Scalable Software Architecture for the Android Beyond the Stratosphere Satellite Network

A Degree Thesis

Submitted to the Faculty of the

Escola Tècnica d'Enginyeria de Telecomunicació de Barcelona

Universitat Politècnica de Catalunya

by

Arnau Prat Sala

In partial fulfilment

of the requirements for the degree in

ENGINYERIA DE SISTEMES ELECTRÒNICS

Advisors:

Elisenda Bou-Balust

Carles Araguz López

Reporting advisor (ponent): Eduard Alarcón

Barcelona, July 2015

UNIVERSITAT POLITÈCNICA
DE CATALUNYA
BARCELONATECH

Abstract

The aim of is project was to conceive, design and develop part of the software for the Android Beyond the Stratosphere project, which is being developed at the Technical University of Catalonia UPC BarcelonaTech, aiming to design a nano-satellite system based on an Android phone. This thesis has focused on the design and implementation of the ABS software architecture and the development of a Software Development Kit (SDK) for this platform. This SDK will allow for apps being uploaded to the satellite while in orbit. These apps, developed by the community, will run on the top of the custom software architecture and will have all the necessary interfaces to the smartphone hardware and on-board payloads. An Arduino Mega ADK board (open-source hardware) will interface the different payloads with the phone, being a low-cost, highly-configurable, fast-development platform for nano-satellites. The software has been designed for the current platform (1 phone) but it has been taken into account modularity and scalability to enable multiple-unit architectures such as satellite constellations or fractionated-spacecrafts.

Resum

L'objectiu d'aquest projecte era concebre, dissenyar i desenvolupar part del software pel projecte Android Beyond the Stratosphere que està sent desenvolupat a la Universitat Politècnica de Catalunya i que aspira a dissenyar un nanosatèl·lit basat en un mòbil Android. Aquesta tesis s'ha centrat en el disseny i desenvolupament d'una arquitectura de software y el desenvolupament d'un Software Development Kit (SDK) per aquesta plataforma. Aquest SDK permetrà pujar aplicacions al satèl·lit mentres aquest estigui en òrbita. Aquestes aplicacions, desenvolupades per la comunitat, s'executaran a sobre de la nostra arquitectura de software i tindrà totes les connexions necessàries amb el hardware del telèfon i càrregues útils. Un Arduino Mega ADK (hardware de codi obert) connectarà les diferents càrregues útils amb el telèfon, essent una plataforma de baix cost i altament configurable. El software s'ha dissenyat per la plataforma actual (1 telèfon) però s'ha tingut en compte modularitat i escalabilitat per permetre múltiples unitats tals com constel·lacions de satèl·lits o satèl·lits fraccionats.

UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH

telecom BCN

Resumen

El objetivo de este proyecto fue concebir, diseñar y desarrollar parte del software para el proyecto Android Beyond the Stratosphere que está siendo desarrollado en la Universidad Politécnica de Cataluña y que aspira a diseñar un nanosatélite basado en un móvil Android. Esta tesis se ha centrado en el diseño y desarrollo de una arquitectura de software y el desarrollo de un Software Development Kit (SDK) para esta plataforma. Este SDK permitirá subir aplicaciones al satélite mientras éste esté en órbita. Estas aplicaciones, desarrolladas por la comunidad, se ejecutarán sobre nuestra arquitectura de software y tendrá todas las conexiones necesarias con el hardware del teléfono y cargas útiles. Un Arduino Mega ADK (hardware de código abierto) conectará las diferentes cargas útiles con el teléfono, siendo una plataforma de bajo coste y altamente configurable. El software ha sido diseñado para la plataforma actual (1 teléfono), pero se ha tenido en cuenta la modularidad y escalabilidad para permitir múltiples unidades tales como constelaciones de satélites o satélites fraccionados.

Acknowledgements

I would like to thank all the people that made this work possible. Foremost, I would like to express my sincere gratitude to Prof. Eduard Alarcón, Elisenda Bou and Carles Araguz for being the advisors of my project and giving me advice every time I need it. Also Marc Marí, from which I learned a lot. Also Miquel Vaquero, for helping me testing the SDK, and the rest of PAE students who have collaborated with the project. Finally I also want to thank my family and my friends for giving me support when I needed it.

Revision history and approval record

Revision	Date	Purpose
0	05/05/2015	Document creation
1	20/06/2015	Document revision
2	06/07/2015	Document revision

DOCUMENT DISTRIBUTION LIST

Name	e-mail
Arnau Prat	arnauprat1@gmail.com
Eduard Alarcón	eduard.alarcon@upc.edu
Elisenda Bou	elisenda.bou@upc.edu
Carles Araguz	carles.araguz@gmail.com

Written by:		Reviewed and approved by:	
Date	08/07/2015	Date	08/07/2015
Name	Arnau Prat	Name	Eduard Alarcón
Position	Project Author	Position	Project Supervisor

Table of contents

Abstract ii
Resum iii
Resumen iv
Acknowledgements vi
Revision history and approval record vii
Table of contents viii
List of Figures x
List of Tables xi
1. Introduction 1
1.1. Context: Android Beyond the Stratosphere project 1
1.2. Project Background 1
1.3. Objectives 2
1.4. State of the art 2
2. ABS Software Architecture 4
2.1. Introduction 4
2.2. System Data Bus 6
2.2.1. MCS Packets and SDB router 6
2.2.2. SDB USB (USB daemon) 6
2.2.3. System States database 7
2.2.4. Result Files database 8
2.3. PayloadApp SDK 10
2.3.1. PayloadApp API 10
2.3.2. PayloadApp framework 12
2.4. Arduino firmware 16
2.4.1. Protocol 16
2.4.2. Firmware 17
2.4.3. USB communication 19
2.5. App_ctrl library (Appmods) 20
2.5.1. Introduction 20
2.5.2. Implementation 20
2.5.3. Application permission database 22
3. Results 24
4. Conclusions and future extension 26

Bibliography 27

Annexes 28

Glossary 31

List of Figures

Figure 1: Subsystems on the architecture 4
Figure 2: ABS software architecture 5
Figure 3: System Data Bus block diagram 6
Figure 4: Example configuration System State 8
Figure 5: Functions SystemStates abs_db library 8
Figure 6: File state evolution 9
Figure 7: Functions ResultFile abs_db library 9
Figure 8:Communication Android applications - ABS Architecture 10
Figure 9: Internal implementation "readSPI" function 11
Figure 10: Screenshot of the PayloadAPP SDK project 12
Figure 11: Functions SDK framework 13
Figure 12: State diagram functions SDK framework 13
Figure 13: PayloadApp SDK UML diagram 15
Figure 14: USB packets basic structure 16
Figure 15: Arduino firmware workflow 18
Figure 16: Example of USB packet 19
Figure 17 *APP_ctrl* library in context of the architecture 20
Figure 18: Android Manifest with custom permissions 21
Figure 19: Steps to extract permissions from manifest 21
Figure 20: Functions App_ctrl library 22
Figure 21: Functions App_ctrl abs_db 23
Figure 22: Path followed by the command analogRead 24
Figure 23: Latency histogram 1 service 24
Figure 24: Latency multiple services 25
Figure 25: Test application for ABS 25

List of Tables

Table 1 SystemState database table 7

Table 2: ResultFiles database table 9

Table 3: Methods SDB class 11

Table 4: USB packets structure 17

Table 5: Basic structure of the ApplicationPermissions table 22

1. Introduction

In this section, the Android Beyond the Stratosphere project in the context of which this work is frameworked will be introduced. After that, the project background and its objectives will be stated. Finally, a survey of the current state of the Art on this topic will presented, to motivate the interest and impact of the project.

1.1. Context: Android Beyond the Stratosphere project

The aim of the Android Beyond the Stratosphere (ABS) project is to develop an open-source, standardized, modular nano-satellite platform based on commercial of the shelf components and open standards. The project seeks to explore the satellite-on-a-phone architecture to enable a low cost modular nano-satellite based on an Android phone.

The final goal of the ABS project is to develop the whole system, with especial efforts on the software (since the hardware is already been given, and minor modifications have to be done to enable a smartphone with the extra capabilities of a nano-satellite). The software has been designed for a single satellite-mission but modularity and scalability have been taken into account in order to enable multiple-unit architectures such as satellite constellations or fractionated spacecrafts.

Keeping the operating system (OS) on the phone (Android OS) will allow for apps being uploaded to the satellite while in orbit. These apps, developed by the community, will run on the top of the custom software architecture and will have all the necessary interfaces to both the smartphone hardware and mission payloads.

An Arduino Mega ADK board (open-source hardware) will interface the different payloads with the phone. Such board is a low-cost, highly-configurable, fast-development platform enabling to add payloads and hardware modules to the phone.

Towards the objective to allow developers and users to create their own applications and test them on the satellite, it has been developed an SDK that will enable Android developers access the different satellite payloads and subsystems in an easy way.

1.2. Project Background

The Android Beyond the Stratosphere project follows the previous work carried out at the Laboratory of Small Satellites and Payloads of the Technical University of Catalonia UPC BarcelonaTech, and is encompassed under the Conceive Design Implement and Operate (CDIO) initiative where different groups of students, collaborate in the development process of CubeSat nanosatellites.

The project officially started on fall 2013, when the project proposal presented by Prof. Eduard Alarcón and the PhD candidate Elisenda Bou, was awarded with one of the Google Faculty Research Awards.

I joined the project as an Advance Engineering Project student the following semester, helping the team with the development of an Android Real-Time OS and the conception of the ABS software architecture. After finishing the course, I was offered the opportunity to continue with the task of developing the software architecture along with the

implementation of an SDK for the Android platform as a senior student.

The tasks presented in this work were decided with my supervisors and me before starting the project and were designed as a result of my previous development activities.

1.3. Objectives

The ABS software architecture is based on the existing ^{3}Cat1 architecture [1] (the first nano-satellite developed at the UPC), but seeks to take a step beyond and use its predecessor's flexibility to work together with Android.

Most of the previous architecture has had to be revisited and redesigned, while other parts have had to be built from the scratch. Most of these parts are related with Android and Arduino, where this project has been more focused on.

The objective of this work has been to develop and design part of the software for the ABS project. Focusing on 1) design and implementation of the ABS software architecture and 2) design and implementation of an SDK for this platform, which will allow to develop Android applications to control payload platforms.

This report includes the design and implementation of four of the modules of the ABS architecture. All the details about the design and implementation of these modules along with the architecture are explained in the next chapter.

Modules designed and implemented in this thesis are, namely:

1) Software Architecture:

- System Data Bus: System command router and data transfer arbitrator that interfaces the rest of modules.

- Arduino firmware: Firmware for Arduino, which interfaces the Android phone with the attached payloads.

- App_ctrl library: Library used by the modules on the custom architecture to control Android applications.

2) PayloadApp SDK.

1.4. State of the art

The development costs and time of a conventional satellite can be very high, and hence, recently, many efforts have been made to minimize the cost by developing smaller satellites. These small satellites serve as low-mass platforms that can be sent to space, allowing corporations, universities and even individuals low-cost access to space. These "miniaturized satellites" can be classified based on their mass, where the most popular classes are: nano-satellite (1 Kg to 10 Kg) and pico-satellite (100g to 1Kg) [2].

To help with the design process of these satellites, CubeSat standard [3] was developed in California Polytechnic State University and Stanford University. This standard defines a

cube-shaped satellite with a nominal length of 10 cm per side and a mass of no more than 1.33 Kg, which typically uses commercial components for its electronics.

Recently, various projects have appeared with the vision of building CubeSats based on mobile phones, known as PhoneSats [4]. Nowadays, smartphones are incredible pieces of engineering capable of a lot of things: communications, built-in sensors, computing, energy management…

Android has also become a very attractive platform for space applications [5], because it's an open source platform, which makes it very easy to customize the software. For example, NASA has an on going project [6], which uses a Nexus S phone running Android OS as on-board computer. However, achieving a complete autonomy system is still a far fetch objective and the results only showed the foundation of such architectures.

In this project, the phone is loaded with special experimental Android apps, which act as the brain of the satellite. In this approach, the custom software runs on top of the Android layer, while in the ABS case, the custom software architecture is built on top of the Linux Kernel and runs side by side with Android, being a more reliable solution.

These projects have been mainly focused hitherto on the hardware, while it has been scarcely approached from the software perspective. There is almost no previous work in general-purpose software architectures targeting these platforms and no previous work has been found on platforms that allow developers to create their own applications oriented for space, such as the Software Development Kit (SDK) presented in this work.

The use of Arduino in space is not new either. Some projects have also appeared with the idea of using it (e.g. Ardusat [7]). However in the ABS project it plays a secondary role, using it as a platform to interface the phone with the attached payloads.

2. ABS Software Architecture

2.1. Introduction

The ABS software architecture is the main controlling software of an ABS module/satellite. It comprises several processes and libraries where each of them is devoted to a particular task. While most parts of the ABS architecture have been developed in C/C++, taking advantage of the fact that Android is a Linux-based OS, others parts have been developed in Java (PayloadApp SDK) and Arduino language (Arduino firmware).

This custom architecture is deployed on top of the Linux OS and operates together with Android components, which run on top of a Java Virtual Machine (JVM) named Dalvik Virtual Machine (VM), which execute the phone's functionalities and Android apps.

The architecture can be divided in three main parts: 1) The PayloadApp SDK: interface the apps running on top of the Android layer with the ABS architecture. 2) The Arduino firmware: interfaces the attached payloads with the ABS architecture. 3) The ABS system manager: is in charge of managing the system and controls safety rules.

Communication between the ABS architecture and the Arduino firmware is performed through USB. While the communication between the PayloadApp SDK and the rest of the inner processes on the ABS architecture is performed through sockets.

Below is a diagram of the different parts/subsystems of the architecture:

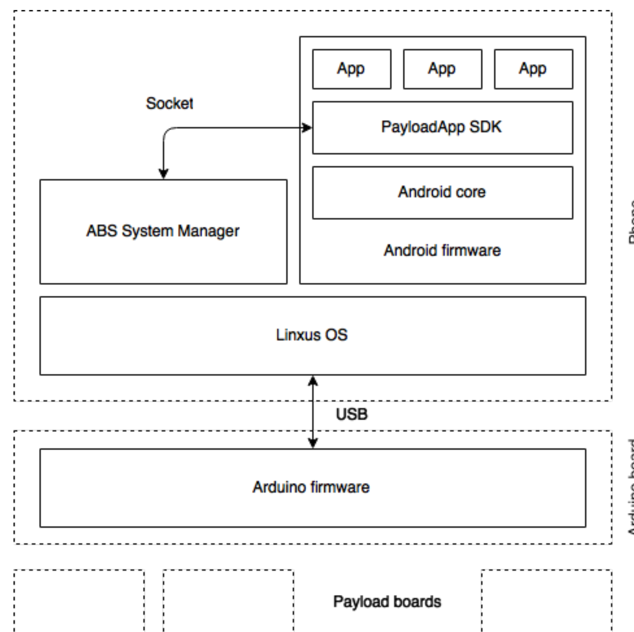


Figure 1: Subsystems on the architecture

The software architecture has been designed separating the different processes and libraries that conforms it in modules. Considering the modules as black boxes makes the software much more changeable, maintainable and minimizes error propagation. It also enables scalability and makes the software general-purpose.

During the design of the software architecture, this has suffered numerous changes, involving many design decisions. This caused that the design stage of the project required slightly more time than envisaged. This was not critical since providing a better software design helped reducing the implementation time.

On top of the architecture (ABS system manager) are the System Core, which manages the system at its highest level and the Process Manager, which converts the high-level goals in low-level processes. Next are the System Data Bus and SDB USB, which interface the different modules on the architecture between them. Following, we have the Hardware-dependent modules (HWmods) and the Distributed System Layer, which connects the architecture with the hardware on the phone and other satellites. Finally, the PayloadApp SDK and the Arduino firmware, interface the architecture with the apps running on top of the Android layer and the attached payloads respectively.

Below is the final design of the ABS software architecture:

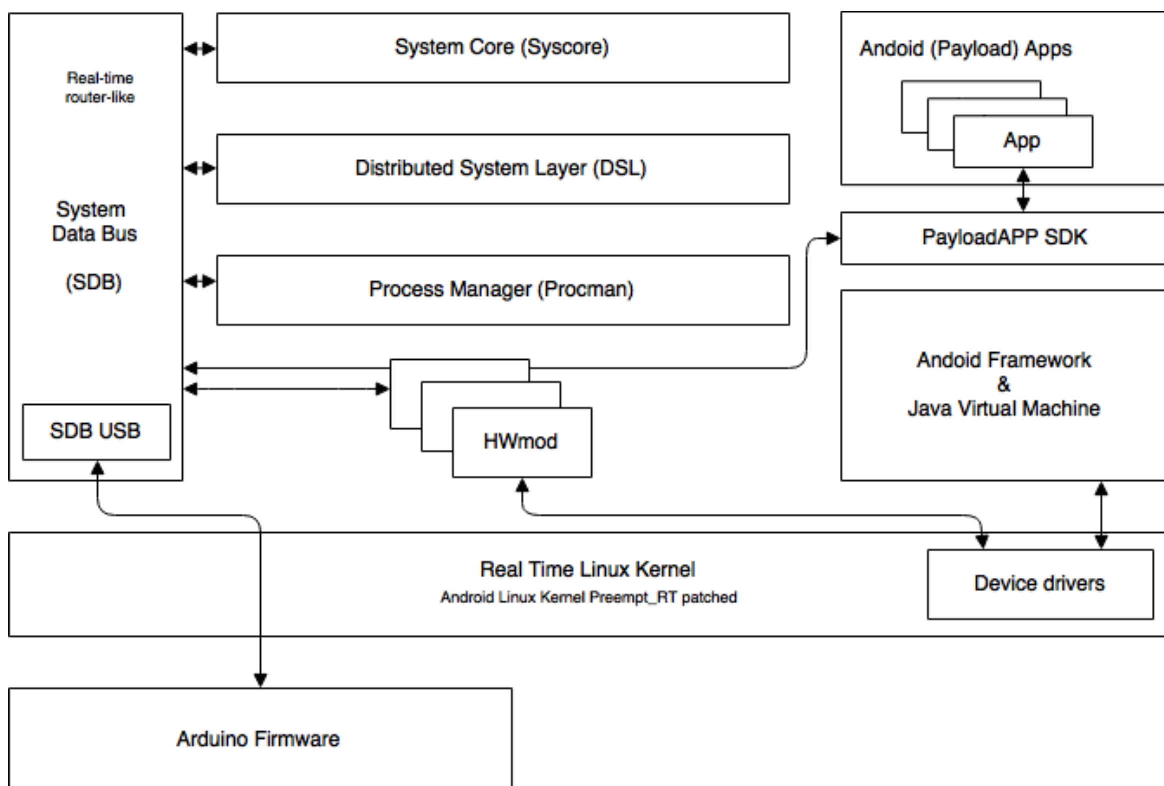


Figure 2: ABS software architecture

The final ABS software architecture (SA) is composed of a total of eight modules. Next sections detail the design and development of four of the modules addressed in this thesis: System Data Bus, PayloadApp SDK, Arduino firmware and App_ctrl library.

2.2. System Data Bus

The System Data Bus is used as a data router or distributor, to send commands to one module to another. In addition to this, the SDB also manages the communication with the Arduino subsystem (SDB USB) and handles global persistent data stored in the *System State* and *Result Files* databases. All this is described below.

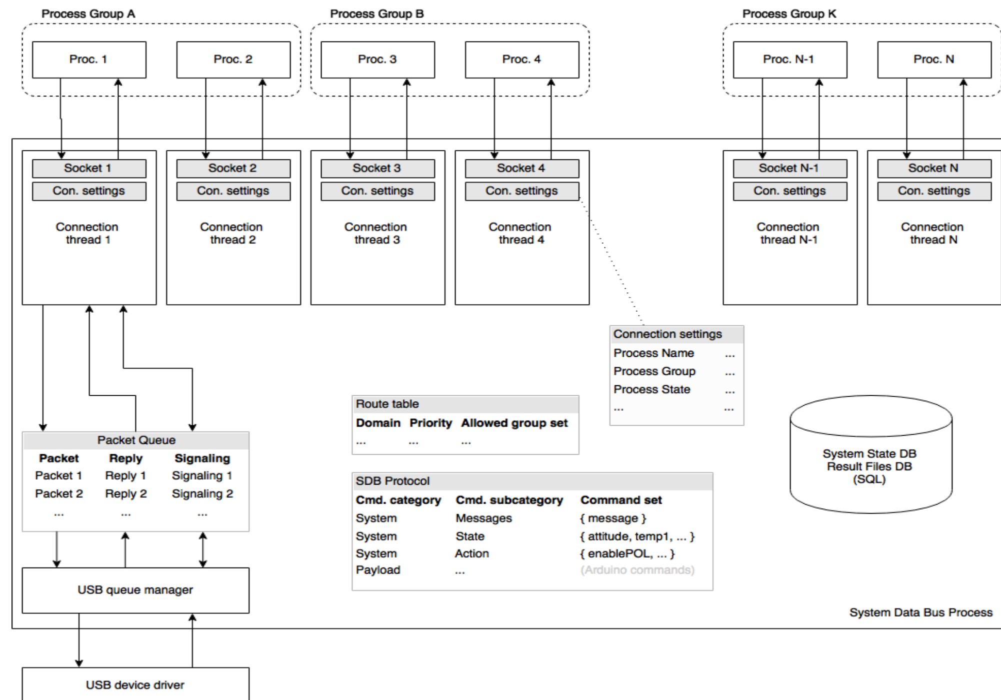


Figure 3: System Data Bus block diagram

2.2.1. MCS Packets and SDB router

The SDB Modular Command System (MCS) is a custom specification, which allows to define the packets (commands) that the SDB accepts. All the SDB commands are defined in a system configuration file, which is processed by the MCS to automatically generate the necessary functions, structures and definitions at compile-time.

These commands can then be used at many levels of the SA and constitutes an important part of the ABS project. The implementation of the MCS and the design of the SDB core have been carried out by Marc Marí and is not covered in this work.

2.2.2. SDB USB (USB daemon)

The SDB USB is the sub module of the SDB that manages the communication with the Arduino subsystem. Its task is to forward those commands that have to go to the Arduino, converting MCSPackets into USBPackets and then sending them through the USB.

The SDB USB should not only send the packets by their order of arrival (first come, first served) but also has to take into account the priority of these packets (the priority depends on the module that sends the packet). This allows critical commands (e.g. system control) to be sent before than the regular ones (e.g. sensor data retrieval).

A priority queue has been implemented which sorts the commands waiting to be sent by their order of priority. As shown on the Arduino Firmware section, this implements a request-reply communication model, hence before sending a new command the SDB USB needs to wait for the response to the previous one, convert it into an MCS Packet and send it to the request module. Once this is done, the next command is sent.

In order to establish the communication with the Arduino board, the same protocol used by the firmware had to be implement on the SDB USB (*Android Open Accessories protocol*). In a normal situation this communication would be performed using the standard *APIs for Android*. However, these APIs can only be used from Android, and the communication had to be done writing and reading (sending and receiving) directly to the driver that controls the *USB accessory communication*: */dev/usb_accessory* [8].

2.2.3. System States database

System States are values or measures generated by a module or sensor on the Architecture, which are of interest to the mission, such as the value of a temperature sensor or the battery state. The System States will be stored in the *SystemState* database. Table 1 shows the basic structure of a System State in the database.

Every System State (e.g., temperature, battery state…) will have their own table and will contain an entry for each of the measures taken so far. Keeping all the measures will allow, for example, study the evolution of a certain state over a certain period of time.

Column	Allowed types	Description
ID	Integer	Pirmary key for the table
Value 1	Integer/Real/Text	Value 1
Value …	Integer/Real/Text	Value …
Value N	Integer/Real/Text	Value N
Time	Integer	UNIX time at which the value was stored

Table 1 SystemState database table

Note that each table can have more than one *Value* column (e.g., the table of the System State Gyroscope would have 3 *Value* columns for components X, Y and Z respectively). The type of the values can also be different for each column (although in the gyroscope example the three of them could be of type Real).

New System States can be defined making use of the Modular Command System mentioned before. To add a new System State, it has to be define the name of the measure, the number of values, the units, expire time of the measure, update function…

Below is an example of how to create a new System State called “temperature_arduino”, which will contain the value of a temperature sensor on the Arduino board (in Kelvins). In this example in can also be seen how this state would only be accessible by the System

Core (*"expire_group" : [{"Syscore"}]*) and the command that will be used to update the State will be reading the analog pin 1 on the Arduino board (command *analogRead(1)*).

```
{
        "name" : "temperature_arduino",
        "description" : "Get temperature from the sensor in the Arduino board",
        "nargs" : 1,
        "raw_data" : false,
        "type" : "state",
        "config" : {
                "update_function" : "analogRead(1)",
                "dimensions" : 1,
                "return_type" : "float",
                "unit" : "K",
                "dimension_name" : null,
                "expire_group" : [{"Syscore"}]
                }
}
```

Figure 4: Example configuration System State

The access to the database by the different modules is provided by the *abs_db* library functions listed below:

```
/* Add a system value */
MCSPacket addSystemStates(MCSPacket *pkg);
/* Get a system value */
MCSPacket getSystemStates(MCSPacket *pkg);
```

Figure 5: Functions SystemStates abs_db library

2.2.4. Result Files database

The *ResultFiles* database keeps track of the different files that could be generated in the system. This database stores information about the files stored in the satellite keeping track of their name, owner or state (i.e. the file has been sent (*Sent*), the file has just been received (*New received*), the file has been deleted (*Deleted*), etc.) Table 2 shows the table that defines this database.

Column	Type	Description
Path	Text	Path of the File
Owner	Text	Module owner of the file
Time	Integer	Time when file created
State	Integer	Current State
Priority	Integer	Priority
Expire_time	Integer	Expiring time

Table 2: ResultFiles database table

Below is a finite state diagram of the field *State* of the *ResultFiles* database, which indicates the file *State* evolution of a file.

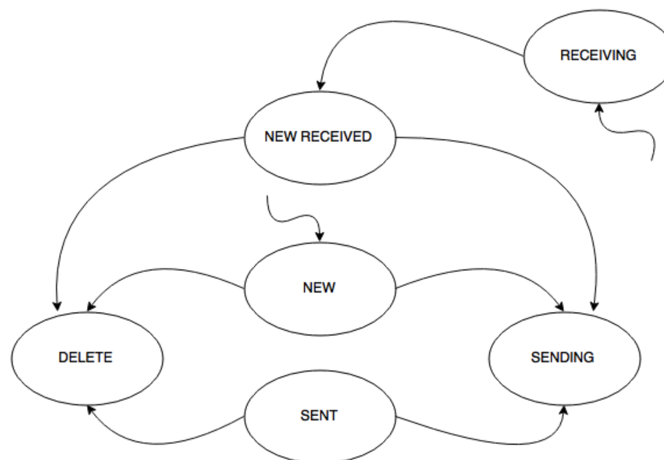


Figure 6: File state evolution

The access to the database by the different modules is also provided by the *abs_db* library, which will specify the following functions:

```
/* Add a result file */
int addResultFile(abs_file *file);
/* Get a result file */
ABSFile *getResultFile(char *filename);
```

Figure 7: Functions ResultFile abs_db library

2.3. PayloadApp SDK

The PayloadApp SDK is a developer framework, which will allow the future community of developers to build Android apps that make use of the payloads and resources on the satellite. The PayloadApp SDK sits between the Android applications and the ABS architecture and connects both of them.

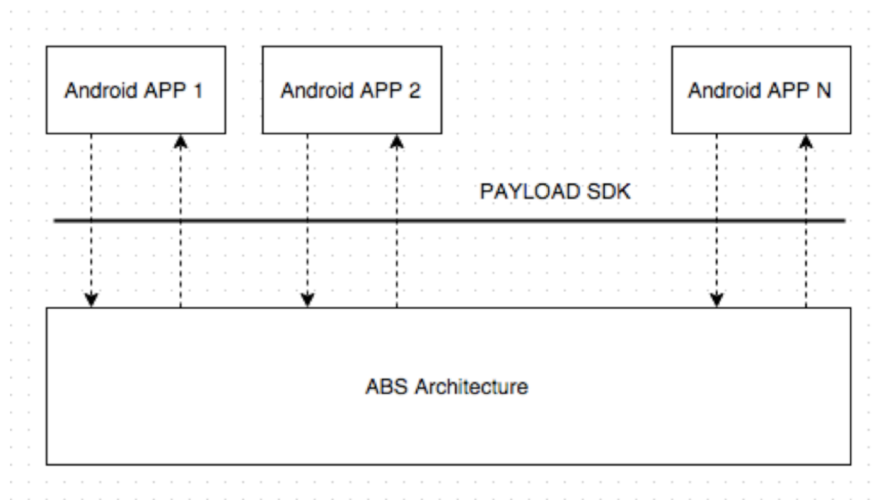


Figure 8:Communication Android applications - ABS Architecture

The SDK presented in this report implements more than 50 functions arranged in different classes (e.g., Arduino, Attitude, Energy, Orbital State, etc.) providing high-level definition functions to control the spacecraft. Also it provides a complete framework for developers to build their own Android applications for nano-satellite platforms.

The full list of functions implemented on the SDK can be found on Annex 1.

2.3.1. PayloadApp API

The PayloadApp API is a collection of functions and classes (see UML diagram on page 25), which enables Android apps to execute system-commands on the architecture through wrapper functions (e.g., getAttitude, getOrbitalState, readSPI, digitalRead etc.), often named the same as the command they invoke.

Internally, the SDK does not implement any of these commands per se. Instead, when the SDK needs to execute a command, it sends a message to the Architecture requesting to do it (similar to a Kernel System Call on Unix).

These calls are invoked by wrapper functions and are the ones that the users will ultimately use. For security reasons, the developers cannot send commands directly to the architecture and must always use these functions.

Following is an example of the internal implementation of one of these functions:

```
Byte[] readSPI(int slaveSelectPin)
{
    /* Create the SDBpacket to be sent */
    SDBPacket pkg = new SDBPacket(READ_SPI, slaveSelectPin, NULL);
    /* Send packet and wait for a response */
    SDBPacket res = sdb.send(pkg);
   /* Return the data of the response packet */
    return res.getData();
}
```

Figure 9: Internal implementation "readSPI" function

For this to work, the PayloadApp SDK must be able to communicate with the custom architecture. A class called SDB (System Data Bus) is in charge of managing the communication. The list of methods of this class are shown below:

Method	**Description**
- connect	It connects to the ABS architecture.
-disconnect	It disconnects from the ABS architecture.
+send	Sends a packet to the system and waits for another packet in response.
+register	Register a callback function that will be executed when certain pkt arrive.

Table 3: Methods SDB class

Although previous versions of this class implemented these methods entirely in Java, the final implementation uses calls to a library written in native code. This design choice was motivated by the following reasons:

1) Simplicity: it can be reused the same libraries that were developed to communicate the rest of the modules of the architecture between them.

2) Consistency: By doing this it avoids to implement the same code for Java and C, eliminating code duplicity and thus minimizing errors.

To include native code in the SDK it was used the Android NDK [9] (Native Development Kit), which enables the use of native code on an Android project, and JNI [10] (Java Native Interface), which defines a way for Java to interact with native code.

The screenshot below shows the PayloadApp SDK project implemented using Android Studio [11]. On the left side the different classes and files that form the project are shown, which combine Java (Java) and native code (JNI). The right side shows part of the code of the native function *send* of the SDB class, where it can be seen how JNI used to obtain the variables of a Java object of the class SDBPacket in C.

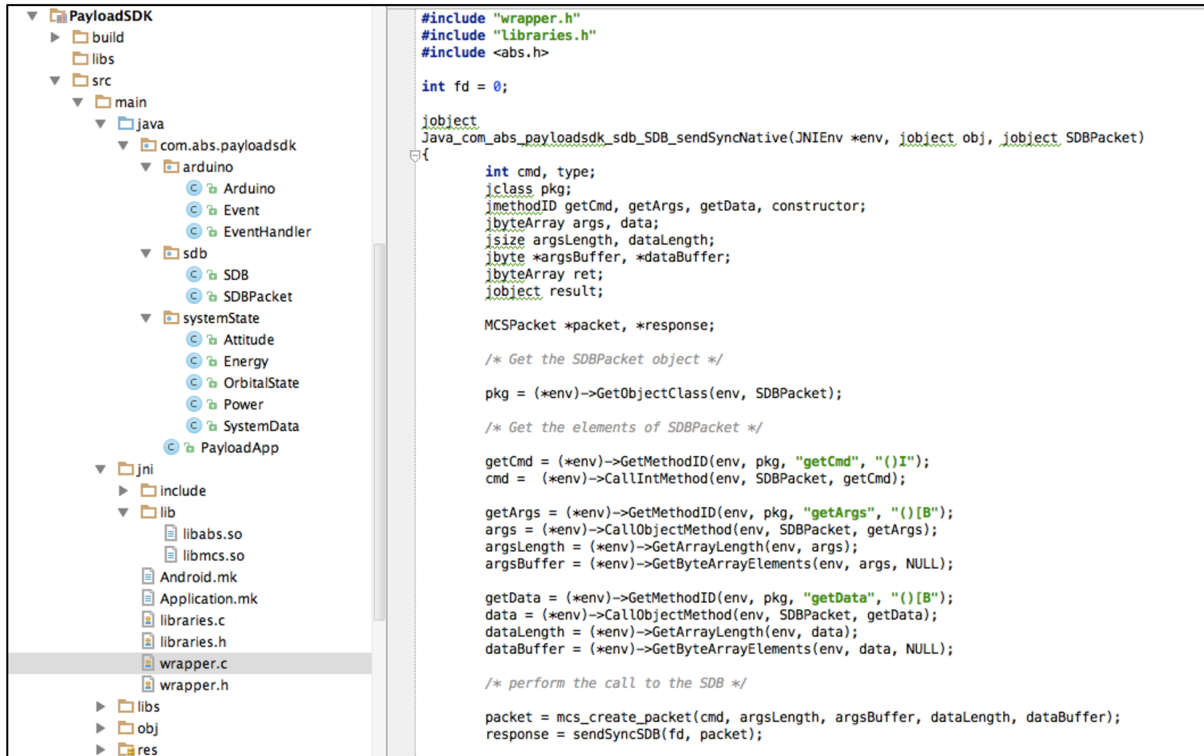


Figure 10: Screenshot of the PayloadAPP SDK project

2.3.2. PayloadApp framework

The PayloadApp SDK provides a framework for final developers, which allows them to build their own applications to control mission-specific payloads. These apps will *extend* the custom class *PayloadApp*, which, at the same time extends from the class *Service*.

A *Service* [12] is an Android application component that runs in background to perform long operations and has no user interface (UI). Since 1) no UI is need it in space and 2) the Android OS will only destroy a Service in a critical situation, it was decided that these apps will be build around a Service.

An example of a Payload App could be an app that takes pictures of the Earth if the satellite is at a certain position (e.g. takes photos when it passes over Barcelona), making use of a GPS attached to the Arduino.

To create a Payload App, the first step is to import the PayloadAPP SDK into our Android project (File -> Module -> Import PayloadApp.aar). Once this is done, it can be created a class that extends from com.abs.payloadsk.PayloadAPP. At this point, the developer will have access to all the functions and classes mentioned above and four public methods, where our code should be written (*check*, *init*, *run* and *halt*).

Each of these 4 functions have different functionalities and are executed at different points in time by the ABS architecture. The table below lists the Payload App basic interface and describes the meaning of each function, and a state diagram of the order in which the ABS architecture (Process Manager) can execute these functions.

Function	Description
Check	Check the state of the hardware and software used in the *run*.
Init	Initialize the hardware and software for the run.
Run	Start performing the routines.
Halt	Stop all routines and reset devices.

Figure 11: Functions SDK framework

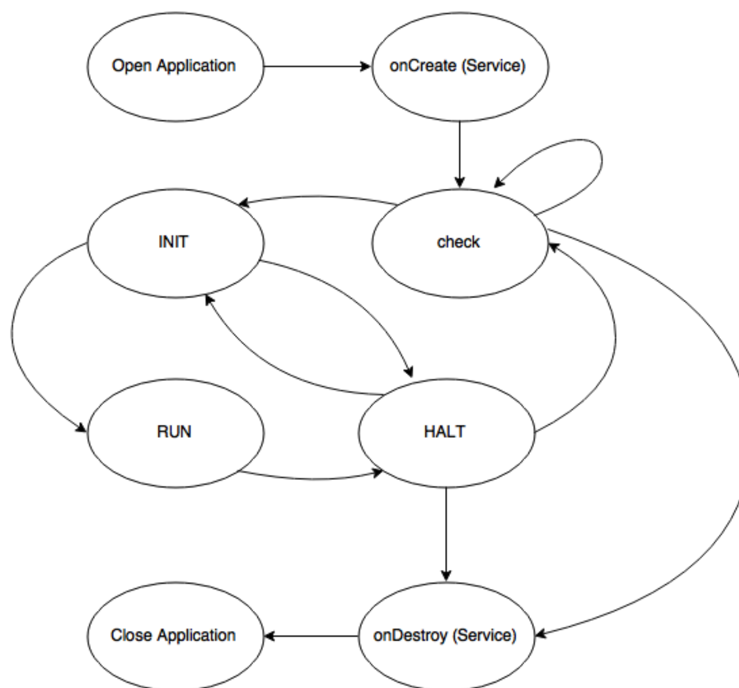


Figure 12: State diagram functions SDK framework

Below is an example of a Payload App application. When this app runs continuously checks a temperature sensor in the Arduino board and, if it is greater than certain threshold turns on an LED and if it is below it turns it off.

The function *check* will determine whether the sensor is working properly or not. The function *init* will set the LED in its initial state. The function *run* will perform the main routine commented before. Finally the function *halt* will set the LED into its initial state.

```
package com.abs.payloadapp;

/* The class needs to extend from PayloadApp */
public class MyPayloadApp extends PayloadApp {

    Arduino arduino;
    private static final int TEMPERATURE_SENSOR = 10; // Definitions
    private static final int LED = 5, LOW = 0, HIGH = 1;

    public MyPayloadApp()
    {
        arduino = this.getArduino();          // Get arduino object
    }

    @Override
    public int check()                        // Check the state of all
    {
        if(!arduino.analogRead(TEMP_SENSOR)) { //Check if temp sensor is working
            return -1;                        //Check failed (return -1)
        } else {
            return 0;                         //Check passed (return 0)
        }
    }

    @Override
    public int init()                         // Initialize all for run
    {
        arduino.digitalWrite(LED, HIGH);      // Put LED initial state (ON)
        return 0;                             // Always need to return 0
    }

    @Override
    public int run()                          // Start main routine
    {
       while(1) {                             // If temp sensor above threshold
          if(arduino.analogRead(TEMPERATURE_SENSOR) > 100) {
              arduino.digitalWrite(LED, HIGH); // Turn on the LED
          } else {
              arduino.digitalWrite(LED, LOW);  // Turn off the LED
          }
        }
        return 0;                             // Always need to return 0
    }

    @Override
    public int halt()                         // Stop main routine and reset
    {
        arduino.digitalWrite(LED, LOW);       // Put LED on orig state (OFF)
        return 0;                             // Always need to return 0
    }
}
```

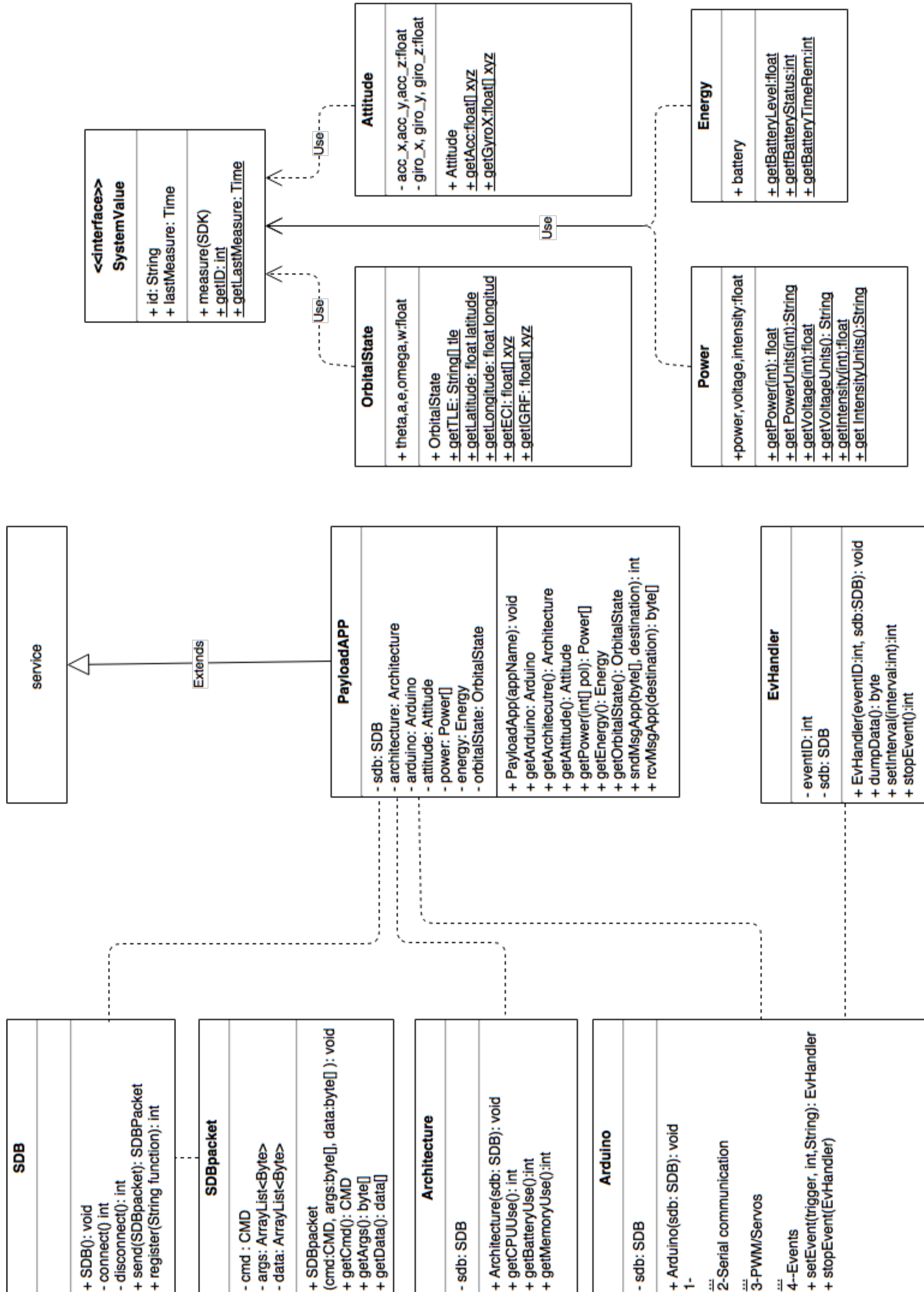


Figure 13: PayloadApp SDK UML diagram

2.4. Arduino firmware

One of the main parts of the ABS project is the Arduino subsystem. This part is in charge of the communication between the phone and the different payloads, where the Arduino platform acts as an interface Payloads-Phone (1 phone - N payloads).

The idea is to be able to control the Arduino (and therefore the attached Payloads) directly from the phone without writing code from the Arduino. Instead, a very generic firmware interprets and executes the commands sent by the phone.

Next section details how the firmware works and the protocol designed to communicate the Android phone and the Arduino via USB.

2.4.1. Protocol

The protocol aims to be as generic as possible, allowing creating any type of command. With the same packet structure one can start an I^2C communication, read the value from a digital pin or even create an event to read a pin every N milliseconds.

This uses a request-reply communication model, the phone sends a command to the Arduino board, which receives and processes, and ultimately, returns a message in response. This response packet can be of the type: ok, ok_data or error.

The protocol also allows the firmware to send messages to the phone without a previous request, although this is restricted to very special events (e.g. if a critical error occurs in the Arduino ADK board and it has to be notified to the Android phone.)

The basic structure for the USB packets is shown below:

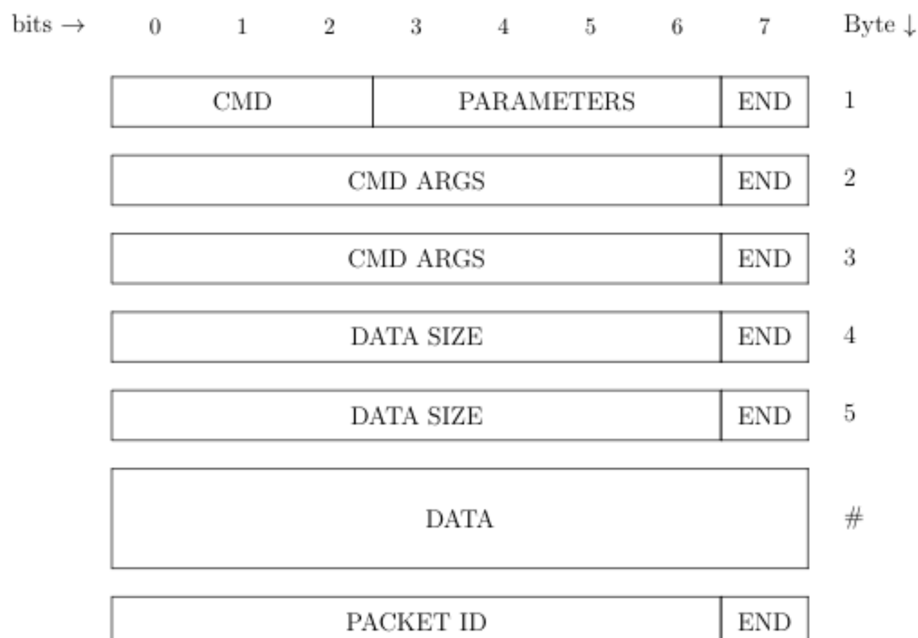


Figure 14: USB packets basic structure

Name	# bits	Description
CMD	3 bits	Defines the type of command to be performed.
Parameters	4 bits	Defines the specific command to be performed.
CMD Args	14 bits	Arguments that a command takes (arg0, arg1).
Data Size	14 bits	Size in bytes of the field Data.
Data	# bytes	Data that a command takes.
Packet ID	7 bits	Packet identifier.
End	1 bit	Indicates the end of a packet. 0 in the last, 1 the rest.

Table 4: USB packets structure

2.4.2. Firmware

The Arduino firmware waits for commands on the USB port. After receiving a command, the firmware executes it. This operation is done over and over again and constitutes the main loop of the Arduino firmware.

In addition are the so-called *events*. *Events* are actions that are executed at very specifically timed intervals. These actions are managed independently from the main loop and are controlled by a timer interrupt. *Events* can be created from the phone with the command *createEvent* along with the action and its repetition time.

Since the execution of an event inside an interrupt routine could be susceptible of nested interrupts, these actions are performed within the main loop and the event routine only decides when they should be executed. When the event routine determines that the time that has elapsed since the last execution of an event is greater or equal than the specified repetition time, it signals it setting a flag up. Then, the main loop knows that needs to execute the event in the next iteration.

The firmware also allows having events triggered by a change in a pin state (external event). This can be used to notify the system if there has been any Latch up (a type of short circuit which can occur in an integrated circuit in space due to radiation) or any other critical event that can occur on the system.

By default, the firmware does not support all the pins to be configured as a trigger, and they should be programmed according to the mission before the launch. This fact is caused by the Arduino microcontroller not being able to configure new pin change interrupts during runtime. This should not constitute a problem, since the pins, which can trigger an event, are known before the launch.

Below is a simplified flow chart of the Arduino firmware:

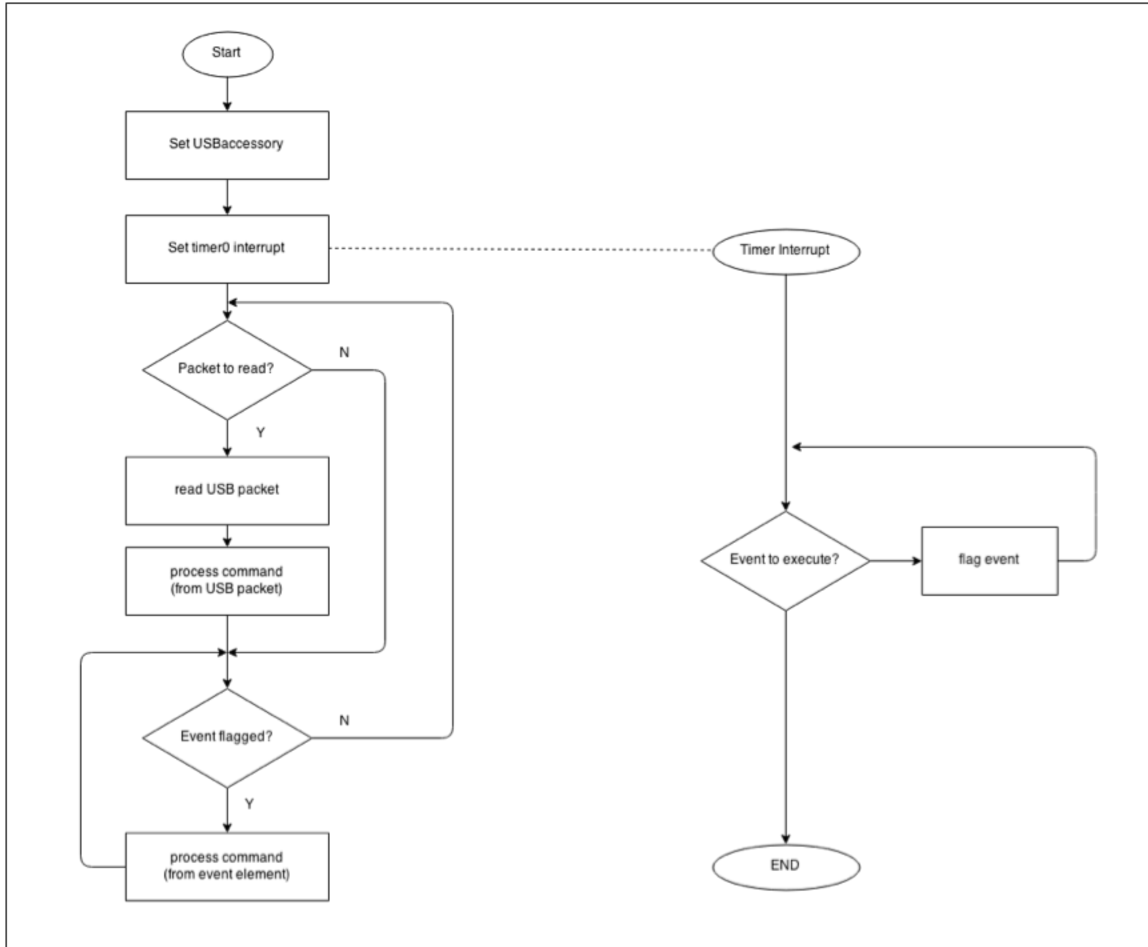


Figure 15: Arduino firmware workflow

A key function here is the process command function. This interprets the packets and executes them. Internally this function is implemented as a switch statement, where it has been programmed what to do for each of the commands. Note that this function is the same for commands and events.

The only difference is that in the first case, the commands to execute are encode in the USB packet and in the events, the commands (actions) are stored in a struct, which every event has, as a byte array. This struct also contains the interval of repetition and the ID of a buffer on the SD for that event.

Those buffers are used to store the data that an event could generate (imagine an event which samples the value of a sensor every 2 seconds). This data can be later retrieve with the command *dumpBuffer*.

The action that an *event* performs is specified in the same command that creates the event itself. This action, which will be decoded by the *execute command* block is encoded inside the data field of the *create event* packet.

The example below shows a packet which creates an event that blinks a LED (toggles a digital PIN) every 0.5 seconds (2 times the basic frequency).

EVENT	CREATE EVENT	1

500 MILLISECONDS	1

-	1

0	1

6	1

BASIC I/O	TOGGLE PIN	1
DIGITAL PIN 4		1
-		1
0		1
0		1
PACKET ID		0

PACKET ID	0

Figure 16: Example of USB packet

The list of actions that an event can perform can be found on Annex 2.

2.4.3. USB communication

The communication between the firmware and the phone is through USB and uses the Android Open Accessory (AOA) protocol [13]. The AOA is used to overcome the problem of connecting to an Android-powered device with an external device and that otherwise would be very complex to do, such as the communication with the Arduino.

When the communication is established, the Android phone enters to what is known as “accessory mode”. In this mode, the Arduino becomes the master and the Android device the slave (accessory). For this to work, the Arduino must have a USB host interface [14].

2.5. App_ctrl library (Appmods)

2.5.1. Introduction

The *App_ctrl* library was originally conceived as a set of processes (called APPmods) in charge of controlling the applications running on top of the Android layer. The Process Manager would be performing this tasks and the *App_ctrl* library would encapsulate the functionalities related with the control of the Android applications, which are the following: install, uninstall, launch and exit a given Android application.

In this scenario the Process Manager performs the monitoring of the Android applications and when some action has to be carried out on these, it calls the corresponding function on the *App_ctrl* library (e.g. if the Process Manager determines that an app needs to be installed, it will call the function *install* on the *App_ctrl* library).

The *App_ctrl* library does not decide which Android application runs at a certain point in time (since this is a job for the Process Manager). However it can ultimately refuse to launch an application if certain requirements are not met.

Imagine an application, which is using a specific pin on the Arduino board that can only be used by one application at a time. Now imagine that the Process Manager requests to launch a second application, which also need to use that same pin. In this case the *App_ctrl* will not launch the app until the resource is freed.

As has been seen in the example, the *App_ctrl* library needs to implement a certain level of control in order to keep track of which resources/permissions are being used by each Android application and if it is necessary act accordingly.

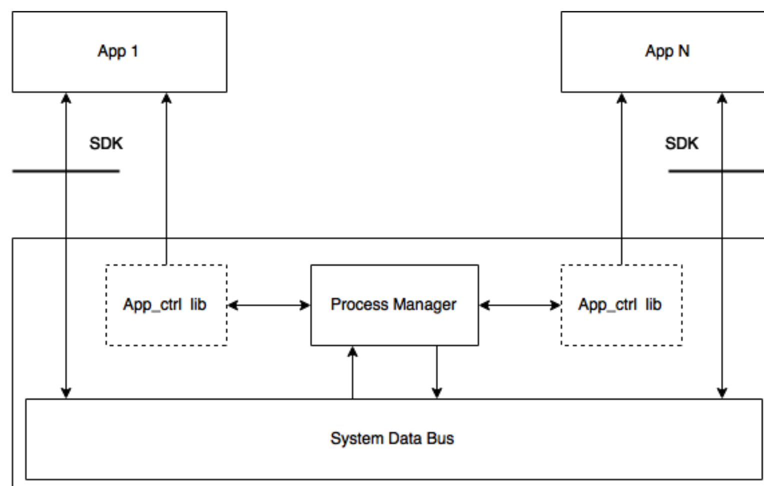


Figure 17 *APP_ctrl* library in context of the architecture

2.5.2. Implementation

As has been said, the *App_ctrl* library needs to keep track of which resources are being used and which will need to be used during runtime. As outlined below, the solution to the problem lies in a file, which all Android apps already have.

Every Android application needs to have an *AndroidManifest.xml* file which presents essential information about the Android application. It has been taken advantage of this fact and it has been used this document to store our own permission.

Below is an example of an *AndroidManifest.xml* from an application, which uses two of our custom permissions (I^2C and SPI communication).

```
<?xml version="1.0" encoding="utf-8"?>
<manifest xmlns:android="..." package="com.abs.payloadapp" >
        <permission android:name="com.i2c" android:label="i2c"></permission>
        <permission android:name="com.spi" android:label="spi"></permission>
        <application>
        …
        </application>
</manifest>
```

Figure 18: Android Manifest with custom permissions

Before executing the app, the *App_ctrl* library will extract the *AndroidManifest.xml* from the Android Application Package (APK) and from this file it will get the list of necessary resources. Next is a diagram with the process.

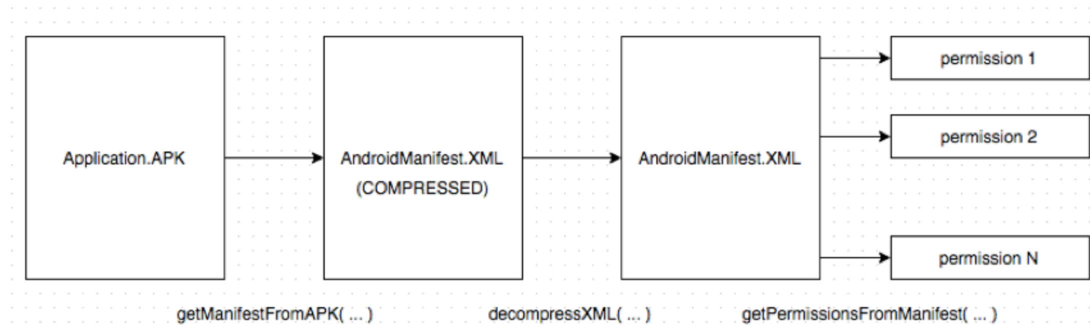


Figure 19: Steps to extract permissions from manifest

Once it has a list of all the resources needed by the app, the *App_ctrl* library will check if any of those resources are already in use. If not, it will update a database, which has searched before for existing permissions, with the new resources and it will launch it. Otherwise, it will send an error code to the controlling module and will not initiate the app.

Likewise, when an application needs to be closed, the *App_ctrl* library will delete those resources that were being used by the app before closing it. In the case of install and uninstall functions, no additional action will be performed.

The functions implemented by the *App_ctr* library are shown below:

```
/* Initiate app_ctrl library */
int init_appctrl();
/* Install a given application */
int app_install(char *filename);
/* Uninstall a given application */
int app_uninstall(char *filename);
/* Launch a given application */
int app_launch(char *filename);
/* Exit a given application */
int app_exit(char *filename);
```

Figure 20: Functions App_ctrl library

Extract the resources needed by the app from its APK was not an easy task. Several solutions can be found (apktools, appt…) [15] which extracts the manifest from an APK and parse it, but they where either too complex or they were not implemented in C. So the chosen solution has been to implement a custom routine from scratch.

To extract the AndroidManifest.xml from the APK it was used the zlib library, which allows us to deflate the APK (Application Package File) and retrieve their files (identical to unpacking a ZIP file). At this point, the AndroidManifest.xml file has been extracted, but needs to be decoded since is in binary format. To do this, it was re-implemented in C a function for Java found on Internet, which did this (decode an XML file). Finally, the file can be parsed and decoded (using regex) and the permissions can be extracted.

2.5.3. Application permission database

The ApplicationPermissions database will keep track of which permissions are being used by each application at a given time. This is being used by the App_ctrl library to decide whether to perform an action over a certain Android app or not.

Table 2 shows the structure for the ApplicationPermissions table:

Column	**Type**	**Description**
Permission_name	Text	Name of the permission
Application_name	Text	Name of the app using the permission
Register_time	Integer	Time which permission (app) starts to be used
Expire_time	Integer	Time which permission (app) stops to be used

Table 5: Basic structure of the ApplicationPermissions table

The library abs_db will provide the following functions:

```
Int addPermission(char *permissionName, char, char *appname, time_t exp_t);

Int deletePermission(char *permissionName);

Int deleteAppPermission( char *appName);

char *findPermission(char *permissionName);
```

Figure 21: Functions App_ctrl abs_db

To create the database presented in this report it has been used SQLite. This database, along with the *App_ctrl* library, has been implemented and tested.

3. Results

In order to verify the functional behaviour of the software implemented, it has been done a series of tests to evaluate the performance of the different parts of the architecture. The tests presented below, try to assess the performance when an app continuously sends a command to the ABS architecture to sense the value of a specific pin on the Arduino.

This test allows us to check the performance of the PayloadApp SDK, System Data Bus, SDB Bus USB and Arduino firmware, since it requires of all this modules to work (see figure below). In this tests it has been measured the latency when one or more apps try to access the same resource at the same time.

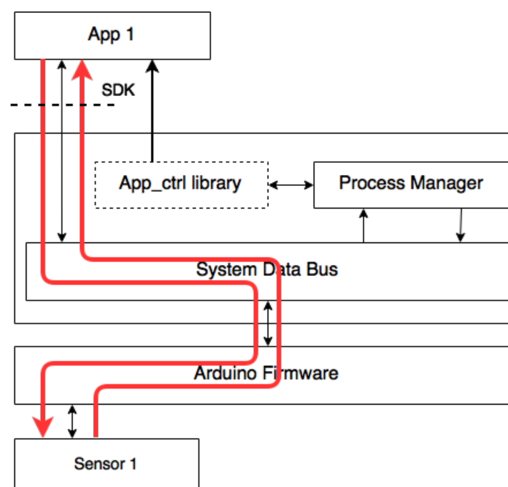
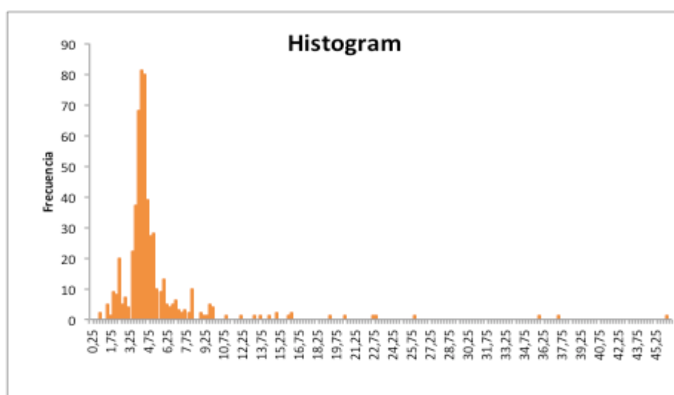


Figure 22: Path followed by the command analogRead

1) Latency test:

In this test, an application call 500 times the command analogRead with an interval of 0 milliseconds between calls, while it measures the time in performing the action. The results showed a mean of 5,09 ms, which it is a moderably good result despite all the round trip. And a standard deviation of 3,12 ms, which is also correct, although it shows that the latencies have some random component. Some errors are also seen (2,20%).



Max value(ms):	46,04
Min value (ms):	0,76
Mean (ms):	5,09
Standard Dev (ms):	3,12
Error %:	2,20%

Figure 23: Latency histogram 1 service

Ideally, the histogram should be a single peak as further to the left as possible, meaning that the time of performing a command is fast and predictable. Finally, there should not be any error and it will need to be found the cause of the errors detected.

2) Stress test:

In this test, multiple services call the same function, while the latency is measured on one of the services and on the Arduino. The test shows that the latency on an app grows exponentially when more services are added, but the time of performing the action on the Arduino remain stable. This is due to the Arduino can only process one packet at a time and when are more apps running, the commands need to wait on the priority queue on the SDB USB to be executed by the Arduino.

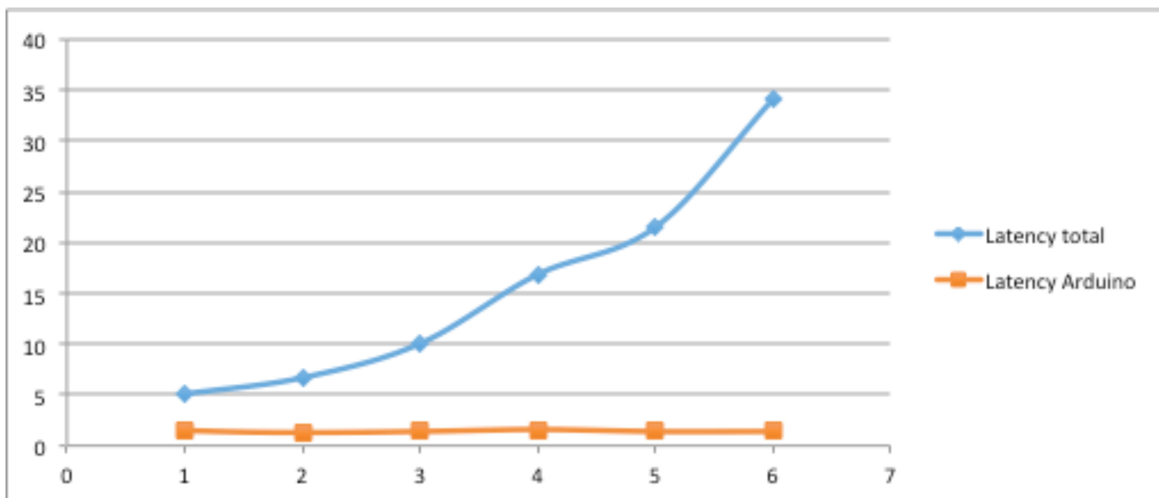


Figure 24: Latency multiple services

Finally, other tests have been executed to check the correct functioning of the different modules implemented. Also, a general-purpose Android application has been developed by a member of the ABS team, Miquel Vaquero, to execute all this tests (the screenshot below shows the main windows panels of this app).

Figure 25: Test application for ABS

4. Conclusions and future extension

This thesis has presented the work conducted on the design and development of part of the software for the ABS Google PhoneSat project. Chapter 1 provided an overview of the ABS project and the objectives of this thesis. Chapter 2 presented the ABS software architecture and its subsystems and it explains in detail the modules implemented. Finally in Chapter 3 some results with the performance of the system were shown.

The purpose of this work has been twofold: on the one hand the design of a software architecture targeted for a nano-satellite system based on an Android phone, along with the implementation and design of three of its modules: Arduino Firmware (interfaces the phone with the attached payloads), System Data Bus (SDB) (interfaces the different modules on the architecture between them), and App_ctrl library (controls the apps running on top of the Android layer.)

On the other hand, this work has shown the design and implementation of an SDK, which will allow for apps being uploaded to the satellite while in orbit. This SDK provides a complete framework and more than 50 functions (commands) giving the bases for in space applications development.

Concluding this report, it is worth mentioning that the objectives specified on the Preliminary Design Review have been fulfilled, and the basis for a fully functional architecture has been established.

Besides this, there are still parts to implement and some of the implemented parts have still some ground for improvement, but the design presented have been build on a very scalable and robust foundation.

A future extension would be to migrate from a single satellite-mission to new mission architectures involving large constellations of nano-satellites. This will allow to create a public access space station (Open Space Station) conformed by swarms of Android-based nano-satellites which will allow the users to interact with its resources through the execution of Android applications.

Bibliography

[1] C. Araguz. *"Towards a modular nano-satellite software platform".* M.S. thesis, Department of Electrical Engineering Universitat Politècnica de Catalunya, Barcelona, Spain, 2014. [Online] Available: http://hdl.handle.net/2099.1/22545. [Accessed: 2 June 2015].

[2] Nasa, "Small Spaceraft Technology State of the Art 2014" [Online] Available: http://www.nasa.gov/sites/default/files/files/Small_Spacecraft_Technology_State_of_the_Art_2014.pdf [Accessed: 2 June 2015].

[3] Wikipedia, "CubeSat" [Online] Available: https://en.wikipedia.org/wiki/CubeSat [Accessed: 2 June 2015].

[4] Wikipedia, "Phonesat" [Online] Available: https://en.wikipedia.org/wiki/PhoneSat [Accessed: 2 June 2015].

[5] J. Guo, J. Chu and E. Gil. *"Onboard autonomy of CubeSat clusters based on smartphone technology".* In 5th International Conference on Spacecraft Formation Flying Mission and Technologies, Munich, 2013.

[6] eoPortal, "Phonesat-1 and 2" [Online] Available: https://directory.eoportal.org/web/eoportal/satellite-missions/p/phonesat-1-2 [Accessed: 2 June 2015].

[7] Ardusat, "Ardusat" [Online] Available: https://www.ardusat.com/ [Accessed: 2 June 2015].

[8] R. Regupathy. *Unboxing Android USB: A hands on approach with real world examples*, 1st ed. New York, USA: Apress, 2014.

[9] Android Dev., "Android NDK" [Online] Available: https://developer.android.com/tools/sdk/ndk/index.html [Accessed: 1 June 2015].

[10] Oracle Java SE documentation, "Java Native Interface 6.0 Specifications - Content" [Online] Available: http://docs.oracle.com/javase/7/docs/technotes/guides/jni/spec/jniTOC.html [Accessed: 1 June 2015].

[11] Android Dev., "Android Studio IDE" [Online] Available: https://developer.android.com/sdk/index.html [Accessed: 2 June 2015].

[12] Android, "AOA Protocol" [Online] Available: https://source.android.com/accessories/protocol.html [Accessed: 2 June 2015].

[13] Android, "Service" [Online] Available: http://developer.android.com/reference/android/app/Service.html [Accessed: 2 June 2015].

[14] Arduino, "Arduino Mega ADK" [Online] Available: http://www.arduino.cc/en/Main/ArduinoBoardMegaADK [Accessed: 2 June 2015].

[15] Apktool, "A tool for reverse engineering Android apk files" [Online] Available: http://ibotpeaches.githb.com/Apktool [Accessed: 2 June 2015].

Annexes

Annex 1: Function list PayloadApp SDK

ARDUINO			
Functions	Arguments	Output	Implemented Ard/And
ARDUINO (BASIC I/O)			
digitalWrite	int pin, int value	int 0 , error	YES/YES
digitalRead	int pin	int [0,1] , error	YES/YES
analogWrite	int pin, int value	int 0 , error	YES/YES
analogRead	int pin	int [0-255] , error	YES/YES
digitalToogle	int pin	int 0 , error	YES/YES
ARDUINO (SERIAL COMUNICATION)			
initi2C	int serial_id [0, MAX_SERIAL],bitrate	int 0 , error	YES/YES
readi2C	int serial_id [0, MAX_SERIAL]	byte[] data, error	YES/YES
writei2C	int serial_id, byte[] data	int 0, error	YES/YES
initSPI	int serial_id [0, MAX_SERIAL],bitrate	int 0, error	YES/YES
readSPI	int serial_id [0, MAX_SERIAL]	byte[] data, error	YES/YES
writeSPI	int serial_id, byte[] data	int 0, error	YES/YES
ARDUINO (PWM/SERVOS)			
startPWM	int pin, int ts	int 0 , error	YES/YES
stopPWM	int pin	int 0 , error	YES/YES
setDC	int pin, int DC	int 0 , error	YES/YES
setTS	int pin, int ts	int 0 , error	YES/YES
ARDUINO (EVENTS)			
createEvent	int interval, Action action, byte[] args	EvHandler, error	YES/YES
(EvHandle) dumpBuffer	NILL	byte[] data, error	YES/YES
(EvHandler) setInterval	int interval	int 0 , error	YES/YES
(EvHandler) setAction	Action action, byte[] args	int 0 , error	YES/YES
(EvHandle) stopEvent	NILL	int 0 , error	YES/YES
ACTIONS			
digitalRead	int pin	//	YES/YES
AnalogRead	int pin	//	YES/YES
digitalToogle	int pin	//	YES/YES
analogSweep	int pin, int step	//	NO/NO
PWMSweep	int pin, int step	//	NO/NO
readi2C	int serial_id	//	YES/YES
writei2C	int serial_id, byte[] data	//	YES/YES
readSPI	int serial_id	//	YES/YES
writeSPI	int serial_id, byte[] data	//	YES/YES
SYSTEM			
ATTITUDE	**getAttitude()**	**Attitude, error**	
getAcc	int axis	float ? , error	NO
getGyro	int axis	float ? , error	NO

ENERGY	getEnergy()	Energy, error	
batteryLevel	NILL	float lv[0-100] , error	NO
batteryStatus	NILL	int status, error	NO
batteryTimeRemaining	NILL	int minutes, error	NO
ORBITAL STATE	**getOrbitalState()**	**OrbitalState, error**	
TLE	NILL	String[] tle, error	NO
latitude	NILL	float latitude, error	NO
longitude	NILL	float longitude, error	NO
ECI	NILL	float[] xyz, error	NO
IGRF	NILL	float[] xyz, error	NO
POWER	**getPower(int pol[])**	**Power[], error**	
getPower	NILL	float power, error	NO
getVoltage	NILL	float voltage, error	NO
getIntensity	NILL	float intensity, error	NO
ARCHITECTURE			
getCPUUse	NILL	int use, error	NO
getBatteryUse	NILL	int use, error	NO
getMemoryUse	NILL	int use, error	NO
ARCHITECTURE			
sndMsgApp	byte[] data, destination	0, error	YES
rcvMsgApp	destination	byte[] data , error	YES

Annex 2: Commands list Arduino Firmware

CMD name	CMD	Parameters	CMD_arg0	CMD_arg1	Data	Response
analog_write	1 (Basic I/O)	0	PIN	value	NILL	ok/error
digital_write	1 (Basic I/O)	1	PIN	value	NILL	ok/error
analog_read	1 (Basic I/O)	2	PIN	NILL	NILL	data/error
digital_read	1 (Basic I/O)	3	PIN	NILL	NILL	data/error
digital_toogle	1 (Basic I/O)	4	PIN	NILL	NILL	ok/error
init i2C	2 (Comms)	0	serial id	bitrate	NILL	ok/error
read i2C	2 (Comms)	1	serial id	NILL	NILL	data/error
write i2C	2 (Comms)	2	serial id	NILL	data	ok/error
initSPI	2 (Comms)	3	serial id	bitrate	NILL	ok/error
readSPI	2 (Comms)	4	serial id	NILL	NILL	data/error
writeSPI	2 (Comms)	5	serial id	NILL	data	ok/error
create	3 (Events)	0	interval	NILL	action	data/error
enable_pin_ev	3 (Events)	1	pin_ev_id	NILL	action	data/error
stop	3 (Events)	s	event id	NILL	NILL	ok/error
set_interval	3 (Events)	3	event id	interval	NILL	ok/error
set_action	3 (Events)	4	event id	NILL	NILL	ok/error
dump_buffer	3 (Events)	5	event id	NILL	NILL	data/error
start	4 (PWM)	0	ts	DC	NILL	data/error
stop	4 (PWM)	1	PWM id	NILL	NILL	ok/error
set DC	4 (PWM)	2	PWM id	DC	NILL	ok/error
ok	0 (control)	0	NILL	NILL	NILL	//
data	0 (control)	1	NILL	NILL	NILL	//
error	0 (control)	2	error	NILL	NILL	//

List of actions

digitalRead	5 (actions)	0	PIN	NILL	NILL	//
AnalogRead	5 (actions)	1	PIN	NILL	NILL	//
digitalToogle	5 (actions)	2	PIN	NILL	NILL	//
analogSweep	5 (actions)	3	PIN	step	NILL	//
PWMSweep	5 (actions)	4	PIN	step	NILL	//
readi2C	5 (actions)	5	serial id	NILL	NILL	//
writei2C	5 (actions)	6	serial id	NILL	data	//
readSPI	5 (actions)	7	serial id	NILL	NILL	//
writeSPI	5 (actions)	8	serial id	NILL	data	//

Glossary

^{3}Cat1 First nano-satellite developed at Tech. Univ. of Catalonia

ABS Android Beyond the Stratosphere

ADK Android Development Kit

AOA Android Open Accessories

API Application Programming Interface

APK Android Application Package

$I^{2}C$ Inter-Integrated Circuits

JNI Java Native Interface

JVM Java Virtual Machine

MCS Modular Command System

NASA National Aeronautics and Space Administration.

NDK Native Development Kit

OS Operating System

RT Real Time

SA Software Architecture

SDB System Data Bus

SDK Software Development Kit

SPI Serial Peripheral Interface Bus

UPC Universitat Politècnica de Catalunya

USB Universal Serial Bus

VM Virtual Machine