaces to interact with it, as well as downloading and installing cross-compilers, source code repositories, and other utilities necessary to perform all the work.

This task will take around 2 months and the resources needed are the Nexus 5, the laptop, and the ARTOS developer.



Name	Begin date	End date
Preliminary Work	10/10/2013	31/12/2013
Setup Environment	10/10/2013	15/12/2013
Mobile Phone Familiarization	01/12/2013	31/12/2013
ARTOS	01/01/2014	01/08/2015
ARTOS Research	01/01/2014	01/06/2015
RTOS analysis	01/03/2014	01/06/2015
Patch and Compile	01/06/2014	01/08/2015
Boot and Debug	01/09/2014	01/08/2015
Test and Benchmark	01/05/2015	01/08/2015
SDB	01/08/2015	01/04/2016
Design	01/08/2015	01/03/2016
Implement	01/10/2015	14/03/2016
Standalone Tests	01/12/2015	01/02/2016
System Tests	02/02/2016	01/04/2016
Documentation	22/02/2016	28/06/2016
GEP	22/02/2016	11/04/2016
Write final report	22/02/2016	20/06/2016
Defence	28/06/2016	28/06/2016

Figure 3.1: Gantt diagram

Mobile phone familiarization When the mobile phone is available, this task will help to learn how to work with the mobile phone. This means learn how to flash a kernel, how to run applications directly on the kernel, how to reset the phone in case of a software mistake, how to access debug ports on the mobile phone...

This will take 1 month, and will start in parallel to the previous task when the Nexus 5 phone is delivered. The resources needed are a Nexus 5, a laptop, and the ARTOS developer.

ARTOS

This group contains all the tasks related to developing ARTOS, and it cannot start before the preliminary work is finished.

ARTOS Research Research in the area of real-time operating systems, the Android source code and Android as a real-time operating system. This includes an analysis of the work already done and the different possibilities to carry this work forward. This task will be done in combination with the rest of the ABS software team, in order to share ideas and knowledge.

The time for the task is 1 year and 6 months, because this research will continue even when some kernel is being implemented, so new ideas can be applied in the process. The resources needed are laptop and the whole ABS software team.

RTOS analysis Familiarization with the RTOSs that will be used on the next phases. This means to check differences and deviations from the standard versions of the kernels, and to test these RTOSs on standard embedded platforms such as Raspberry Pi.

The time allocated for this task is 1 year and 3 months, and this task cannot start if the previous task, the research, has not started. This task is overlapped with the rest because it may be necessary to check the correctness of procedures and changes inserted in a kernel in more standard conditions. But also, a new OS may be checked while performing the other tasks. The resources needed are laptop, embedded ARM platform and ARTOS developer.

Patch and compile Patch the selected kernel with the selected patch, and work on the modifications until it compiles without any errors. This process is done following the compiling trace, and comparing the errors indicated with the modifications introduced by the patch.

The time allocated for this task is 1 year and 2 months, and it cannot start until the previous task, RTOS analysis, has started. This task is overlapped with the rest because changes in patches in following tasks may introduce compilation errors. The resources needed are Nexus 5, laptop and ARTOS developer.

Boot and debug When the kernel can compile without any error, it will be flashed on the smartphone and booted. Any errors that appear in the boot log will be corrected, and the process repeated.

The time allocated for this task is 11 months, and it can only start after some kernel is patched and compiled. This task overlaps with the rest because kernels will be in debug mode even if the kernel can boot to the Android userspace, as errors can appear in runtime. The resources needed are Nexus 5, laptop and ARTOS developer.

Test and benchmark Once the prepared kernel can load the Android userspace, it is necessary to run stability tests and real-time tests. These tests will give a quantitative look on the performance of this kernel, in comparison with the original kernels.

This task will take 3 months, and it cannot start until there is a kernel that can boot, obtained in the previous task. The resources needed are Nexus 5, laptop and ARTOS developer.

SDB

This group contains all the tasks related to developing the SDB, and it cannot start before the preliminary work is finished. It could be combined with the development of ARTOS, but it will be implemented afterwards.

Design Design of the functionalities of the SDB, so it is modifiable, robust and efficient. This task will be done in collaboration with the rest of the ABS software team, as this component communicates the rest of the components of the system.

The time allocated for this task is 8 months, and it will overlap with the rest of the tasks because there are independent sections that can be designed while other sections are being implemented. But it is also important to revise the design as the implementation moves forward, in case for any design problem. The resources are laptop and the whole ABS team dedicated to software.

Implement Implementation of the SDB, following the design of the previous point. The difficulties found in this step will help to finish the design.

The time allocated for this task is 6.5 months, and it cannot start until part of the design is performed. This task is overlapped with the other tasks, because problems detected in testing will be solved through implementation, and design problems may be detected in this phase. The resources are laptop and SDB developer.

Standalone tests Run unit tests on the SDB, and correct the implementation errors.

The time allocated for this task is 2 months, and it cannot start until part of the implementation is performed. The resources are Nexus 5, laptop and SDB developer.

System tests Combine the SDB with the rest of the components of the system, and test their behaviour and integration. This task is done in collaboration with the rest of the ABS team dedicated to software, as these are system-wide tests.

The time allocated for this task is 2 months, and it cannot start until standalone tests are finished, as it makes no sense to test the whole system before the parts are tested individually. The resources are Nexus 5, laptop and the ABS team working on software.

Documentation

This group contains all the tasks related to writing this thesis. It is important to start after the preliminary work and research, and it will be carried during spring semester of 2016.

GEP Project management period.

The time for this task is 2 months from the initial to the final presentations. The resources needed are laptop and thesis author.

Write final report With all the results from the different parts of the project, and the work from GEP, the thesis will be written.

The time allocated for this task is 4 months, and this task cannot finish before all the technical work and GEP are finished. The resources needed are laptop and thesis author.

Defence One week after the thesis has been finished and submitted, the thesis will be defended, marking an end for the project.

This task takes only one day, and marks the end of the project. The resources needed are laptop and thesis author.

3.2 Cost evaluation

3.2.1 Introduction

The following table presents the estimated budget for the project. It also includes a 5% contingency margin for unexpected expenses.

Description	Cost (in €)
Human resources	10000
Hardware resources	338.61
Software resources	0
Other	50.68
Contingency	519.46
Total	**10908.75**

Table 3.1: Total costs

3.2.2 Human resources

The human resources considered for the budget are one junior software developer for the whole project, and one junior software developer and one PhD student for the team-sharing parts.

As mentioned in the planning, the project consists of 1000 hours. The team-sharing parts consist of 100 hours, because they are composed of periodical meetings.

Description	Hours	Cost per hour (in €)	Total cost (in €)
Junior software developer	1000	8	8000
Junior software developer	100	8	800
PhD student	100	12	1200
Total			**10000**

Table 3.2: Costs of human resources

3.2.3 Hardware resources

There are two kinds of hardware resources for this project. On one side, a Nexus 5 mobile phone and interfacing cables, which are bought specifically for this project. On the other side, a Raspberry Pi and laptop, which were bought before the beginning of the project, and will be used afterwards.

The Nexus 5 phone has a cost of 400 €, but only 300 € will be detailed in the costs because at the end of this project, the phone will be used in other projects until the end of its lifespan.

The laptop and Raspberry Pi costs are calculated as a function of its expected lifespan of 4 years.

Description	Hours	Cost (in €)	Total cost (in €)
Nexus 5		400	300
Interface cables		10	10
Raspberry Pi	100	40	0.11
Laptop	1000	1000	28.5
Total			**338.61**

Table 3.3: Costs of hardware resources

3.2.4 Software resources

The software used in this project is free software that has no associated cost.

3.2.5 General expenses

This section includes other costs not considered in previous sections.

The electricity cost considered for this project, will be of 0.12 €/kWh

Description	Hours	Power (W)	Total cost (in €)
Laptop	1000	374	44.88
Mobile phone	1000	44	5.28
Raspberry Pi	100	44	0.52
Total			**50.68**

Table 3.4: Costs of general expenses

Both the internet connection and the space to carry this project will be provided by the university free of charge, so they is not considered in this budget.

3.2.6 Budget control

Budget is basically dependent on the time specified in the Gantt diagram. To ensure budged compliance, the time planning must be followed. The table 3.5 helps to control the budget by detailing the budget for each of the tasks in the project.

Hardware failures would not modify the planning because they can be solved in a matter of hours, as the hardware used can be easily replaced.

Task	Human costs (in €)	Hardware costs (in €)	Other costs (in €)	Total cost (in €)
Preliminary work	727.2	312.59	4.56	1044.35
ARTOS	6105.6	16.53	29.39	6151.52
SDB	2196.8	6.05	10.63	2213.48
Documentation	969.6	3.43	6.08	979.11
Total	10000	338.61	50.68	**10389.29**

Table 3.5: Total costs per task

3.3 Sustainability

3.3.1 Economic sustainability

The budget described in the previous chapter is the minimum budget for this kind of project. Hardware components are kept to a minimum, and software is free.

There is a long time of development, which increases human resources costs, and also part of hardware costs. But the development time described in the planning is a reasonable time to keep budget tight, as well as giving enough time to get results from the research.

The project described in this document is part of the ABS project. The ABS project funds this project in exchange for the research results.

3.3.2 Social sustainability

The project described in this document allows lots of different areas to benefit from real-time OSs in a common embedded device such as a smartphone. This would allow the use of more user-friendly devices in critical areas, and reduce the learning curve for professionals in those areas.

On the other hand, the area of satellites is currently only available to space agencies or business. Nano-satellites allowed the access to space to smaller business, universities and very few hobbyists. But the ABS project makes access to space simpler and cheaper for any individual or organization, by offering an open, modular and reliable satellite platform. And the project also opens a platform, the Open Space Station, to run software experiments from anywhere and by anyone in the world. These results make science and technology more reachable worldwide.

3.3.3 Enviromental sustainability

All the hardware used in this project will be reused. The Nexus 5 phone and cable interfaces will be used by other members of the ABS team to test software or hardware components, whereas the laptop and Raspberry Pi will continue to be used in the personal projects of the developers. At the end of the life of these components, they will be disposed according to its composition.

Adding real-time capabilities to the Android OS can benefit the execution time and resource usage of applications running on this platform, increasing energy efficiency. This would reduce battery charging and electricity costs, reducing ecological footprint.

4. Android Real-Time Operating System

4.1 Platform

Research in the area of Android Real-Time Operating System is done based on the platform chosen for the ABS project, a Nexus 5 mobile phone.

The Nexus 5 mobile phone by LG Electronics contains a system on chip (SoC) Qualcomm Snapdragon 800. This SoC contains a quad-core 2.26 GHz processor [19]. In the source code for both Android and Linux, the Nexus 5 is codenamed "Hammerhead" [45], and the SoC is "MSM8974" [46]. This platform is not merged on the mainline Linux kernel, it is only present in the Android specific tree, in conjunction with the rest of MSM-based platforms [47].

The Android version used during the development is 4.4 (codename KitKat). For this processor and OS version, the Linux kernel version available is version 3.4.

As introduced in chapter 2, the methodology to develop this kernel is iterative:

1. Apply patch and analyze and correct mismatches.
2. Compile. Solve compilation errors and warnings and compile again.
3. Boot. Flash the kernel on the phone and boot. Solve runtime errors and boot again.

4.1.1 Patch

The first step is done using the tool `patch`[1]. A patch is a file with a list of changes in a file. Each change is described by a line number, a number of context lines before and after, and the lines that are removed or added. The output from `patch` shows where the mismatches are, and these are analyzed manually. A mismatch is produced, usually, by one these three causes:

- **Different line number**. If line numbers do not match between the original file and the target file, `patch` may still find the region, and apply the changes showing a warning with the line offset.
- **Changed context**. If there are small differences in the context of the change, `patch` may still be able to detect if that is the correct region. In this case, it applies the changes and shows a warning. Otherwise, the changes are not applied and it shows an error.
- **Cannot find section to change**. This produces an error, and the changes are not applied. This problem may be due to non-existing regions in the target file, changes in function parameters, or other issues.

Errors are reviewed manually, and a decision is taken for each issue. For example, if there is a section that has similar context, with a difference in indentation or with extra lines in the context, the changes on that section are applied. Or if the lines changed are function calls that changed parameters, the lines are changed manually, and the parameters are set to the target file parameters.

[1]Manual page: `http://linux.die.net/man/1/patch`.

4.1.2 Compiler

The Android Open Source Project (AOSP) [48] provides a compiler to build the kernels [49]. This compiler is a GCC compiler [50] configured for the specific necessities of Android kernels.

Errors in compilation are usually consequence of errors in the application of the patch, or incompatibilities between modified files and files new in the target. An example could be a driver only present in the target, that uses a header file that is modified by the patch. Like with patches, these errors are reviewed manually and corrected.

4.1.3 Flashing and booting

The compilation of a kernel generates a binary file, named `zImage-dtb`. This file is a compressed kernel image with a compiled device tree attached. A device tree file is structured file that describes the physical devices of the system [51].

To generate a bootable image, this kernel image is combined with a ramdisk image[2], and a command line using the `mkbootimg` script [53] provided by AOSP. The physical offsets for these components (table 4.1), a valid ramdisk image, and a valid command line can be extracted from an original boot image using the script `split_boot` [54].

Base address	0x00000000
Kernel offset	0x00000800
Second stage bootloader offset	0x00F00000
Tags offset	0x02700000
Ramdisk offset	0x02900000

Table 4.1: Physical memory offsets of the boot components for the Hammerhead platform

This bootable image can be flashed into the device using `fastboot` [55] through USB. When this new image boots, it is necessary to check if it can load the system, or if it contains errors. For this purpose, the Nexus 5 mobile phone comes with a built-in debug and serial port on the headphone [56]. If a voltage greater than 3 V is applied on the microphone entrance, left audio and right audio are converted into a serial debug port (table 4.2). This serial port is essential to modify and debug the kernel, because it is the only channel that can output the kernel boot traces. The cable used during development can be seen in figure 4.1.

Headphone	**Connector**[3]	**Serial**
Microphone	Sleeve	3.3 V
Ground	Ring 2	GND
Right audio	Ring 1	TX
Left audio	Tip	RX

Table 4.2: Pin assignments for Hammerhead headphone serial port

Kernels can usually be debugged remotely using the `kgdb` debugger [58] through a channel (which is usually a serial port). In the case of the kernel for the Hammerhead platform, it is not possible to enable `kgdb`.

[2]A basic filesystem used as a step to boot the whole system [52].

[3]4 pole connector or TRRS connector has 4 sections called (from tip to top) tip, ring 1, ring 2 and sleeve. And hence the name: Tip Ring Ring Sleeve connector [57]

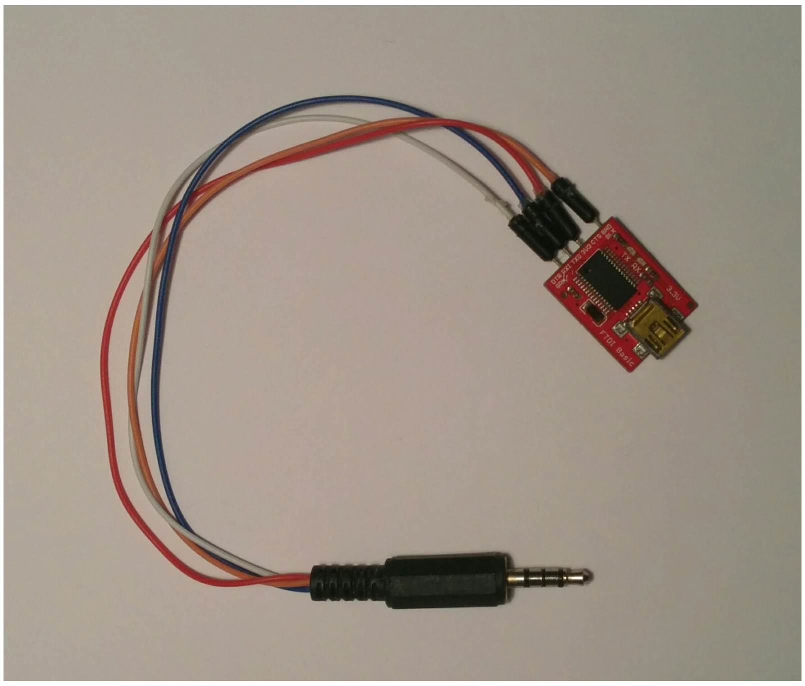


Figure 4.1: Headphone to UART cable

Enabling `kgdb` produces no effect, and enabling `kbd` (keyboard interface for `kgdb`) or running `kgdb` test suite causes a boot failure.

As the target hardware is not designed for easy debugging, the only way to debug the kernel is through code analysis and tracing with `printk` statements.

The different results from the boot stage depend on the kernel used and the patches applied.

4.2 Real-time patch alternatives

As explained in subsection 1.4.1, the two main options to generate an ARTOS considered in the context of the ABS project are:

- Modify the Linux kernel to enable real-time capabilities. This can be done with Preempt RT [24].
- Introduce a hypervisor and a real-time secondary kernel. The most common option for this kind of modification is Xenomai [25].

As it is shown in section 4.6, the real-time performance of Xenomai is better than Preempt RT. But modifying a kernel to enable Xenomai requires expert knowledge of the hardware platform [59]. Moreover, as it is explained in section 4.4, the interrupt handling in MSM platform is particular to this architecture, which would imply deep modifications in the Xenomai interrupt propagation.

Starting from version 3, Xenomai also offers a single kernel configuration, relying on the real-time capabilities of the kernel underneath (which may be modified with Preempt RT). This is not the behaviour expected, so this option has been discarded.

On the other hand, there is Preempt RT. It is a patch for the Linux kernel that introduces the features expected in a RTOS in Linux [60]: threaded interrupts, preemptible spinlocks, mutex priority inheritance, and high resolution timers, among others. To apply a Preempt RT patch, it is only necessary to have a

Linux kernel with a patch available for that specific version. For example, Linux version 3.4 has available patches.

If the kernel used is not a mainline Linux kernel, it may be necessary to adjust certain features to the architecture.

4.3 Kernel alternatives

4.3.1 Direct approach

The most obvious option to generate an ARTOS is to tackle the problem directly: apply the real-time patches on top of a Hammerhead kernel. This option is not trivial, as there is a big difference between a kernel for Hammerhead and the mainline Linux kernel:

- **Drivers**. Most of the hardware components of the Nexus 5 do not have a matching driver on the mainline Linux kernel. These drivers are only present in the Hammerhead specific kernel. Some examples are: camera, power subsystem, vibrator, CPU frequency scaling...
- **Device tree images**. Mainline Linux kernel does not generate kernel images with a compiled device tree attached (`zImage-dtb`).
- **Architecture-specific changes**. Changes in ARM architecture for Android-specific use cases or MSM platform particularities, such as in memory management.
- **General kernel changes**. There are several changes in the generic sections of the kernel, probably oriented towards a lower power consumption, or optimizations for the MSM platform. These changes include block devices core, memory management, workqueues, and scheduler core, among others.
- **Android specific drivers**. Android introduces some features in the kernel such as Ashmem[4] or the Binder[5] among others [61].

It is also possible to do it the other way around, by using a Hammerhead patch on top of a mainline Linux with real-time. But the sheer number of differences between Linux and Hammerhead makes this option very difficult.

4.3.2 Emulator

Because of the lack of debugging capabilities, the Android emulator [62] (codename Goldfish) is an option to develop ARTOS. This solution is not perfect, because, although Goldfish contains all Android-specific code, it lacks all machine-specific code, but the easier debugging may help test features and changes.

This option has been discarded because the Android emulator is based on QEMU, and QEMU lacks real-time emulation in its code translation mode[6]. If real-time cannot be emulated, it is not possible to test the result.

Nonetheless, a Goldfish kernel could be used as a base kernel to apply the patches, because it is halfway between an Android device kernel and a mainline Linux kernel. This option has been discarded because it requires the application of two patches: add real-time on Goldfish, and add all MSM specific code. This would increase significantly the amount of work needed.

[4]Android Shared Memory, similar to POSIX shared memory with a simpler file-based interface, and with support for discarding memory.
[5]Android-specifing interprocess communication mechanism, and remote procedure call system.
[6]In translation mode, QEMU translates the guest OS machine code to the host architecture machine code on the fly

4.3.3 Version porting

Another option would be to use a more modern Linux version, instead of version 3.4. This may allow the use of already merged features, to avoid adding them with patches.

This would require to combine patches for the kernel 3.4 with patches for a more modern kernel, or find patch alternatives for the chosen version.

Considering the different versions available, porting the kernel to a more modern version creates more problems than solutions.

4.4 Hammerhead and Preempt RT

As introduced in the previous sections, the alternative that may work better is to use a Hammerhead kernel and patch it with the Preempt RT patch.

This patch does not apply cleanly, due to the modification in the scheduler core, the workqueue, and the memory allocation system. With these sections corrected, and other minor changes, the kernel can boot (without real-time enabled).

If at this point `mpdecision` is disabled, this kernel can already run in Preempt RT base mode.

`mpdecision` is the binary responsible for adjusting CPU frequency on runtime. As it is distributed as a binary file, it cannot be modified. Moreover, CPU scaling is strongly discouraged when running in real-time environments [63]. For these reasons, `mpdecision` is left disabled.

Preempt RT has two working modes: RT base mode, and RT full mode [64]. RT base helps to debug RT full, and has some options disabled.

To obtain full real-time, more modifications are needed. Threaded interrupts introduce two major errors in Hammerhead drivers:

- The Nexus 5 contains GPIO multiplexers that rise hardware interrupts when there is an event. The interrupt handler in the kernel reads the GPIO multiplexer information and rises an interrupt with this new information. This last step is done using the same low-level wrappers that are used to process hardware interrupts. This is fine when interrupts are processed isolatedly, but, when interrupts run as a thread, this may cause unexpected behaviours. It is necessary to block external interrupts before processing this secondary interrupt.
- GPU register access is mutex-protected. If an interrupt needs access to one of these registers, it is not necessary to lock the mutex, because it is in interrupt mode. In case interrupts are threaded, it is necessary to lock the mutex before doing the operations.

Another problem related to full real-time is the use of spinlocks in critical areas. In Preempt RT, spinlocks are preemptible, which means they can sleep, so critical sections must use real spinlocks, called "raw spinlocks".

With all these changes, the mobile phone is still not able to load the full Android user space, but it is possible to connect and use the mobile phone through USB.

4.5 Tests

The testing tool used to measure latency is based on the cyclictest test released with the Xenomai source code [44]. This code has all Xenomai configuration checks removed, and the possibility to use the system default priority (for testing a standard Linux system).

Cyclictest measures the difference between the expected wake up time and the real wake up time of the function `clock_nanosleep` with a given sleep time. This difference is the latency necessary to wake up the task.

To stress the system, some artificial load is generated. The load is a process performing additions and subtractions. The number representing the load means the number of simultaneous processes of this type running in parallel to the test.

The results from the Nexus 5 mobile phone are compared with a more standard ARM embedded board, a Raspberry Pi model B [65]. This platform has a BCM2835 SoC with a 700 MHz ARMv6 uniprocessor, and runs a Linux version 3.8.13.

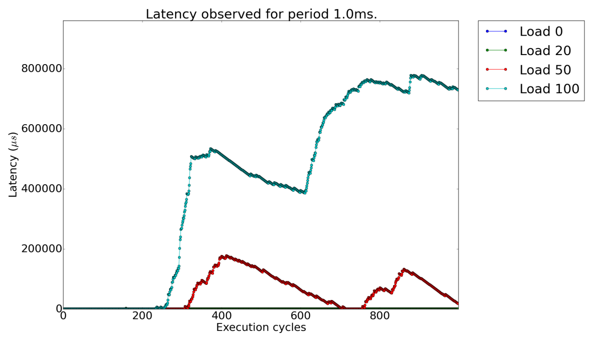
As they are platforms with very different computing power, the plots cannot be compared directly. For this reason, the Nexus 5 is tested with lower period and higher load than the Raspberry Pi.

4.6 Results

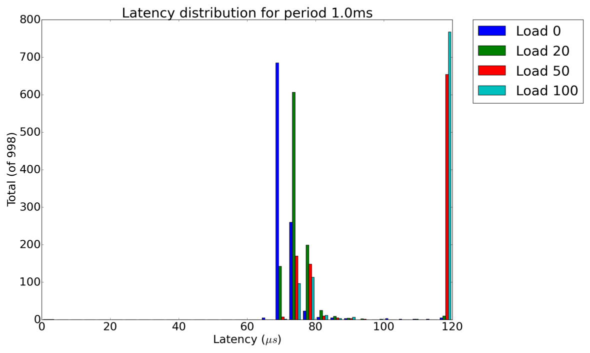
The variables that are explored in these test results are number of load processes, and sleep time (or period) for the cyclictest measures.

The data is presented in two formats: line plots and histograms. Line plots show the latency tendency and progression through time, whereas histograms help visualize the mean and extreme values. Each plot is run for a certain cyclictest period, and several load values, each represented with a different color.

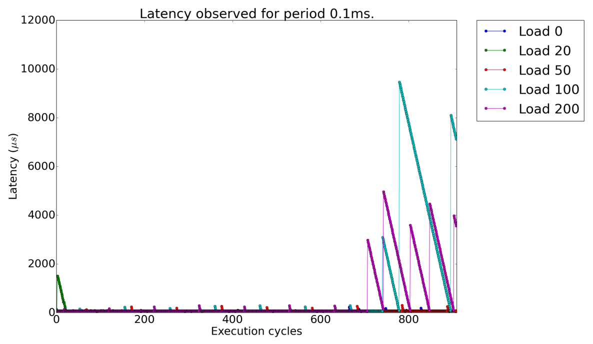
4.6.1 Non-real-time



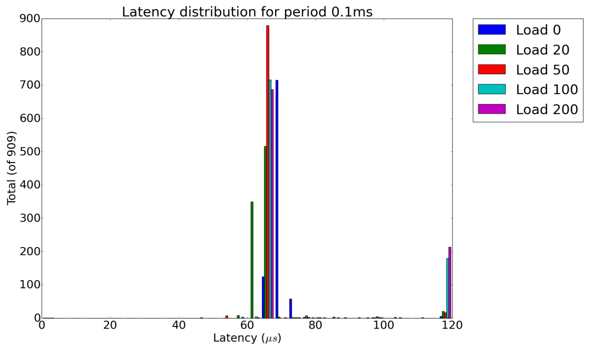
(a) Raspberry Pi line plot for period 1ms



(b) Raspberry Pi histogram for period 1ms



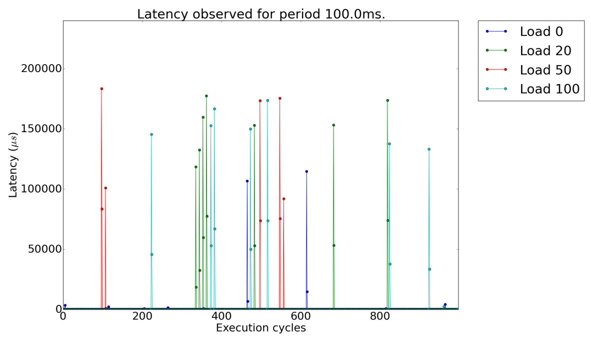
(c) Android line plot for period 0.1 ms



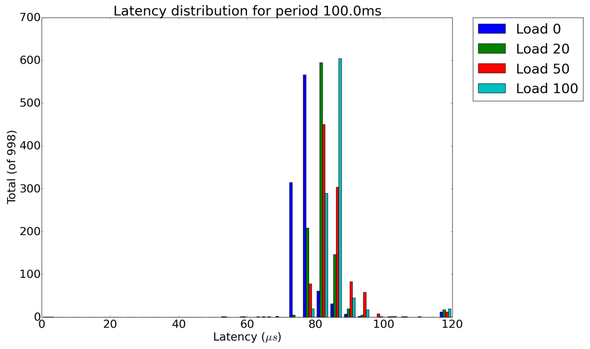
(d) Android histogram for period 0.1 ms

Figure 4.2: Latency measurements for a non-real-time system with small period

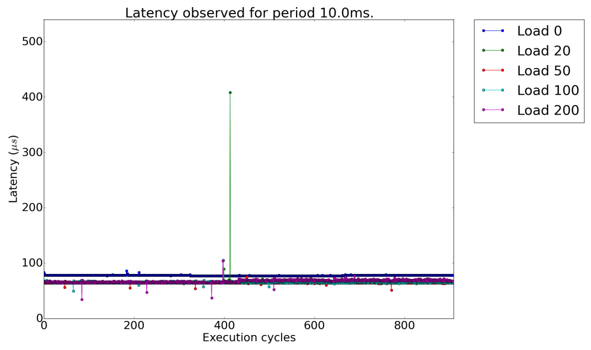
The figure 4.2 is a Linux system (without real-time enabled) with very low period. This produces a latency accumulation in both platforms when the load is high, because the system cannot cope anymore with both the deadlines and the load. In the line plots, it is not possible to see some of the loads, due to the high values of the other loads, but the histogram shows that these loads have values of around 70 μs in both platforms, with a few high latency peaks.



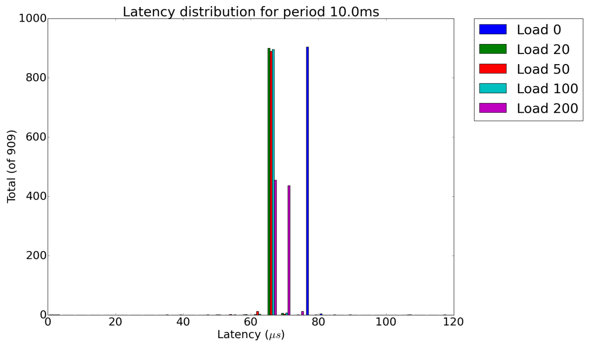
(a) Raspberry Pi line plot for period 100 ms



(b) Raspberry Pi histogram for period 100 ms



(c) Android line plot for period 10 ms

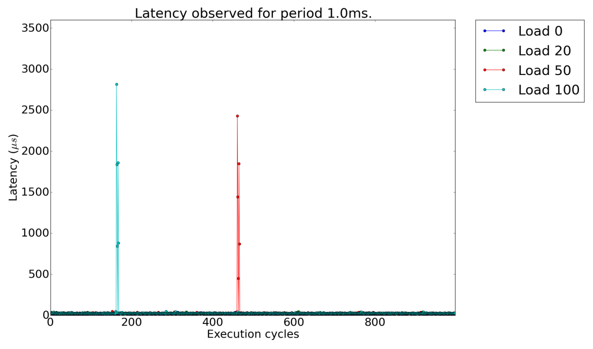


(d) Android histogram for period 10 ms

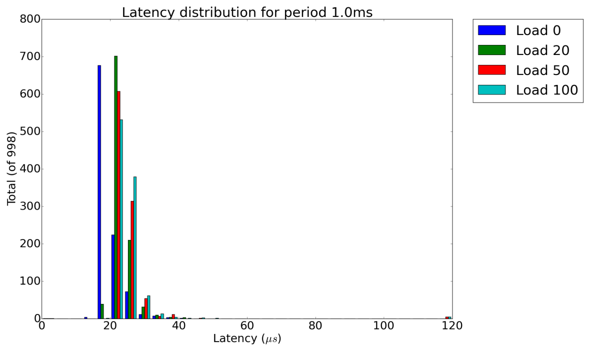
Figure 4.3: Latency measurements for a non-real-time system with high period

The figure 4.3 is also a Linux system, but with a higher period. In this case, the latency does not accumulate, but there are very significant peaks in latency of more than 150 ms in the case of the Raspberry Pi for all the loads tested. In the case of Android, the system behaves surprisingly well in the test time-frame.

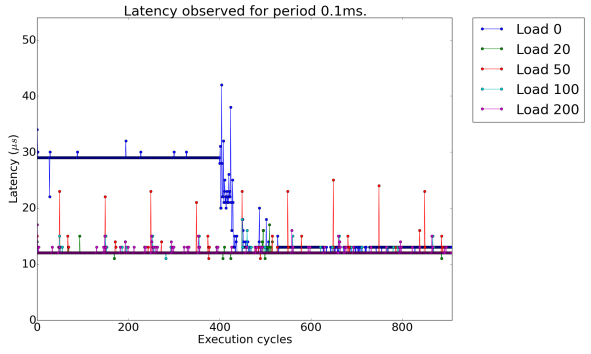
4.6.2 Preempt RT Base



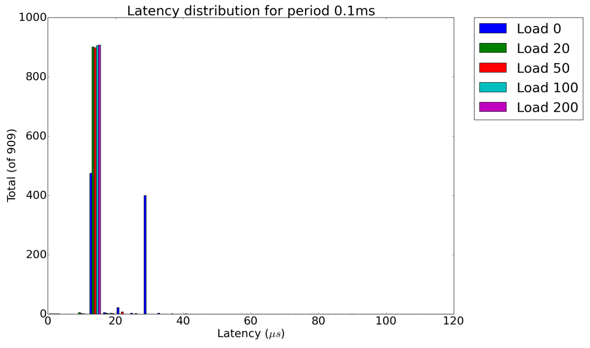
(a) Raspberry Pi line plot for period 1ms



(b) Raspberry Pi histogram for period 1ms



(c) Android line plot for period 0.1 ms



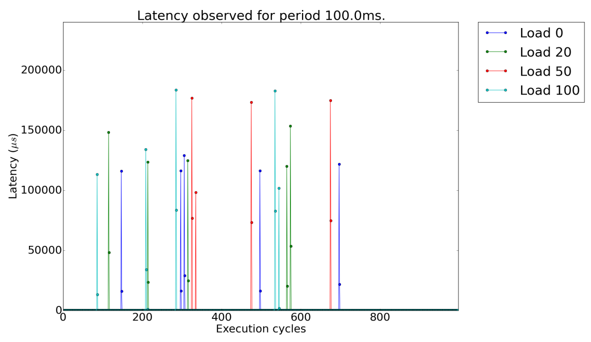
(d) Android histogram for period 0.1 ms

Figure 4.4: Latency measurements for a Preempt RT Base system with small period

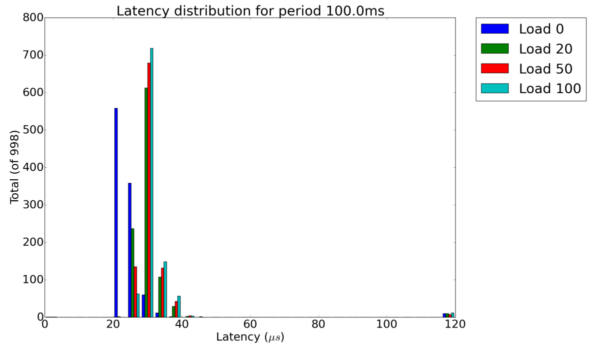
The figure 4.4 is a Linux system with Preempt RT base enabled and with very low period. In the case of the Raspberry Pi, there are a number of high peaks (of the order of 2.5 ms), but the system is stable in general. Android, on the other side, shows a very good and stable behaviour.

The figure 4.5 is a Linux system with Preempt RT base enabled and with a higher period. In the case of the Raspberry Pi, the number and size of the latency peaks is even higher than in the previous case. This may be cause of the period, because when the period is low, the test is shorter, and it is more difficult to catch work peaks in the underlying system. In the case of Android, like with lower period, the performance is very stable.

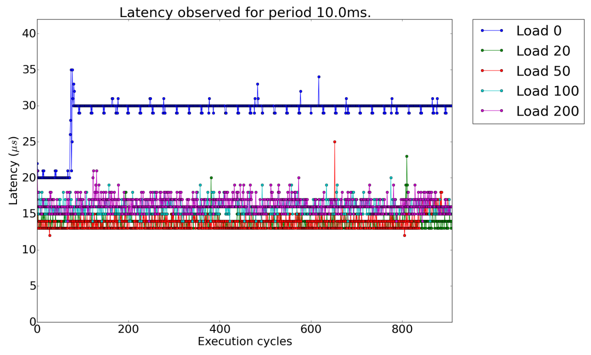
It is interesting that without load, the latency is higher than with load. This may be because the system enters a low-power mode.



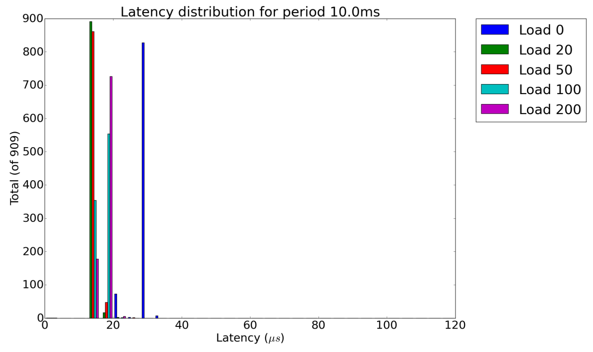
(a) Raspberry Pi line plot for period 100 ms



(b) Raspberry Pi histogram for period 100 ms



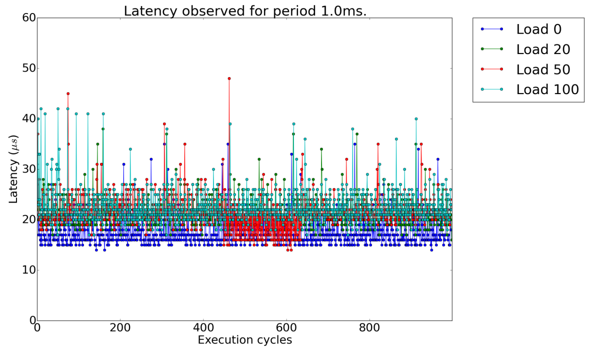
(c) Android line plot for period 10 ms



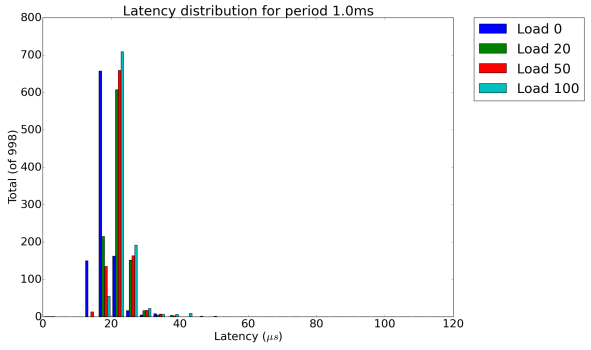
(d) Android histogram for period 10 ms

Figure 4.5: Latency measurements for a Preempt RT Base system with high period

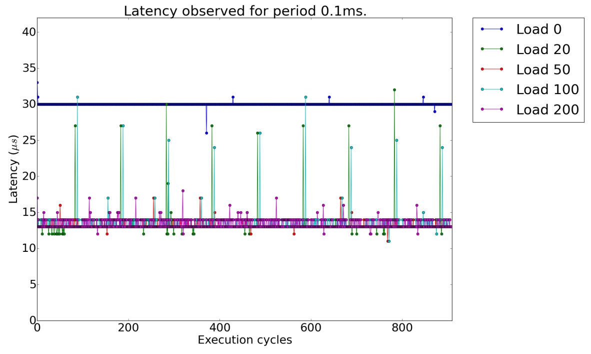
4.6.3 Preempt RT full



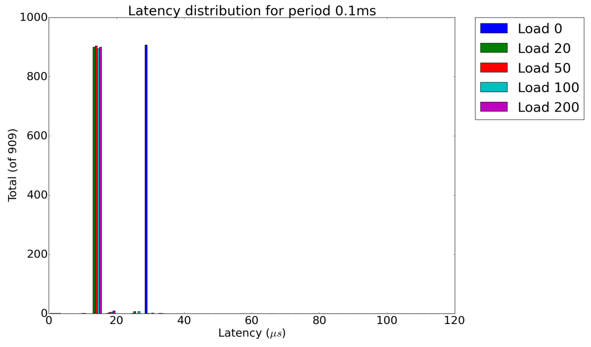
(a) Raspberry Pi line plot for period 1ms



(b) Raspberry Pi histogram for period 1ms



(c) Android line plot for period 0.1 ms



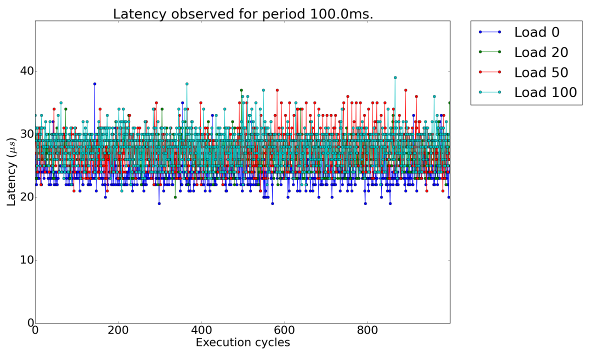
(d) Android histogram for period 0.1 ms

Figure 4.6: Latency measurements for a full Preempt RT system with small period

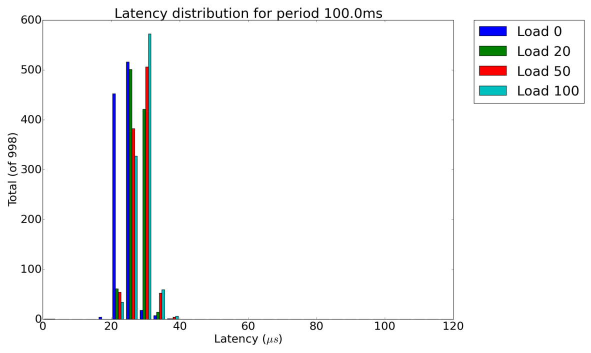
The figure 4.6 is a system with the full Preempt RT patch enabled and with low period. As expected, the latency is contained and there are no peaks in any of the systems tested.

In Android, the minimum latency is higher than in Preempt RT base. This is perfectly possible, because a real-time system is not intended to run fast, but to run predictably, and the mechanisms to do so may slow down the normal operation of the system.

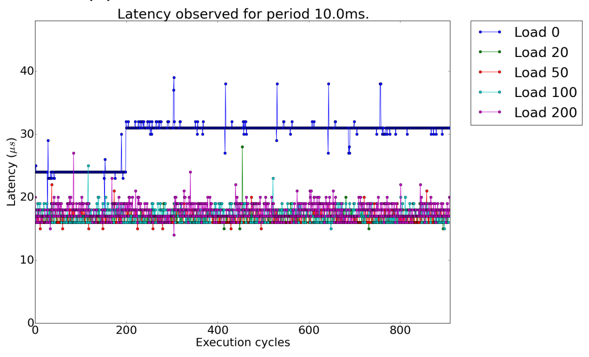
The figure 4.7 is a system with the full Preempt RT patch enabled and with a higher period. Again, the systems behave as it is expected, and the latency is bounded.



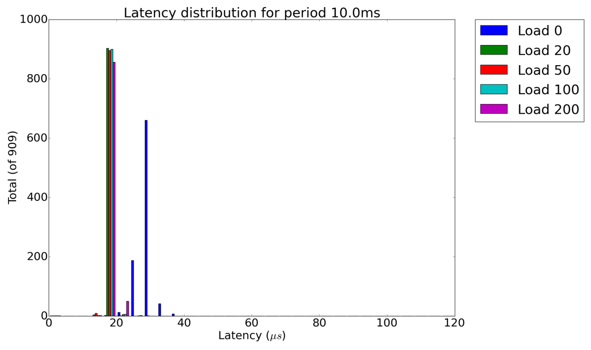
(a) Raspberry Pi line plot for period 100 ms



(b) Raspberry Pi histogram for period 100 ms



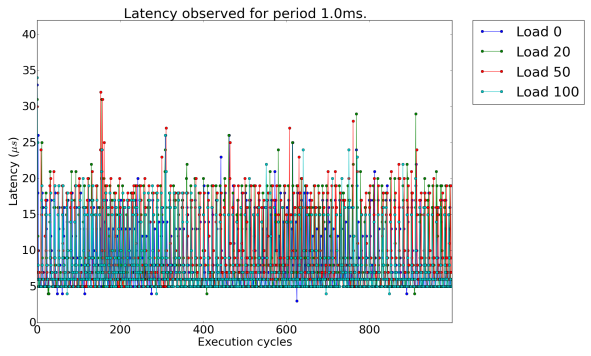
(c) Android line plot for period 10 ms



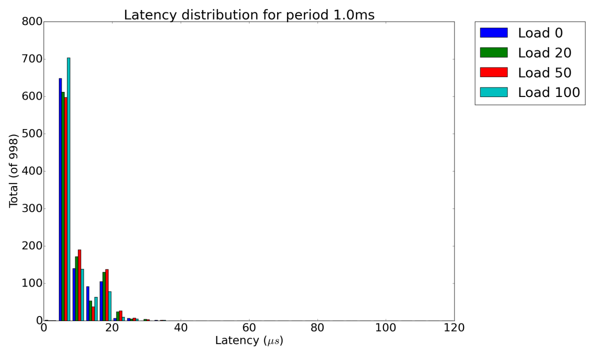
(d) Android histogram for period 10 ms

Figure 4.7: Latency measurements for a full Preempt RT system with high period

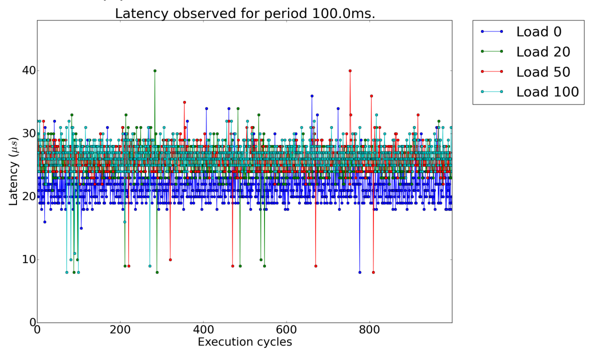
4.6.4 Xenomai



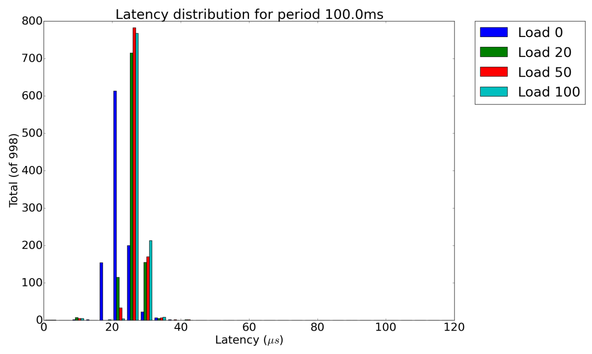
(a) Raspberry Pi line plot for period 1 ms



(b) Raspberry Pi histogram for period 1 ms



(c) Raspberry Pi line plot for period 10 ms



(d) Raspberry Pi histogram for period 10 ms

Figure 4.8: Latency measurements for a Xenomai system

Xenomai in dual-kernel configuration was also tested in the Raspberry Pi. The figure 4.8 shows the performance of this configuration. The minimum latency is of just 5 μs in the case with very low period, and of less than 20 μs in the case with higher period. These results are much lower than the results with Preempt RT. But this is expected because of the dual-kernel nature of Xenomai.

In conclusion, these results show both the necessity of using a RTOS to ensure latency bounds and scheduling determinism, and the performance of the different real-time options studied. Due to its higher computing power, Android generally shows more stability than the Raspberry Pi, even if the testing parameters are worse (lower period and higher load) than the Raspberry Pi. In the context of the ABS project, the stability obtained with Preempt RT base is enough for running satellite control software reliably, with a latency of less than 40 μs in every case. The performance of this kind of system is not the best (Xenomai shows better latency figures), but the number of changes required to obtain this solution is smaller, which means it can be adapted to other hardware platforms more easily.

5. System Data Bus

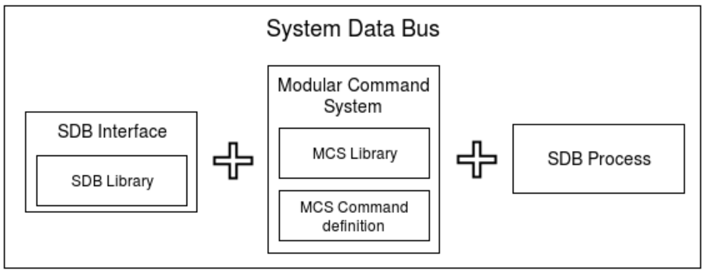


Figure 5.1: System Data Bus components

5.1 Design

The System Data Bus or SDB is the component of the software architecture responsible for distributing all the information fast, reliably and transparently through the system.

The SDB is a core component of the software architecture as it interconnects any other modules in the system. And it must be modular, so it can adapt to any mission.

For all these reasons, the SDB has been designed like a network switch, where multiple devices (in the case of the SDB, processes) can be connected an can interact between themselves. This is represented in figure 5.2.

Every process connected to the SDB has its counterpart inside, named "module" in figure 5.2. This module is the responsible for interacting with the process using the communication channel, and it also stores state information, including the group of the process. These groups indicate the privileges of a module: what commands can and cannot be sent, which commands can be received...

The SDB also contains a USB subsection, represented as "USB queue manager" in the figure. It is responsible for controlling the packets in and out from the payloads board.

To allow for maximum modularity, the Modular Command System or MCS is also designed. In figure 5.2, this is depicted as "SDB internal data". The MCS describes the packet structure and allows mechanisms to modify freely the information contained in the packets.

The specific characteristics of the SDB are the reason why kernel network management is not sufficient. The SDB is designed for per-packet filtering and routing based on groups and names, while ensuring the isolation between layers and reducing the amount of system-wide information needed. Kernel network management, on the other side, would not allow for such a fine-grained filtering, and would require the broadcasting of the addresses of each software component.

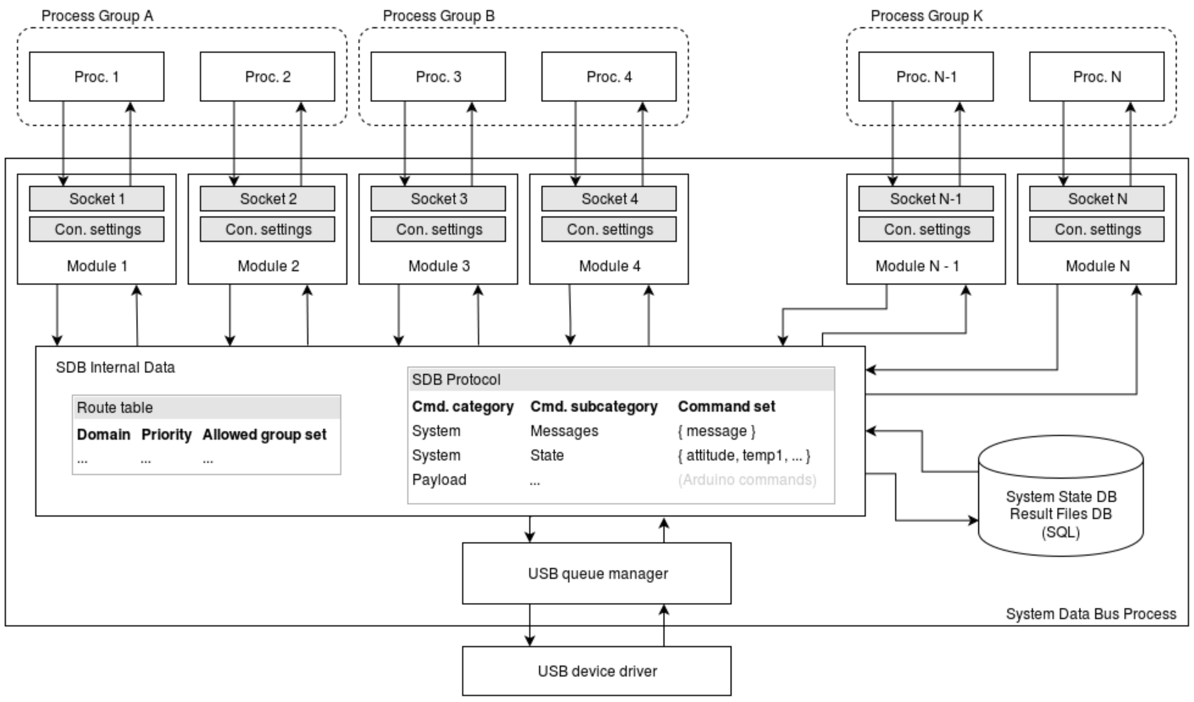


Figure 5.2: System Data Bus design

Finally, as the SDB needs to adapt for any architecture of any size and shape, it has been designed multi-threaded. Each module and section of the SDB runs on a separate thread that is usually in sleep mode to save resources.

5.2 Implementation

5.2.1 SDB components

A part from the main SDB thread, the USB section and the modules introduced in figure 5.2, two other components are introduced in implementation: the director and the observer.

The database management does not have a dedicated component. It is a set of resources and functions that can be called from any of the SDB components. More information on the databases can be found in [66].

The diagram of this implementation is shown in figure 5.3

Main thread

This thread is the responsible to create all other threads and initialize the different components. Once the system is up and running, it blocks for external signals. When a signal is received, this thread wakes all threads and requests them to exit cleanly.

The first implementation used POSIX `pthread_cancel`[1], and a clean-up function in every thread. But this function is not present in the libc implementation used in Android[2]. This final implementation uses a global "exit" variable and two mechanisms:

[1]Manual page: `http://linux.die.net/man/3/pthread_cancel`
[2]`http://stackoverflow.com/a/26824177/947194`

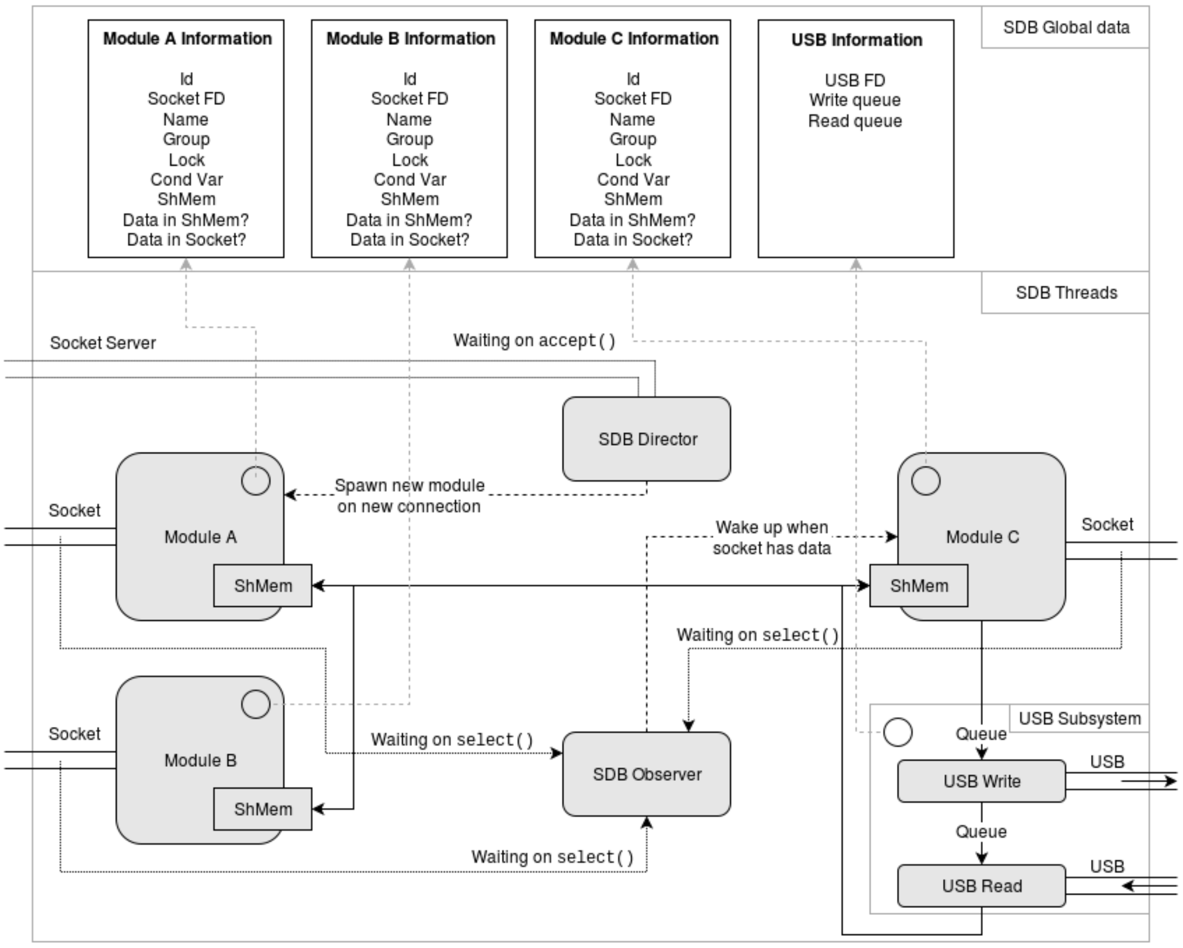


Figure 5.3: System Data Bus implementation

- If a thread would normally be blocked waiting for data from a file descriptor (socket or USB), the thread is also blocked on a pipe[3]. When the main thread requests a clean exit, data is written to the pipe to wake up the thread so it observes the global "exit" variable.
- If a thread is normally blocked waiting on a condition variable[4], the main thread signals that variable to wake up the thread. This way, the thread observes the global "exit" variable and exits cleanly.

SDB director

Given the design of the SDB, it is important to know when processes are trying to connect to the SDB. This is the reason why sockets[5] are used to communicate the SDB with the external processes.

The implementation originally used TCP sockets. The first latency tests showed high latency peaks and high variability due to TCP internal implementation. To achieve a much more consistent and reliable latency, the final implementation uses UNIX sockets.

The SDB director initializes the SDB socket as a socket server, and monitors incoming connections. The director is usually blocked in the `accept`[6] function. When there is a new connection, this thread creates a

[3]Manual page: `http://linux.die.net/man/2/pipe`
[4]Manual page: `http://linux.die.net/man/3/pthread_cond_wait`
[5]Manual page: `http://linux.die.net/man/2/socket`
[6]Manual page: `http://linux.die.net/man/2/accept`

new SDB module to handle that new process.

SDB observer

SDB modules are usually blocked waiting for incoming data either from inside the SDB (a shared memory region protected by a mutex and a condition variable) or from outside the SDB (a socket). To avoid waiting in busy loops consuming valuable resources, it is necessary to wait blocked for packets in both channels at the same time. The SDB observer is the necessary component to achieve that goal.

The SDB observer uses the `select`[7] function to block on the sockets of all active modules. When a socket contains data, the flag that indicates "data from the socket is available" is set in the corresponding module, and it is woken up by signaling its condition variable.

New modules may be created while the SDB is blocked. For this reason, the observer also blocks on a pipe. When a new module is ready, it writes to this pipe, in order to wake the SDB observer up. This is the same mechanism used by the main thread to close the system cleanly but, in this case, the "exit" global variable is not set.

Because `select` is level-triggered instead of edge-triggered, the SDB observer would not block again until the data is read from the socket. To minimize the use of resources, the observer does not block on sockets whose modules have the flag "data from the socket available" set. When modules have read the data, they wake up the observer so it can check on their socket again.

SDB modules

The SDB modules are the core of the SDB and its *raison d'être*. SDB modules are directly connected to an external process using a socket, and they are connected to every other SDB module with a shared memory region. These shared memory regions are protected by mutexes, and have condition variables. Therefore, threads can block until the state of the region changes.

A SDB module is created by the SDB director when a new connection is started. First, all data structures and variables are initialized. Then, a handshake packet is read from the socket. This handshake packet contains the name and group of the process being controlled. The name must be unique, because it is used for routing. When the module is ready, it wakes the SDB observer up, so it can acknowledge the new module. Finally, the SDB module blocks waiting for packets either from the socket, or from the memory region.

When a packet arrives to the module, these cases can happen:

- If a packet comes internally (from the memory region), it is registered in the internal module data, and sent through the socket.
- If a packet with a request comes externally (from the socket), it is registered in the internal module data and it is processed. This request may be forwarded to another module, or it may be processed immediately, depending on the request.
- If a packet with an answer comes externally (from the socket), it is sent to the module that generated the request. This information is stored in the internal data structures of the module.

Processes can request disconnection by sending the "bye" request. This will clear all data structures and close the thread of the corresponding module.

[7]Manual page: `http://linux.die.net/man/2/select`

SDB USB

The SDB USB subsection contains not one but two threads. The writing thread is blocked on a priority queue waiting for incoming data from a SDB module to the USB. When a packet arrives, it is converted to the USB packet format and sent through USB using the USB Accessory protocol [67].

The reading thread is blocked reading on the USB. When a packet arrives from the USB, it is converted to the SDB packet format and sent to the module that made the request. More information on the USB subsection and the Arduino board can be found in [66].

5.2.2 Built-in quality of service monitoring

The SDB is a complex system with many threads running in parallel. This kind of system is inherently difficult to debug and test. Moreover, it is important to obtain performance measurements from the SDB in order to check the fulfillment of its goals. Specifically, there are two metrics that have a huge impact on the rest of the system:

- **Packet error rate**. Losing or discarding packets would make the SDB unreliable. As a message distributor, the SDB must ensure high reliability in all conditions. This is already ensured in the current implementation as the SDB always returns an answer, even if it has to be an error. Packets are only discarded when they are malformed, which indicates a bug in the channel (unlikely), the interface library or the generator. Moreover, a request-answer transmission does not have a timeout, which means, an unanswered command may produce a block in some software component, but this also denotes a bug in some library or component.
- **Latency**. The SDB is the responsible to transmit information and orders through the system. It is then important that the SDB does not slow down the transmission significantly to avoid any side effect on the command execution, and, by extension, the mission itself.

For all these reasons, the SDB includes built-in QoS logging that registers every step of every packet inside the module. This logging can be enabled and disabled per module at runtime, and the data logged is dumped on request directly to a file.

A sample output from this SDB subsection for an module called "app2" can be seen in listing 5.1

```
in   |714931337 |0  |1|    |1461886049562450
ready|714931337 |0  |1|    |1461886049562541
in   |2082005750|0  |0|app2|1461886049562917
out  |2082005750|0  |0|app2|1461886049562998
in   |2082005750|253|0|    |1461886049563289
out  |2082005750|253|0|    |1461886049563297
ready|2082005750|0  |0|app2|1461886049563307
in   |1116046432|0  |4|    |1461886049602444
```

Listing 5.1: Output of QoS logging utility[8]

The columns represent:

- `in`: event. It could be one of "in" (the packet entered the module), "out" (the packet exited the module), "ready" (all operations on the packet have finished) or "scrap" (the packet was discarded).
- `714931337`: ID of the packet.
- `0`: Type of command. More information in section 5.4.
- `1`. Command. More information in section 5.4.
- *Blank*. Destination of the command, which may be implicit in the type of command.
- `1461886049562450`. Absolute time in microseconds from UNIX epoch [68].

[8]This output is padded with extra spaces for readability

The QoS log serves the purpose mentioned:

- **Debugging**: Being able to trace where each packet is in every moment inside the SDB, and being able to trace when a bogus packet is detected helps the process of debugging of both the SDB and the components connected to the SDB.
- **Performance measures**: The trace output contains the time of each event of a packet inside the SDB. This information allows to obtain latency measurements and check the SDB performance. Sections 5.5 and 5.6 are devoted to this purpose.

5.3 SDB Libraries

In order to simplify operations with the SDB, a library has been developed. This library contains wrappers to common functionalities and adds some extra functionalites for the components of the software architecture that will interact with the SDB.

An important addition of these libraries are callbacks. Callbacks are functions that are run automatically when a given command arrives.

5.3.1 Methods

The methods in this library are detailed in table 5.1. Functions with integer return value return 0 when the function succeeds and a negative error code when it fails.

`int sdb_connect( const char *name, SDBGroup group)`	Connect to the SDB with the given name and group.
`int sdb_disconnect(void)`	Send the "bye" command to the SDB and close the open socket.
`int sdb_register_callback( MCSCommand cmd, void (*callback) (MCSPacket *pkt_in, MCSPacket **pkt_out))`	Register a callback for the given command. The callback gets as input the packet that triggered the callback, and outputs the answer to that input.
`int sdb_unregister_callback( MCSCommand cmd)`	Unregister the callback for the given command.
`int sdb_send_sync( MCSPacket *pkt_in, MCSPacket **pkt_out)` `int sdb_send_sync_and_free( MCSPacket *pkt_in, MCSPacket **pkt_out)`	Send a packet and block until the answer is received. The `_and_free` version frees the input packet after being sent.

`int sdb_send_async( MCSPacket *pkt_in, SDBPendingPacket **pkt_out)`
`int sdb_send_async_and_free( MCSPacket *pkt_in, SDBPendingPacket **pkt_out)`
	Send a packet and return a handler to check the state of the transmission. See listing 5.2 for implementation detail. The `_and_free` version frees the input packet after being sent.

`int sdb_wait_async( SDBPendingPacket *pkt)`	Block until the asynchronous transmission is completed. Out packet can be read from `pkt->pp_pkt`.
`int sdb_check_async( SDBPendingPacket *pkt)`	Check the state of the asynchronous transmission. It returns 1 if transmission is completed, 0 if transmission is not completed, or a negative error code in case of error.
`void sdb_pending_packet_free( SDBPendingPacket *pkt)`	Free an asynchronous transmission handler.

Table 5.1: Public interface for the SDB library

```
typedef struct SDBPendingPacket {
    unsigned int pp_id;
    MCSPacket *pp_pkt;
    bool pp_valid;
    pthread_mutex_t pp_lock;
    pthread_cond_t pp_cond;
} SDBPendingPacket;
```

Listing 5.2: SDBPendingPacket format

5.3.2 Usage

Any process that wants to communicate with the SDB should use these libraries. The usual steps are:

1. Register callbacks using `sdb_register_callback` for those commands that can be received at any moment.
2. Connect to the SDB using `sdb_connect`.
3. Do all necessary operations and communications using `sdb_send_sync` and `sdb_send_async`.
4. If the process finishes, disconnect from the SDB using `sdb_disconect` before exiting.

An example HWmod using these libraries can be found in appendix A

5.3.3 Implementation

Data structures

The SDB library is implemented based on two lists:

- **SDB callback list**: this list contains the pairs of command - callback that are registered.
- **SDB pending packet list**: this list contains all the pending packets. This is, the packets that have been sent but have not received an answer yet.

These lists are initialized using a constructor function (called when the library is loaded), and they are freed using a destructor function (called when the library is unloaded).

A pending packet is defined as in listing 5.2, where each field means:

- `pp_id`: Id of the packet.
- `pp_pkt`: Memory region for the answer packet.
- `pp_valid`: True if `pp_pkt` is valid.
- `pp_lock`: Lock for this pending packet.
- `pp_cond`: Condition variable for this pending packet. It is signaled when `pp_valid` is set to true. It is used by `sdb_async_wait` and `sdb_send_sync`.

Connection and disconnection

The function `sdb_connect` opens a socket connection to the SDB and sends the handshake packet. It also starts a thread responsible for watching the socket for incoming data. When this thread receives a packet, three cases can happen:

- If a packet is an answer from a previously sent command, the request packet is retrieved from the internal data, and marked as ready. Any threads locked for this packet are woken up.
- If a packet contains a command that has a callback registered, a new thread is created to run the callback. The return packet of the callback will be sent as an answer.
- Otherwise, the packet is answered with an error code.

This reading thread can also detect when the socket has been closed unexpectedly because, in this case, a read on the socket returns no data. When this happens, it will try to reconnect to the SDB automatically.

The function `sdb_disconnect` sends a "bye" packet to the SDB, and requests the reading thread to exit. This operation is done in the same way the main thread in the SDB notifies the other threads to exit: with a pipe and a global variable.

It is important to note that a call to `sdb_disconnect` does not unregister callbacks. If a new connection to the SDB is started afterwards, it will still have the same callbacks.

Sending packets

Packets can be sent to the SDB using the functions `sdb_send_sync` and `sdb_send_async`. Both functions add a new element to the pending packet list and send the given packet.

The `sync` function blocks on the pending packet condition variable until the answer packet is received.

The `async` function returns the pending packet handler. This handler can be managed with the functions `sdb_check_async`, which returns the current state, or `sdb_wait_async`, that blocks at the pending packet handler until the answer packet is received.

5.4 Modular Command System

5.4.1 Introduction

The keystone for achieving the goals of the SDB, high modularity while ensuring robustness, is the Modular Command System or MCS. The MCS allows the user to define and customize what commands and packets are valid inside the SDB.

The MCS has two main components: the configuration file and the helper library.

5.4.2 Configuration file

The MCS configuration file is a simple JSON [69] formatted file. JSON is used because it allows for simple reading and modification both manually and automatically. A full example of this configuration file can be found in appendix B.

The MCS configuration file has the format from listing 5.3.

```
{
"command_list" : [
    command1,
    command2,
    ...
    commandN
]
}
```

Listing 5.3: MCS Configuration File

With each command defined as in listing 5.4

```
{
    "name" : "string",
    "description" : "string",
    "nargs" : integer,
    "raw_data" : boolean,
    "type" : "string",
    "config" : config
}
```

Listing 5.4: MCS Command definition

The fields that define a MCS command are:

- `name`: Command identifier string. Must be composed just by lowercase letters, numbers (not the first character) and underscores. It must be unique per each command.
- `description`: Human readable description, used for creating documentation. Might be `null`.
- `nargs`: Fixed number of arguments that this command accepts. Might be zero.
- `raw_data`: This field will be `true` when the command also has arbitrarily long data attached. In case there is no raw data, it should be `false`.
- `type`: Type of the command. It is one of `state`, `message` or `payload`.
- `config`: configuration fields dependent on the type of command.

Messages

Commands of type `message` are simple messages sent from one module of the SDB to another, and each one of them should be able to understand it. In this case, the field `config` is defined as in listing 5.5.

```
{
    "destination" : "string",
    "origin_groups" : ["string", "string", ...],
    "destination_groups" : ["string", "string", ...],
    "response_size" : integer
}
```

Listing 5.5: MCS messages specific configuration

The configuration fields in listing 5.5 are:

- `destination`: The destination for this message. It can be static (the value is the name of the destination), an argument (the value is `@arg`), or it can be `null` (the message is for the SDB).
- `origin_groups`: The array of groups that can send this command. Groups not mentioned cannot send the command. The wildcard `any` can be used, and all other elements in the array can be ignored.
- `destination_groups`: The array of groups that can receive this command. If the destination is static or `null`, it is ignored. Groups not mentioned cannot receive the command. The wildcard `any` can be used, and all other elements in the array can be ignored.
- `response_size`: number of bytes expected in the response. It can be 0 if there is no expected data, or -1 if the size is unknown beforehand.

States

If the command is of type `state`, it relates with a system state, such as the value of a temperature sensor. The values requested using these commands are stored in an intermediate database. The field `config` for states is defined in listing 5.6.

```
{
    "destination" : "string",
    "dimensions" : integer,
    "return_type" : "string",
    "unit" : "string",
    "dimension_name" : "string"
}
```

Listing 5.6: MCS states specific configuration

The configuration fields in listing 5.6 are:

- `destination`: Destination of this message.
- `dimensions`: Fixed number of dimensions that this state has. Must be one or bigger.
- `return_type`: Type of each of the dimensions of the state. It is one of `int`, `float` or `string`.
- `unit`: Human readable unit of the state. Might be `null`.
- `dimension_name`: Human readable name of the dimensions in the order returned by the update function. Might be `null`.

Payloads

Commands of type `payload` are messages that will be sent through the USB subsystem to the payload controller (Arduino board). In this case, the field `config` is defined as in listing 5.7.

```
{
    "command" : integer,
    "parameters" : integer,
    "arguments" : [integer, "string", ...],
    "data" : "string",
    "response_size" : integer
}
```

Listing 5.7: MCS payload specific configuration

The configuration fields in listing 5.7 are tightly related to the USB packet specification detailed in [66]:

- `command`: The `cmd` field for the command.

- `parameters` : The `parameters` field for the command.
- `arguments`: Array with the same number of elements as in `nargs`. Might be the value of an argument in the command, using `@argX`, where X is the number of argument, starting by 0.
- `data`: The `data` field in the command. It might be `null`, `@arg` to indicate the same data as in the packet, or hexadecimal numbers (to send binary data).
- `response_size`: Number of bytes expected back as the command response. It can be 0 if there is no expected response, or -1 if the size is unknown beforehand.

Translation

This configuration file is processed automatically when the ABS software architecture is compiled. This process generates the C header file that guides the decisions of the SDB. Detail on the translation output can be found in figure 5.4, and an example of a result file in appendix B.

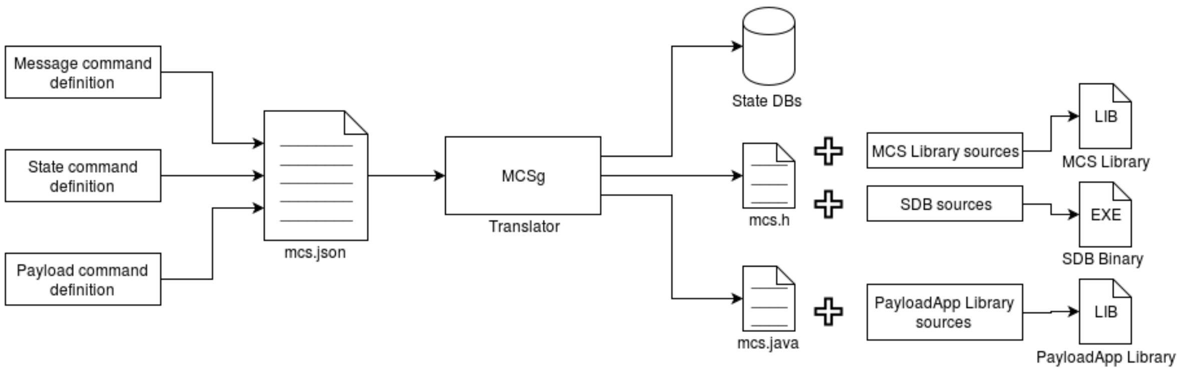


Figure 5.4: MCS translation process

5.4.3 MCS Library

The other important part of the MCS is the library. This library defines MCS packets and helper functions.

The MCS packet format is shown in listing 5.8

```
typedef struct MCSPacket {
    unsigned int id;
    enum MCSType type;
    unsigned short cmd;
    unsigned short nargs;
    unsigned char *args;
    char *dest;
    unsigned short data_size;
    unsigned char *data;
} MCSPacket;
```

Listing 5.8: Definition of a MCS packet

The elements for these packets are:

- `id`: unique ID.
- `type`: Type of command.
- `cmd`: Command.
- `nargs`: Number of arguments.

- `args`: List of arguments, with size defined by `nargs`.
- `dest`: Destination of the packet, if not specified implicitly by the command or the command type.
- `data_size`: Size of raw data.
- `data`: Raw data, with size defined by `data_size`.

The valid types for MCS packets are messages, states, payload, OK, OK with data or error. A MCS packet of type message, state or payload is answered with a packet with type OK (command succeeded), OK with data (command succeeded and generated output data) or error (command failed). These requests and answers are linked in the system because both share the same packet ID.

To simplify operations with MCS packets, the set of functions in table 5.2 is defined.

`MCSPacket *mcs_read_command( int rfd, int wfd)`	Read a MCS packet from a file descriptor.
`int mcs_write_command( MCSPacket *pkt, int fd)` `int mcs_write_command_and_free( MCSPacket *pkt, int fd)`	Write a MCS packet to a file descriptor. The `_and_free` version frees the input packet after being sent.
`MCSPacket *mcs_ok_packet( const MCSPacket *from, void *data, size_t size)` `MCSPacket *mcs_ok_packet_id( unsigned int id, void *data, size_t size)` `MCSPacket *mcs_ok_packet_data( const MCSPacket *from, void *data, size_t size)` `MCSPacket *mcs_ok_packet_data_id( unsigned int id, void *data, size_t size)` `MCSPacket *mcs_err_packet( const MCSPacket *from, int err_code)` `MCSPacket *mcs_err_packet_id( unsigned int id, int err_code)`	Create a packet of type OK, OK with data, or error, given the request packet. In the `_id` version, the packet id is used directly, instead of the request packet.

`MCSPacket *mcs_create_packet( MCSCommand cmd, unsigned short nargs, unsigned char *args, unsigned short data_size, unsigned char *data)` `MCSPacket *mcs_create_packet _with_dest( MCSCommand cmd, char *dest, unsigned short nargs, unsigned char *args, unsigned short data_size, unsigned char *data)`	Create a new packet given all necessary fields. If the `_with_dest` version is used, the packet destination can be specified (only in the case the command allows it).
`int mcs_err_code_from_command( const MCSPacket *pkt)`	Extract the error code from an error packet.
`const char *mcs_command_to_string( const MCSPacket *pkt)`	Convert an MCS command to a string, for debug purposes.
`bool mcs_is_answer_packet( const MCSPacket *pkt)`	Returns true if the packet is of type OK, OK with data, or error.
`MCSCommand mcs_command( MCSPacket *pkt)`	Extract the MCS command from the packet.
`void mcs_free(MCSPacket *pkt)`	Free a MCS packet structure.

Table 5.2: Public methods in the MCS library

5.5 Testing

As stated in subsection 5.2.2, debugging the SDB is challenging. To simplify this operation, a number of unit tests has been developed. These tests can run after compilation to verify that changes in code did not introduce errors in the system. Current tests include:

- Check that all types of MCS commands can be generated, and send and received through a channel, using the MCS library functions.
- Check that the SDB can be started, and messages can be sent from and to two different modules.
- Check that a connection to the SDB can be established and messages can be sent and received using the SDB library functions.

The other tests that can be run on the SDB are stress tests, which can also be used to analyze the performance of the SDB in harsh conditions. As introduced in subsection 5.2.2, the metric that defines the SDB performance is *packet latency*. This latency can be studied in the SDB using the output from the QoS subsystem. Considering all the parameters that can be changed in a packet and all the ways a packet can be sent, five variables are considered when studying latency in the SDB:

- **Packet type**. Depending on the packet type, the processing is different. The tests that were run are focused only on packets of type message.

- **Packet rate** or time between packets. This time is between a full connection (request sent and answer received) and the next.
- **Size of the data** in the packet.
- **Number of modules** connected to the SDB.
- **Relation between modules**. The tests that were run use the two extreme cases: relation all to one (a2o), where all modules send packets to the same module, and all to all (a2a), where the module N just send to the module N + 1 (and therefore, it receives from module N - 1).

In order to sweep these variables automatically, a test suite has been developed. This test suite contains a fully configurable test module and a test controller.

The test controller has the following inputs:

- Number of packets sent per test case per module.
- Packet rate interval (minimum, maximum and step).
- Data size interval (minimum, maximum and step).
- Number of sending modules interval (minimum, maximum and step).
- Relation between modules.

When the test controller is run, all the parameters are read and checked, an output file is created and the SDB process is started. Next, the intervals for packet rate, data size and number of modules are swept. For every possible combination of these variables (a test case), the required number of modules are spawned. These modules are created with the test case configuration, and they are synchronized to start in the same moment. The test controller then waits until the sender modules finish, and sends a `SIGINT` signal to the receiver module (if it is present). Once all test cases are run, a `SIGINT` signal is sent to the SDB to request a clean exit, and the test controller exits.

The test module can be configured with the following input parameters:

- Name of the module.
- Type of module, which can be receiver (will only wait for incoming packets, and all the other parameters are ignored), or sender.
- Destination of the packets.
- UNIX time to start sending packet. It is used to synchronize modules.
- Packet rate.
- Data size.
- Number of packets to send.

When the test module is run, all the parameters are read and checked, and an output and log files are created. Then, callbacks for the two types of message used (test message and test message with data) are registered, and the test module connects to the SDB. The next step is to request QoS logging for this module, and sleep until the test start time. When the test module wakes up, it starts to send packets with the conditions specified in the input parameters. In the case it does not have to send packets, it waits for the `SIGINT` signal. Finally, a request to dump the QoS logging to the output file is sent, and the test module disconnects from the SDB and exits.

This test suite allows to test the SDB in a wide variety of conditions and scenarios.

5.6 Results

In the current SDB version, all the tests run without any error, which indicates the SDB in its current state is perfectly usable.

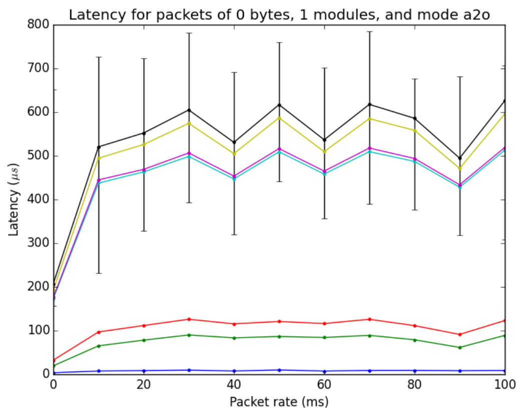
As introduced in subsection 5.2.2, these tests can also be used to study the performance of the SDB, in specific, the latency of packets. Using the test suite described in section 5.5, and a Python script to process automatically the test output, a few latency plots for different cases were generated.

The test results are represented using line plots, with several lines and several dots. Each dot represents 100 packets per module transmitted in the same test case, with the maximum and minimum values indicated with error bars. Modules are synchronized at the beginning of each test case. Each of the lines represent an event inside the SDB:

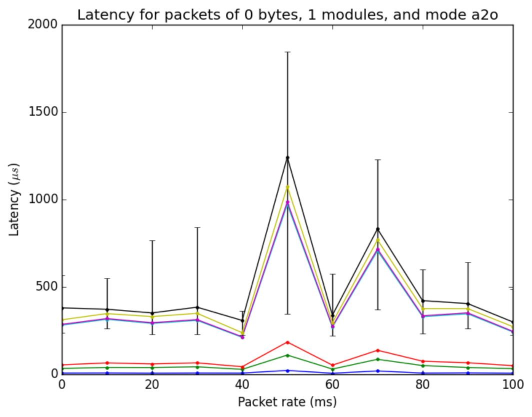
- Time 0: time when the packet entered the sender SDB module.
- Blue (—●—): time when the packet left the sender SDB module.
- Green (—●—): time when the packet entered the receiver SDB module.
- Red (—●—): time when the packet left the receiver SDB module (through the socket).
- Cyan (—●—): time when the packet entered the receiver SDB module (through the socket). The difference between the red line and the cyan line is also the latency of two socket transmissions plus the processing time in the receiving process.
- Magenta (—●—): time when the packet left the receiver SDB module.
- Yellow (—●—): time when the packet entered the sender SDB module.
- Black (—●—): time when the packet left the sender SDB module. This is also the total latency for the packet.

All the test results are run on both a reference platform and the target platform. The reference platform is a laptop with an Intel i5 processor with 2 cores and 4 execution threads, and 4 GB of RAM memory. The target platform is an LG Nexus 5 [19].

5.6.1 Packet rate



(a) PC



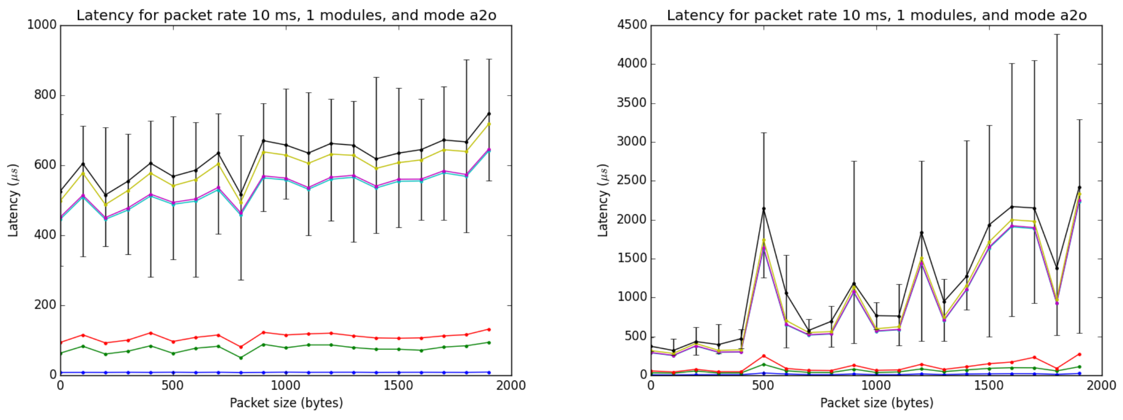
(b) Android

Figure 5.5: SDB latency measurements with variable packet rate

The figure 5.5 is the plot corresponding to a fixed number of modules (1 sending and 1 receiving) with mode all to one and a fixed data size (0 bytes) for a variable packet rate (from 0 to 100 ms in steps of 10 ms).

This plot shows an average latency for PC of less than 650 μs for any packet rate, except for the case with 0 ms between packets, where the latency drops. This may be caused by scheduling policies. In Android, the average latency is less than 500 μs, except for the cases with 50 ms and 70 ms between packets. This may be caused by background processes or some power-saving mechanism.

5.6.2 Data size



(a) PC (b) Android

Figure 5.6: SDB latency measurements with variable data size

The figure 5.6 is the plot corresponding to a fixed number of modules (1 sending and 1 receiving) with mode all to one and a fixed packet rate (10 ms) for a variable data size (from 0 to 1900 bytes in steps of 100 bytes).

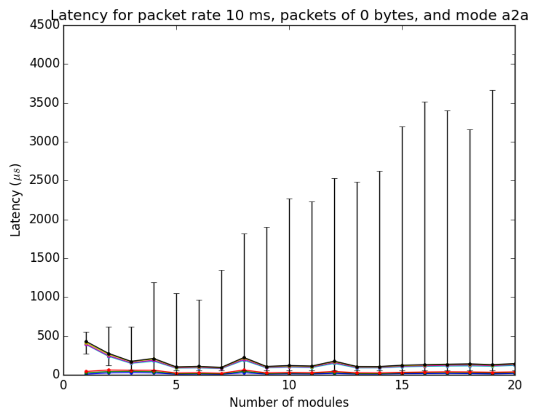
In the PC case, there is a very slight increase in latency with larger data sizes, whereas in Android it is much more intense. Again, Android shows some variability and peaks due to background processes or low power mechanisms.

5.6.3 Modules with relation all to all

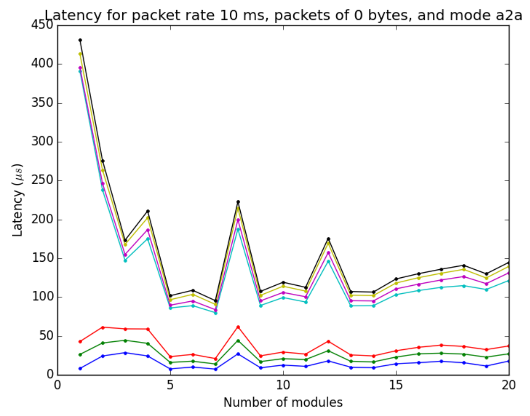
Figures 5.7 and 5.8 are the plots corresponding to variable number of modules (from 1 to 20 in increments of 1) with mode all to all. The data size is fixed to 0 bytes. Note that the case of one module in mode all to all corresponds to one module talking to itself. In this case, the versions with and without error bars are shown for better clarity because the peaks are higher.

In figure 5.7, packet rate is 10 ms. This packet rate allows the system to process all packets comfortably, resulting in a more or less constant average latency. As the system load is higher, the maximum latency values are also higher than previous test cases. In the PC case, the higher load allows the system to devote more resources in running the SDB and the processes connected, giving a very low latency of less than 150 μs in general. On the other side, the Android system with higher load shows higher latency, of around 1500 μs in average. The case of one module is special, because it represents one module talking to itself. This case has a higher latency in PC than in Android.

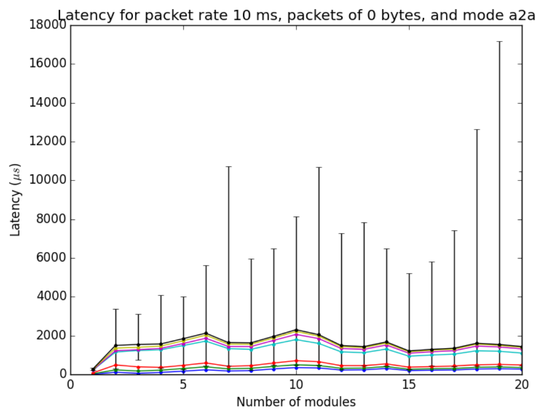
In figure 5.8, packet rate is 0. This means, all modules are trying to send as soon as they receive the previous answer. In this case, both the PC and Android show an increase in latency. As in the other test cases, Android shows peaks in the latency values.



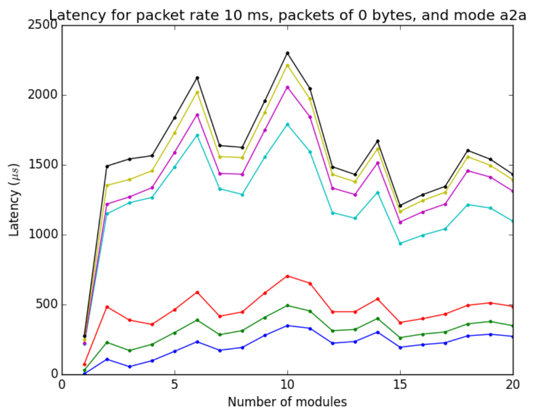
(a) PC with error bars



(b) PC without error bars

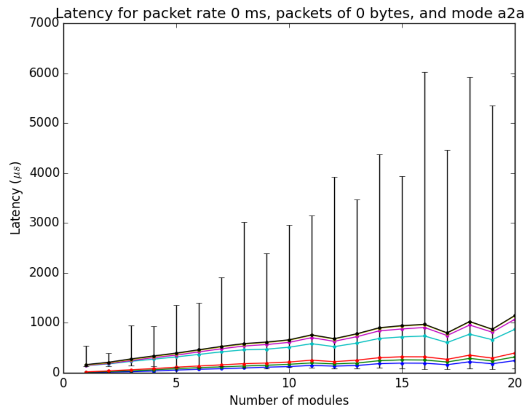


(c) Android with error bars

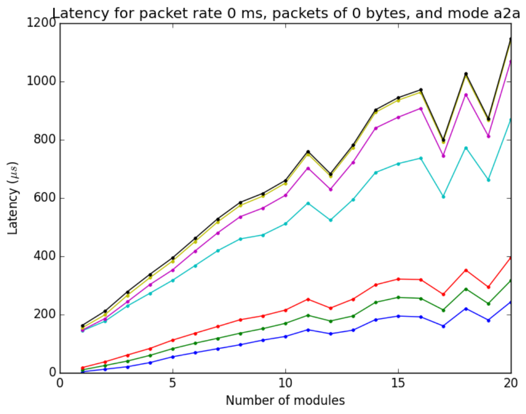


(d) Android without error bars

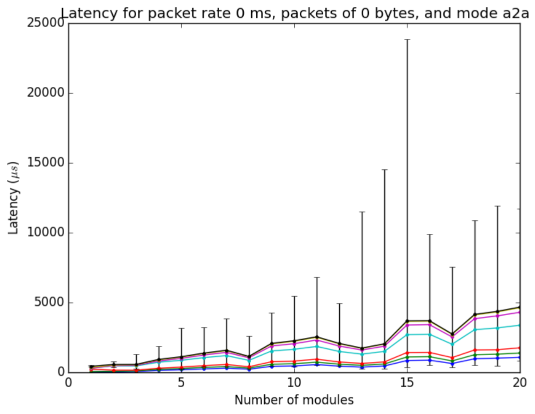
Figure 5.7: SDB latency measurements with variable number of modules, relation all to all and packet rate 10 ms



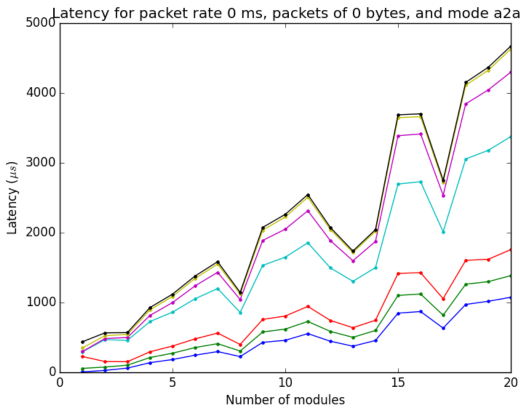
(a) PC with error bars



(b) PC without error bars



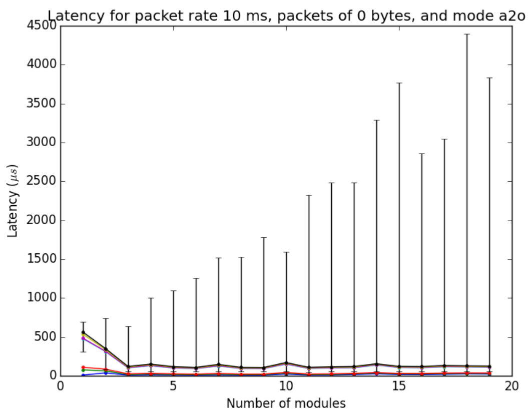
(c) Android with error bars



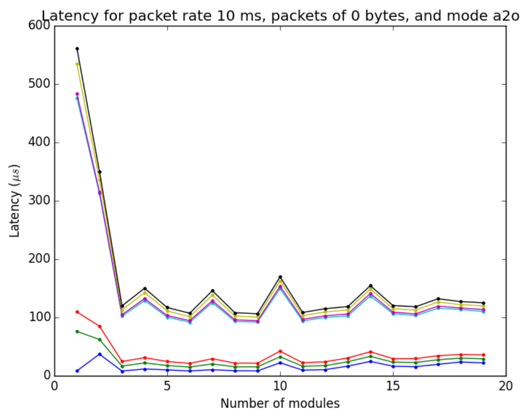
(d) Android without error bars

Figure 5.8: SDB latency measurements with variable number of modules, relation all to all and packet rate 0 ms

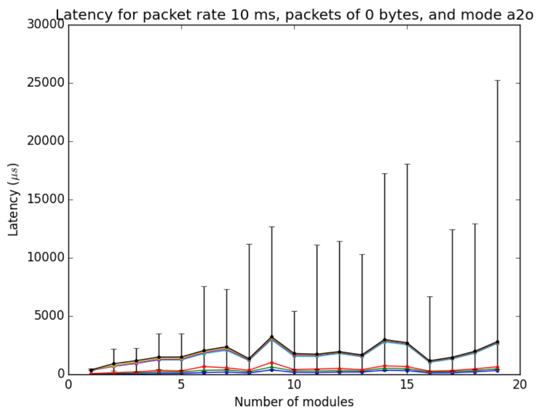
5.6.4 Modules with relation all to one



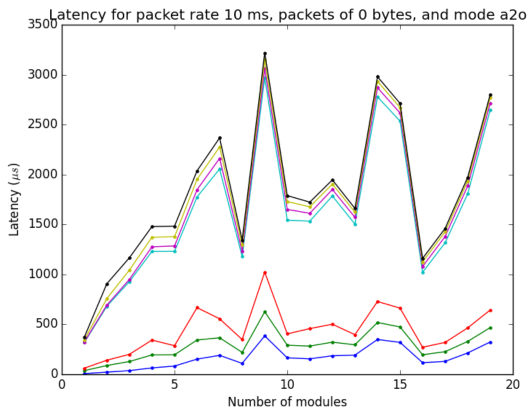
(a) PC with error bars



(b) PC without error bars



(c) Android with error bars



(d) Android without error bars

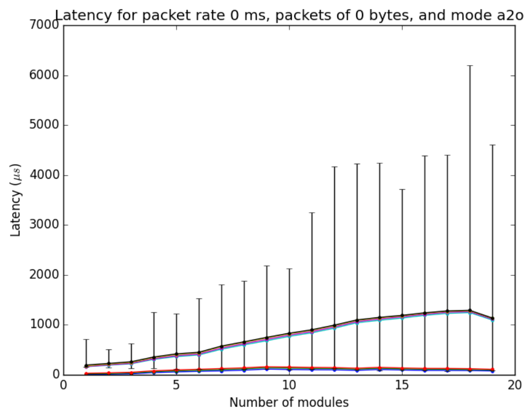
Figure 5.9: SDB latency measurements with variable number of modules, relation all to one and packet rate 10 ms

Figures 5.9 and 5.10 are the plots corresponding to a variable number of modules (from 1 to 19 in intervals of 1, plus a receiving module) in mode all to one. The data size is fixed to 0 bytes. Again, the versions with and without error bars are shown.

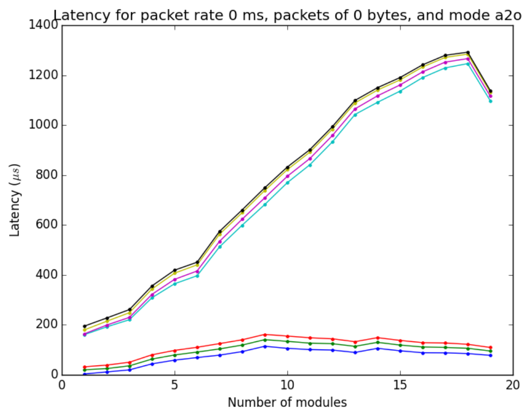
In figure 5.9, packet rate is 10 ms. This case is very similar to figure 5.7, but the Android system copes with the load much worse. In this case, it seems that one and two modules in PC are less efficient than in the other cases.

Finally, figure 5.10 shows a very extreme case: all modules sending packets to one module as fast as they can. This test case shows a steady latency slope, as it would be expected.

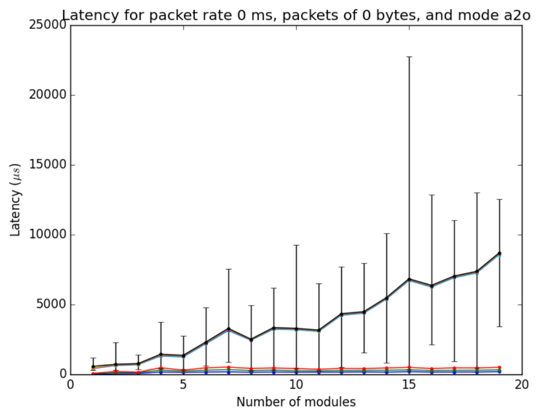
In conclusion, the SDB shows very good figures of latency, of very few milliseconds for a PC in the worst-case scenario, and less than 10 ms for an Android in the same scenario. It is important to mention that most of the contribution and variability on the latency is not in the SDB itself, but the processing of the packets outside of it: sending through the socket, generating an answer, and sending the answer back to the SDB. In general, Android shows higher latency and higher variability than the PC, both for the difference in power and the difference in the operating system.



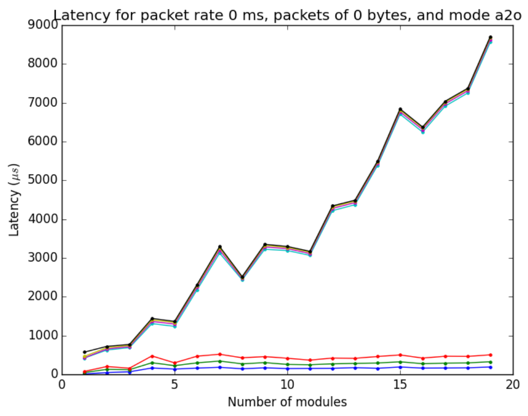
(a) PC with error bars



(b) PC without error bars



(c) Android with error bars



(d) Android without error bars

Figure 5.10: SDB latency measurements with variable number of modules, relation all to one and packet rate 0 ms

6. Conclusion

This project has been oriented to the design and implementation of an Android real-time kernel and a message distributor as part of a software platform for open nano-satellite development in the context of the Android Beyond the Stratosphere project. The software developed is published online as open-source code[1] that can be reviewed, used and modified by anyone interested.

This project has been developed in the margins established in the project management section (chapter 3).

6.1 Android Real Time Operating System

The Linux kernel underneath Android has been modified to enable real-time capabilities, and it is now possible to run native applications with contained latency. Although the resulting system does not have all real-time features enabled, its latency margins are good enough for the ABS project.

These results are an important step not only for the ABS project, but for many other projects that would benefit from having a Real Time Operating System on a user-friendly platform such as Android. Examples could be medicine, where biological signals must be measured with strict latency, or industry, where new control systems could be implemented on Android devices, and many more.

6.2 System Data Bus

On the other side, the core message distributor for the ABS software architecture, the SDB, has been developed following nano-satellite software design standards. The result is a very modular and flexible platform, while still ensuring robustness in its operation. All the tests run on this software component show a very predictable performance with a very small latency, which indicate that this message distributor does not affect in any significant way the performance of the components that use it.

The SDB, and by extension, the whole ABS software architecture can be modified for any nano-satellite mission and can be deployed in a very short period of time. The only steps required are:

1. Download the software architecture from the official sources.
2. Decide on the specific MCS commands required for this nano-satellite mission and introduce them in the MCS configuration.
3. Develop the necessary Hardware Modules and App Modules to run all the experiments specific to the mission.
4. Compile and write to the target device, the Android phone.

[1]The ABS software architecture can be found at `https://github.com/abs-platform/abs-software`. The ARTOS can be found at `https://github.com/abs-platform/artos-d2`, with documentation about the process at `http://abs-platform.github.io/artos-d2/`

6.3 Future work

The work described in this thesis fulfills the objectives proposed. but there is still a lot of work pending to improve the results even further.

The current ARTOS implementation uses Preempt RT in its base mode, because the system with Preempt RT full is still not able to load the Android user space. To achieve a proper ARTOS that can be used in any kind of application (industrial, medical, aerospace...), it is critical to solve the issues with the current full version.

The SDB is also functional in its current state, but there are still improvements that can be added, such as merging the SDB observer and SDB director, which would simplify the SDB implementation. Another possible improvement is to add functionalities specifically designed for HWmods in the libraries, such as registering a callback that halts the module. Some functionalities are also still missing in the SDB, such as the libraries to interact with Java applications, or databases. Moreover, the Distributed System Layer, which enables inter-satellite communication inside a constellation of nano-satellites, has not been designed yet.

All these issues must be addressed in the future to obtain a fully functional platform that can be used in any nano-satellite or nano-satellite constellation project.

Bibliography

[1] Multiple authors. CubeSat Design Specification. Revision 13. 2015. Available online: http://3www.cubesat.org/images/developers/cds_rev13_final2.pdf. Accessed: 2016-02-28.

[2] Roger Jové Casulleras. *Contribution to the development of pico-satellites for Earth observation and technology demonstrators*. PhD thesis, Universitat Politècnica de Catalunya, 2015. Available online: http://hdl.handle.net/10803/286237. Accessed: 2016-02-28.

[3] AUUSAT 4 project homepage. http://www.space.aau.dk/aausat4/. Accessed: 2016-02-24.

[4] Swiss Cube, the first swiss satellite. http://swisscube.epfl.ch/. Accessed: 2016-02-24.

[5] Planet Labs approach. https://www.planet.com/approach/. Accessed: 2016-02-24.

[6] Tyvak Nano-Satellite Systems INC homepage. http://tyvak.com/. Accessed: 2016-02-24.

[7] ISIS Innovative solutions in space homepage. http://www.isispace.nl. Accessed: 2016-02-24.

[8] ESA. GOCE ESA mission. http://www.esa.int/Our_Activities/Observing_the_Earth/GOCE/Facts_and_figures. Accessed: 2016-02-28.

[9] ESA. TDRS-K NASA mission. http://www.space.com/19559-nasa-launches-relay-satellite-tdrs-k.html. Accessed: 2016-06-10.

[10] M. Unwin, M. Meerman, and M. Sweeting, Sir. A NanoSatellite to Demonstrate GPS Oceanography Reflectometry. In *16th AIAA / USU Conference on Small Satellites*, 2002.

[11] Carles Araguz López. Towards a modular nano-satellite software platform: Prolog constraint-based scheduling and system architecture. Bachelor thesis, Universitat Politècnica de Catalunya, 2014. Available online: http://hdl.handle.net/2099.1/22545. Accessed: 2016-02-24.

[12] C. Araguz et. al. On autonomous software architectures for distributed spacecraft: A Local-Global policy. In *Aerospace Conference, 2015 IEEE*, 2015.

[13] Alessandro Golkar and Ignasi Lluch i Cruz. The Federated Satellite Systems paradigm: Concept and business case evaluation. *Acta Astronautica*, 111:230 – 248, 2015.

[14] O3b Networks. http://www.o3bnetworks.com/technology/. Accessed: 2016-02-25.

[15] OneWeb. http://oneweb.world. Accessed: 2016-02-25.

[16] Christopher Boshuizen, James Mason, Pete Klupar, and Shannon Spanhake. Results from the Planet Labs Flock constellation. 2014.

[17] S Kenyon, CP Bridges, D Liddle, R Dyer, J Parsons, D Feltham, R Taylor, D Mellor, A Schofield, and R Linehan. STRaND-1: Use of a $500 smartphone as the central avionics of a nanosatellite. In *Proceedings of the 2nd International Astronautical Congress 2011,(IAC'11)*, 2011.

[18] James J Cockrell, Bruce Yost, and Andrew Petro. PhoneSat - the smartphone nanosatellite. Technical Report NASA FS-2013-04-11-ARC, ARC-E-DAA-TN8811, NASA Ames Research Center; Moffett

Field, CA United States, 2013. Available online: http://ntrs.nasa.gov/search.jsp?R=20140008908. Accessed 2016-06-19.

[19] Nexus 5. https://store.google.com/product/nexus_5. Accessed: 2016-06-12.

[20] Android Developers. http://developer.android.com/index.html. Accessed: 2016-02-25.

[21] Santiago Rodrigo Muñoz. A scalable distributed autonomy system for fractionated satellite missions. Bachelor thesis, Universitat Politècnica de Catalunya, 2016. Available online: http://hdl.handle.net/2117/83416. Accessed: 2016-06-19.

[22] Preempt RT frequently asked questions. https://rt.wiki.kernel.org/index.php/Frequently_Asked_Questions. Accessed: 2016-06-19.

[23] FreeRTOS. http://www.freertos.org/. Accessed: 2016-02-28.

[24] Preempt RT. https://rt.wiki.kernel.org/index.php/Main_Page. Accessed: 2016-02-28.

[25] Xenomai. http://xenomai.org/. Accessed: 2016-02-28.

[26] Jesper Bønding, Kasper F Jensen, Marc Pessans-Goyheneix, Morten B Tychsen, and Kasper Vinther. Software framework for reconfigurable distributed system on AAUSAT3. 2008.

[27] SUSE Linux enterprise server for high performance computing. https://www.suse.com/products/server/hpc. Accessed: 2016-06-11.

[28] Raspberry Pi. https://www.raspberrypi.org/. Accessed: 2016-06-11.

[29] Android is now used by 1.4 billion people. http://www.theverge.com/2015/9/29/9409071/google-android-stats-users-downloads-sales. Accessed: 2016-11-06.

[30] ART and Dalvik. https://source.android.com/devices/tech/dalvik/. Accessed: 2016-06-11.

[31] Android TV. https://www.android.com/tv/. Accessed: 2016-02-28.

[32] Android Auto. https://www.android.com/auto/. Accessed: 2016-02-28.

[33] Sung Wook Moon et. al. Implementation of smartphone environment remote control and monitoring system for Android operating system-based robot platform. In *Ubiquitous Robots and Ambient Intelligence (URAI), 2011 8th International Conference on*, pages 211–214, 2011.

[34] P. Bryant, G. Gradwell, and D. Claveau. Autonomous UAS controlled by onboard smartphone. In *Unmanned Aircraft Systems (ICUAS), 2015 International Conference on*, pages 451–454, 2015.

[35] iOS. http://www.apple.com/ios/. Accessed: 2016-06-06.

[36] Windows Phone. https://www.microsoft.com/en-us/windows/phones. Accessed: 2016-06-06.

[37] Luís Nogueira Cláudo Maia and Luís Miguel Pinho. Evaluating Android OS for embedded real-time systems. *6th International Workshop on Operating Systems Platforms for Embedded Real-Time Applications (OSPERT 2010)*, 2010.

[38] Bhupinder S Mongia and Vijay K Madisetti. Reliable real-time applications on Android OS. *IEEE Electrical and Computer Engineering Electrical and Computer Engineering*, 2010.

[39] L. Perneel, H. Fayyad-Kazan, and M. Timmerman. Can Android be used for real-time purposes? In *Computer Systems and Industrial Informatics (ICCSII), 2012 International Conference on*, pages 1–6, 2012.

[40] Igor Kalkov et. al. A real-time extension to the Android platform. In *Proceedings of the 10th International Workshop on Java Technologies for Real-time and Embedded Systems*, JTRES '12, pages 105–114, 2012.

[41] Wolfgang Mauerer et. al. Real-time Android: Deterministic ease of use. In *Proceedings of Embedded World Conference*, 2012.

[42] NASA. Core Flight Executive (cFE). http://opensource.gsfc.nasa.gov/projects/cfe/. Accessed: 2016-02-28.

[43] stress. http://linux.die.net/man/1/stress. Accessed: 2016-03-01.

[44] Cyclictest. https://xenomai.org/documentation/xenomai-2.6/html/cyclictest/. Accessed: 2016-02-28.

[45] Factory images for Nexus devices. https://developers.google.com/android/nexus/images. Accessed: 2016-06-12.

[46] Snapdragon 800. https://www.qualcomm.com/products/snapdragon/processors/800. Accessed: 2016-06-12.

[47] Kernel tree for Qualcomm chipsets. https://android.googlesource.com/kernel/msm.git. Accessed: 2016-06-12.

[48] Android Open Source Project. https://source.android.com/. Accessed: 2016-06-12.

[49] Building kernels. https://source.android.com/source/building-kernels.html. Accessed: 2016-06-12.

[50] GCC. https://gcc.gnu.org/. Accessed: 2016-06-12.

[51] Device Tree for dummies. https://events.linuxfoundation.org/sites/events/files/slides/petazzoni-device-tree-dummies.pdf. Accessed: 2016-06-12.

[52] initrd. http://www.tldp.org/LDP/Linux-Filesystem-Hierarchy/html/initrd.html. Accessed: 2016-06-12.

[53] mkbootimg. https://android.googlesource.com/platform/system/core/+/master/mkbootimg/. Accessed: 2016-06-12.

[54] split_boot. http://forum.xda-developers.com/showthread.php?t=2319018. Accessed: 2016-06-12.

[55] Running builds. https://source.android.com/source/running.html. Accessed: 2016-06-12.

[56] Building a Nexus 4 UART debug cable. https://www.optiv.com/blog/building-a-nexus-4-uart-debug-cable. Accessed: 2016-06-12.

[57] Phone connectors. https://en.wikipedia.org/wiki/Phone_connector_%28audio%29. Accessed: 2016-06-19.

[58] Using kgdb, kdb and the kernel debugger internals. https://www.kernel.org/pub/linux/kernel/people/jwessel/kdb/. Accessed: 2016-06-12.

[59] Porting Xenomai dual kernel to a new ARM SoC. http://xenomai.org/2014/09/porting-xenomai-dual-kernel-to-a-new-arm-soc. Accessed: 2016-06-13.

[60] Preempt RT HOWTO. https://rt.wiki.kernel.org/index.php/RT_PREEMPT_HOWTO#About_the_RT-Preempt_Patch. Accessed: 2016-06-13.

[61] Android kernel features. http://elinux.org/Android_Kernel_Features. Accessed: 2016-06-13.

[62] Control the Android emulator from the command line. https://developer.android.com/studio/run/emulator-commandline.html. Accessed: 2016-06-13.

[63] Build an RT application. https://rt.wiki.kernel.org/index.php/HOWTO:_Build_an_RT-application. Accessed: 2016-06-13.

[64] Inside the RT patch. http://elinux.org/images/b/ba/Elc2013_Rostedt.pdf. Accessed: 2016-06-13.

[65] Raspberry Pi hardware. http://elinux.org/RPi_Hardware. Accessed: 2016-06-14.

[66] Arnau Prat Sala. Scalable software architecture for the Android Beyond the Stratosphere satellite network. Bachelor thesis, Universitat Politècnica de Catalunya, 2015. Available online: http://hdl.handle.net/2117/79265. Accessed: 2016-06-19.

[67] R. Regupathy. *Unboxing Android USB: A hands on approach with real world examples*. SpringerLink : Bücher. Apress, 2014.

[68] Unix time. https://en.wikipedia.org/wiki/Unix_time. Accessed: 2016-06-02.

[69] JSON. http://json.org/. Accessed: 2016-06-02.

A. SDB Library example

```c
#include <mcs.h>
#include <sdb.h>
#include <sdb_group.h>

int hardware_start(void)
{
    int ret;
    unsigned char arguments[2] = {0x6, 1};
    MCSPacket *pkt_out;
    /* Set pin 0x6 on the Arduino board */
    MCSPacket *pkt_in = mcs_create_packet(MCS_PAYLOAD_DIGITAL_WRITE, 2,
                                          arguments, 0, NULL);
    ret = sdb_send_sync(pkt_in, &pkt_out);

    if (ret == 0) {
        if (pkt_out->type != MCS_TYPE_OK) {
            ret = mcs_err_code_from_command(pkt_out);
        }
        mcs_free(pkt_out);
    }

    mcs_free(pkt_in);

    return ret;
}

int hardware_stop(void)
{
/* Stop hardware */
}

int hardware_status(void)
{
/* Check hardware status */
}

MCSPacket *do_experiments(void)
{
/* Run experiments */
}

void check(MCSPacket *pkt_in, MCSPacket **pkt_out)
{
    int ret = hardware_status();
    if (ret == NOERROR) {
        *pkt_out = mcs_ok_packet(pkt_in);
    } else {
        *pkt_out = mcs_err_packet(pkt_in, ret);
    }
}

void init(MCSPacket *pkt_in, MCSPacket **pkt_out)
{
    int ret = hardware_start();
    if (ret == NOERROR) {
        *pkt_out = mcs_ok_packet(pkt_in);
    } else {
        *pkt_out = mcs_err_packet(pkt_in, ret);
    }
}

void run(MCSPacket *pkt_in, MCSPacket **pkt_out)
{
    *pkt_out = do_experiments();
}

void halt(MCSPacket *pkt_in, MCSPacket **pkt_out)
{
    int ret = hardware_stop();
    if (ret == NOERROR) {
        *pkt_out = mcs_ok_packet(pkt_in);
    } else {
        *pkt_out = mcs_err_packet(pkt_in, ret);
    }
}

int main(int argc, char **argv)
{
    sdb_register_callback(MCS_MESSAGE_CHECK, check);
    sdb_register_callback(MCS_MESSAGE_INIT, init);
    sdb_register_callback(MCS_MESSAGE_RUN, run);
    sdb_register_callback(MCS_MESSAGE_HALT, halt);

    sdb_connect("app", SDB_GROUP_HWMOD);

    while (1) {
        pause();
    }
}
```

Listing A.1: Example of a HWmod using the SDB Libraries and the MCS Libraries

B. MCS Library specification details

B.1 JSON full example

```
{
"command_list" : [
    {
    "name" : "procman_start",
    "description" : "Start the Process Manager",
    "nargs" : 0,
    "raw_data" : false,
    "type" : "message",
    "config" : {
        "destination" : "procman",
        "origin_groups" : ["syscore"],
        "destination_groups" : [],
        "response_size" : 0
        }
    },
    {
    "name" : "soc",
    "description" : "Get state of charge of the batteries",
    "nargs" : 0,
    "raw_data" : false,
    "type" : "state",
    "config" : {
        "destination" : "eps",
        "dimensions" : 1,
        "return_type" : "float",
        "unit" : "%",
        "dimension_name" : null
        }
    },
    {
    "name" : "arduino_get_pin",
    "description" : "Get the value of the given pin in the Arduino board",
    "nargs" : 1,
    "raw_data" : false,
    "type" : "payload",
    "config" : {
        "command" : 1,
        "parameters" : 6,
        "arguments" : ["@arg0"],
        "data" : null,
        "response_size" : 1
        }
    }
]
}
```

Listing B.1: Full example of a MCS configuration file

B.2 C translation output

```
/* AUTOGENERATED. DO NOT MODIFY */

#ifndef __AUTO_MCS_H
```

```c
#define __AUTO_MCS_H

#ifndef __MCS_H
#error "This header should not be included directly. Include mcs.h"
#endif

typedef enum MCSCommand {

} MCSCommand;

static const struct MCSCommandOptionsMessage mcs_command_message_list[] =
{

};

#define mcs_command_message_list_size

static const struct MCSCommandOptionsState mcs_command_state_list[] =
{

};

#define mcs_command_state_list_size

static const struct MCSCommandOptionsPayload mcs_command_payload_list[] =
{

};

#define mcs_command_payload_list_size

#endif
```

Listing B.2: Skeleton for the MCS C header file

The definitions of the elements from listing B.2 are specified in listing B.3, where each field name corresponds to the same field name in the configuration file.

```c
/* Datatypes for the autogenerated lists of commands */
typedef struct MCSCommandOptionsCommon {
    const char *name;
    unsigned short nargs;
    bool raw_data;
    int response_size;
} MCSCommandOptionsCommon;

struct MCSCommandOptionsMessage {
    struct MCSCommandOptionsCommon cmd;
    const char *destination;
    SDBGroup origin_groups[SDB_GROUP_MAX];
    SDBGroup destination_groups[SDB_GROUP_MAX];
};

struct MCSCommandOptionsState {
    struct MCSCommandOptionsCommon cmd;
    const char *destination;
    unsigned int dimensions;
};

struct MCSCommandOptionsPayload {
    struct MCSCommandOptionsCommon cmd;
    unsigned char command;
    unsigned char parameters;
    const char *arguments[2];
    const char *data;
};
```

Listing B.3: Structures used for the MCS C translation

```c
/* AUTOGENERATED. DO NOT MODIFY */

#ifndef __AUTO_MCS_H
#define __AUTO_MCS_H

#ifndef __MCS_H
#error "This header should not be included directly. Include mcs.h"
#endif
```

```

typedef enum MCSCommand {
    MCS_MESSAGE_PROCMAN_START       = 0,
    MCS_STATE_TEMPERATURE_ARDUINO   = 65536, /* 0x10000 */
    MCS_PAYLOAD_ARDUINO_GET_PIN     = 131072, /* 0x20000 */
} MCSCommand;

static const struct MCSCommandOptionsMessage mcs_command_message_list[] =
{
    {
    .cmd = {
        .name = "procman_start",
        .nargs = 0,
        .raw_data = false,
        .response_size = 0,
    },
    .destination = "procman",
    .origin_groups = {SDB_GROUP_SYSCORE},
    .destination_groups = {},
    },
};

#define mcs_command_message_list_size 1

static const struct MCSCommandOptionsState mcs_command_state_list[] =
{
    {
    .cmd = {
        .name = "soc",
        .nargs = 1,
        .raw_data = false,
        .response_size = 4,
    },
    .destination = "eps",
    .dimensions = 1,
    },
};

#define mcs_command_state_list_size 1

static const struct MCSCommandOptionsPayload mcs_command_payload_list[] =
{
    {
    .cmd = {
        .name = "arduino_get_pin",
        .nargs = 1,
        .raw_data = false,
        .response_size = 1,
    },
    .command = 1,
    .parameters = 6,
    .arguments = {"@arg0"},
    .data = NULL,
    },
};

#define mcs_command_payload_list_size 1

#endif
```

Listing B.4: Full example of an auto-generated MCS C header